Open access • Journal Article • DOI:10.1007/S10619-021-07322-5

Parallel Query Processing in a Polystore — Source link

Pavlos Kranas, Boyan Kolev, Oleksandra Levchenko, Esther Pacitti ...+3 more authors

Institutions: Technical University of Madrid, University of Montpellier

Published on: 03 Feb 2021 - Distributed and Parallel Databases (Springer US)

Topics: Query language, Parallel processing (DSP implementation), Data integration, SQL and Joins

Related papers:

- Parallel Polyglot Query Processing on Heterogeneous Cloud Data Stores with LeanXcale
- Augmented Access for Querying and Exploring a Polystore
- Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue
- CloudMdsQL: querying heterogeneous cloud data stores with a common language
- Integrating Big Data and Relational Data with a Functional SQL-like Query Language

Share this paper:

View more about this paper here: https://typeset.io/papers/parallel-query-processing-in-a-polystore-1updnsiz86

Parallel Query Processing in a Polystore

Pavlos Kranas, Boyan Kolev, Oleksandra Levchenko, Esther Pacitti, Patrick Valduriez, Ricardo Jiménez-Peris, Marta Patiño-Martinez

► To cite this version:

Pavlos Kranas, Boyan Kolev, Oleksandra Levchenko, Esther Pacitti, Patrick Valduriez, et al.. Parallel Query Processing in a Polystore. Distributed and Parallel Databases, Springer, In press, pp.39. 10.1007/s10619-021-07322-5 . lirmm-03148271

HAL Id: lirmm-03148271
https://hal-lirmm.ccsd.cnrs.fr/lirmm-03148271

Submitted on 22 Feb 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Parallel Query Processing in a Polystore

Pavlos Kranas[1,2] · Boyan Kolev[1,3] · Oleksandra Levchenko[3] · Esther Pacitti[3] · Patrick Valduriez[3] · Ricardo Jiménez-Peris[1] · Marta Patiño-Martinez[2]

Abstract The blooming of different data stores has made polystores a major topic in the cloud and big data landscape. As the amount of data grows rapidly, it becomes critical to exploit the inherent parallel processing capabilities of underlying data stores and data processing platforms. To fully achieve this, a polystore should: (i) preserve the expressivity of each data store's native query or scripting language and (ii) leverage a distributed architecture to enable parallel data integration, i.e. joins, on top of parallel retrieval of underlying partitioned datasets.

In this paper, we address these points by: (i) using the polyglot approach of the CloudMdsQL query language that allows native queries to be expressed as inline scripts and combined with SQL statements for ad-hoc integration and (ii) incorporating the approach within the LeanXcale distributed query engine, thus allowing for native scripts to be processed in parallel at data store shards. In addition, (iii) efficient optimization techniques, such as bind join, can take place to improve the performance of selective joins. We evaluate the performance benefits of exploiting parallelism in combination with high expressivity and optimization through our experimental validation.

Keywords Database integration · Heterogeneous databases · Distributed and parallel databases · Polystores · Query languages · Query processing

[1] LeanXcale, Madrid, Spain
{pavlos, bkolev, rjimenez}@leanxcale.com

[2] Distributed Systems Lab at Universidad Politécnica de Madrid, Spain
pavlos.kranas@alumnos.upm.es, mpatino@fi.upm.es

[3] Inria, University of Montpellier, CNRS, LIRMM, France
firstname.lastname@inria.fr

1 Introduction

A major trend in cloud computing and data management is the understanding that there is no "one size fits all" solution [34]. 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 this makes use of the semi-structured nature of documents. And because of the big volumes of data, both databases are partitioned 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 data processing frameworks (such as Spark). And third, the query needs to be expressive enough, so as to combine an SQL subquery (to the relational database) with an arbitrary code in a scripting language (e.g. JavaScript), that requests a dataset from the document database, and another script (e.g. in Python or Scala for Spark), that requests a chain of transformations on the unstructured data from the HDFS log before involving it into relational joins. Existing polystore solutions provide SQL mappings to underlying data objects (document collections, raw files, etc.). 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 address the problem of leveraging the underlying data stores' scripting (querying) mechanisms in combination with parallel data retrieval and joins, as well as optimizability through the use of bind joins.

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 [11, 16, 17, 29, 30, 33] typically respect the autonomy of the underlying data stores and rely on a mediator/wrapper approach to provide mappings between a common data model with a query language and each particular data store's data model. CloudMdsQL [23, 26] with its MFR (map/filter/reduce) extensions [9] 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, 15, 20, 28, 39] 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 [25], 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 aforementioned 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 [19] 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 query engine with intra-query and intra-operator parallelism that operates over a standard SQL interface. We validate the concept with various join queries on four diverse data stores and scripting engines.

This paper is a major extension of [24]. The new material addresses the support of distributed processing platforms such as Apache Spark by enabling the ad-hoc usage of user defined map/filter/reduce (MFR) operators as subqueries, yet allowing for pushing down predicates (e.g. for bind join conditions) and parallel retrieval of intermediate results. We present the major challenges we face in supporting this (Section 3) and introduce extended motivating examples (Section 5.1). We also apply our approach for parallel integration with Spark, together with its architectural and implementation details (Section 5.5). The experimental evaluation has been also extended accordingly, to address an example use case scenario, where unstructured data, stored in HDFS, must go through transformations that require the use of programming techniques like chaining map/reduce operations, which should take place before the data are involved in relational operators. Related work has been extended as well.

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

---

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

2 Related Work

The problem of accessing heterogeneous data sources has long been studied in the context of multidatabase and data integration systems [31, 36]. 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. 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. Polystore systems can be divided between loosely-coupled, tightly-coupled, and hybrid [10], which we discuss briefly later in this section. Since loosely- and tightly-coupled systems address only partially our problem, we will focus in more detail on hybrid systems as the state-of-the-art category where our work fits in. We also add a fourth category of the recent parallel workflow management systems, which provide polystore support.

With respect to **combining SQL and map/reduce** operators, a number of SQL-like query languages have been introduced. HiveQL is the query language of the data warehousing solution Hive, built on top of Hadoop MapReduce [35]. 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 [12] 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 [40]. 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.

Loosely-coupled polystores are reminiscent of multidatabase systems in that they can deal with autonomous data stores, which can then be accessed through the polystore common interface as well as separately through their local API. Most loosely-coupled systems support only read-only queries. Loosely-coupled polystores follow the mediator/wrapper architecture with several data stores (e.g. NoSQL and RDBMS). BigIntegrator [29] 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 [33] 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++ [30] 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 [16, 17] 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. In addition to typical loosely-coupled systems, some polystore solutions [11, 14, 22] consider the problem of optimal data placement and/or selection of data source, mostly driven by application requirements. Estocada [11] is a self-tuning polystore 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. Estocada has been recently extended with a novel approach for efficient cross-model query processing, using queries as materialized views and performing view-based query rewriting [3].

Tightly-coupled polystores have been introduced with the goal of integrating Hadoop or Spark for big data analysis with traditional (parallel) RDBMSs. Tightly-coupled polystores trade autonomy for performance, typically operating in a shared-nothing cluster, taking advantage of massive parallelism. Odyssey [20] enables storing and querying data within HDFS and RDBMS, using opportunistic materialized views. MISO [28] 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 [39] 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 [15] 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.

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.

Spark SQL [6] 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 [32] is a distributed SQL query engine, running on a shared-nothing cluster of machines, and designed to process interactive analytic queries against data sources of any size. Presto follows the classical distributed 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 [4] 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 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.

Myria [38] 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. Non-relational systems, such as MongoDB, are supported by defining relational semantics for their operations and adding rules to translate them properly into the relational algebra, used by Myria's relational algebra compiler RACO.

Impala [5] is an open-source distributed 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.

Parallel workflow management systems is a recent category of solutions for big data processing that aims to decouple applications from underlying data processing platforms. They typically facilitate applications by choosing the best platform to execute a particular workflow or task from a set of available platforms (e.g. Hadoop, Spark,

Giraph, etc.). Musketeer [18] achieves this through a model that breaks the execution of a data processing workflow in three layers: first, the workflow is specified using a front-end framework of user's choice, e.g. SQL-like query or vertex-centric graphic abstraction; then, the specification is transformed into an internal representation in the form of a data-flow DAG; and finally, code is generated to execute the workflow against the target platform. This saves the tedious work of manually porting workflows across platforms in case some platform is found to be better suited for a particular workflow. More recently, RHEEM [2] enhanced the concept by allowing a particular subtask of the workflow to be assigned to a specific platform, in order to minimize the overall cost. It also introduces a novel cost-based cross-platform optimizer [27] that finds the most efficient platform for a task and an executor that orchestrates tasks over different platforms with intermediate data movement. Thus, RHEEM can integrate data from different data stores (hence act as a polystore) by assigning different operators from the query plan to different engines, e.g., perform selections on base tables and associated joins at the RDBMS to exploit indexes, then ship intermediate data and perform other joins at Spark to exploit parallelism. Teradata IntelliSphere [7] addresses the problem of accessing multiple data stores (called "remote systems") that may be heterogeneous, but must have an SQL-like interface. Each remote system, however, can be a polystore by itself. Teradata is responsible for building an SQL query plan and deciding where each SQL operator (e.g. join or aggregation) will execute on one of the IntelliSphere's systems (either Teradata or a remote system). An important problem the system focuses on is the cost estimation of SQL operators over remote systems. Machine learning techniques are leveraged to train neural networks to approximate the execution time of an operator based on characteristics of its input relation(s).

Comparative analysis. Table 1 summarizes the functionality of polystore systems that enable parallel processing across diverse DBMS clusters (we will use for brevity the term "parallel polystores"). We compare the systems with respect to the features that we hereby address, mainly the parallel support of MongoDB, data processing platforms, and optimizability of selective joins through semi joins across data stores. We exclude loosely-coupled systems, as they do not provide parallel data integration. Workflow managers dispatch the execution of a query/workflow plan to underlying data processing platforms, hence can access MongoDB through the platforms, e.g., Spark using the Spark MongoDB connector. Tightly-coupled systems can perform parallel joins, but since they are focused only on the tight integration between RDBMS and Hadoop stores, cannot be extended to support NoSQL stores. Hybrid systems (besides LeanXcale) usually access document data stores through extended relational mappings, with the added support of flattening operators (UNNEST) to express queries over nested documents. With respect to semi joins across data stores, only a few of the systems are capable. JEN and LeanXcale are applying semi-joins as an optimization technique – JEN with its efficient zigzag join that exchanges bloom filters between the HDFS and RDBMS datasets and LeanXcale through bind joins. With other systems, semi-joins may be supported, but must be explicitly programmed.

Table 1. Comparison of parallel polystores.

System	Query interface	Supported data stores	Supports document databases (MongoDB)	Supports data processing platforms	Cross-platform semi joins	Extensible
Tightly-coupled systems						
Polybase	SQL	RDBMS, HDFS	N	N	N	N
HadoopDB	SQL-like (Hive QL)	RDBMS, HDFS	N	N	N	N
Odyssey/Miso	SQL	RDBMS, HDFS	N	N	N	N
JEN	SQL	RDBMS, HDFS	N	N	Zig-zag bloom join	N
Workflow managers						
Musketeer	SQL-like + Graph queries; extensible	RDBMS, HDFS, NoSQL	Through underlying platform, e.g. Spark	Multiple, incl. Spark	N	Y
RHEEM	RheemLatin (imperative)	RDBMS, HDFS, NoSQL	Through underlying platform, e.g. Spark	Multiple, incl. Spark	Explicitly programmed	Y
Teradata IntelliSphere	SQL-like	RDBMS, HDFS, NoSQL	Through another SQL-like polystore	Any, with SQL-like interface	N	Y
Hybrid polystores						
SparkSQL	SQL-like	RDBMS, HDFS, NoSQL	Relational mappings	Spark, natively	Explicitly programmed	Y
Presto	SQL-like	RDBMS, HDFS, NoSQL	Relational mappings	N	N	Y
Apache Drill	SQL-like	RDBMS, HDFS, NoSQL	Relational mappings	N	N	Y
Myria	MyriaL (relational model)	RDBMS, HDFS, NoSQL	Relational mappings	N	N	Y
Impala	SQL	RDBMS, HDFS, NoSQL	Relational mappings	N	N	Y
LeanXcale [24]	SQL-like + native queries	RDBMS, HDFS, NoSQL	Native JavaScript	N	Bind join optimization	Y
LeanXcale [this]	SQL-like + native queries + MFR	RDBMS, HDFS, NoSQL	Native JavaScript	Spark, through native Scala or MFR	Bind join optimization	Y

Concluding remarks. Although these systems enable parallel integration with data clusters (like MongoDB), none of them supports the combination of massive parallelism with native queries and the optimization of bind joins, which is addressed by the LeanXcale distributed query engine. In particular, the parallel query processing with bind joins through SQL queries is not supported by any of the hybrid systems. For example, with Spark SQL it is possible to do a bind join, but this must be defined programmatically by a developer. This, however, limits the use cases, since a data analyst cannot easily take advantage of this feature through an SQL query. With LeanXcale, once the named tables (subqueries to data stores) are defined by the system developer or administrator, they can be easily used and involved in joins (including bind joins) through the SQL interface. Moreover, enabling native queries and scripts allows to fully exploit the power of the underlying data stores, as opposed to using static mappings to a common data model.

3 Motivation and Problem Statement

Existing parallel polystore query engines [4, 5, 6, 32, 38] address the problem of accessing in parallel partitions from tables, document collections, or arbitrary distributed datasets, exploiting the parallel capabilities of a diverse set of underlying distributed data stores and performing parallel joins on top. This is typically done by making a number of query engine workers connect independently to data store shards. As an example, elaborated in our previous work [24], we addressed the scenario of joining a sharded document collection, residing in a MongoDB cluster, with a partitioned table from a distributed relational database or a distributed HDFS dataset. This setup works well when the underlying datasets can be logically mapped to relations, so that joins can efficiently take place at the parallel polystore query engine. Even if the MongoDB data has to undergo transformations, expressed through user-defined JavaScript functions, this can still be handled in parallel by making each worker initiate the execution of the custom JavaScript code against the MongoDB shard assigned to it and collect its partition of the intermediate data.

In other cases, complex transformations may need to be applied to a distributed dataset (e.g. through specific map-reduce blocks that have no analogue in relational algebra terms), before the data can be processed by means of relational operators. This problem was addressed in [9], by allowing distributed data processing frameworks (in particular Apache Spark) to be accessed as data stores and queried through the semi-declarative MFR notation. To achieve this, the query engine creates a session at the Spark driver, then translates the MFR subquery to code (in Scala or Python for Spark), delegates this code to Spark for execution, and collects the intermediate data through the same Spark driver session.

The limitation. However, the collection of this intermediate result set is centralized, since the Spark driver simply merges the data from all the partitions of the final RDD into a single non-partitioned result set. Thus, even a distributed query engine cannot exploit parallelism in retrieving a Spark RDD, since only one worker will collect the entire RDD through the Spark driver, which is the limitation we want to overcome in this paper as an extended version of [24].

The challenge. This limitation comes from the fact that the query engine, like in most, if not all, parallel polystores, is designed so that each of the involved parallel workers *initiates* connection to a node of the underlying data management system and *pulls* its partition of a dataset. However, in the case of Spark, there is no way to directly access the data of an RDD partition. Therefore, the query engine would be forced to use a single worker to retrieve the entire RDD through the Spark driver, serially. We address this problem by introducing an architecture, where each RDD partition (more precisely, the Spark worker that processes the partition) is instructed through generated code to find and connect to a query engine worker and to *push* the partition data. Doing this in parallel and uniformly across Spark and query engine workers is the major challenge of the current extension of our work.

Objectives. We can now summarize the objectives of our work as the following requirements to our distributed query engine (DQE):

- Parallel data processing: DQE parallelizes the execution of every relational operator.
- Parallel retrieval from data stores: DQE workers access independently data store shards to retrieve partitioned data in parallel.
- Autonomy of data stores: DQE does not rely on full control over the data stores; they can be used independently by other applications.
- Highly expressive queries: adopt he polyglot approach of the CloudMdsQL query language to allow data store native queries or scripts to be expressed as inline subqueries.
- Optimizability: incorporate optimization techniques, such as bind join and MFR rewrite rules (see Section 4.2) to boost the performance of selective queries.
- Extensibility: allow for other parallel data stores to be added by implementing adapters through a flexible programming interface (DataLake API, see Section 5.2).
- Parallel data push model: data store shards can connect to DQE workers independently to push partitioned data in parallel. This allows for distributed data processing frameworks, such as Spark, to be supported by the DQE.

While most of these requirements have already been addressed by other systems in the literature, we emphasize on the combination of all of them and in this extended version particularly pay attention to the last one.

Comparisons. To choose competitor systems to experimentally compare our solution with, we followed the criteria determined by our evaluation queries, i.e. parallel joins between a sharded document collection and the partitioned result of a workflow query to a distributed data processing platform. In particular, we want to stress on the full parallelism to access the underlying datasets, in our case, a MongoDB collection and a Spark RDD, in the context of expressive subqueries. Considering our comparative analysis (Table 1), we exclude tightly-coupled systems since they do not support document stores. Among hybrid systems, although Presto and Drill support well parallel query processing across SQL and NoSQL stores, the only one that provides parallel support of distributed data platforms is Spark SQL, as it uses Spark natively. As for workflow managers, although they can orchestrate efficiently relational operators across platforms, they do not provide query execution themselves; for example, a parallel join between MongoDB and Spark would be dispatched for execution at Spark by both Musketeer and RHEEM, so we would consider this comparison as equivalent to comparing with Spark. Therefore, we target Spark SQL as the only relevant system to evaluate our contributions against.

4 Background

This section presents the concepts we consider towards the design our solution. We start with an overview of the CloudMdsQL query language focusing on the polyglot capabilities and optimization techniques. Then we discuss the distributed architecture of the LeanXcale query engine, which we use to enable the parallel capabilities of our polystore.

4.1 CloudMdsQL Query Language

The CloudMdsQL language [26] 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 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 API call or JavaScript code. The subquery of expression `T1` is subject to rewriting by pushing into it the filter condition `y <= 3`, to increase efficiency.

CloudMdsQL also provides a `CREATE NAMED EXPRESSION` command that allows an expression to be defined and stored in a global catalog in order to be referenced in several queries, similarly to SQL views and stored procedures/functions. This can facilitate the work of data analysts who just need to run SQL queries on predefined views over the underlying data stores, without the need to deeply understand the specifics of the data technologies and data organization.

MFR extensions. To address distributed processing frameworks (such as Apache Spark) as data stores, CloudMdsQL introduces a formal notation that enables the ad-hoc usage of user-defined map/filter/reduce (MFR) operators as subqueries to request data processing in an underlying big data processing framework [9]. An MFR statement represents a sequence of MFR operations on datasets. 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. 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 text file containing a list of words. To count the words that contain the string 'cloud', we write the following composition of MFR operations:

```
T4(word string, count int)@spark = {*
   SCAN(TEXT, 'words.txt')
  .MAP(KEY, 1)
  .REDUCE(SUM)
  .FILTER( KEY LIKE '%cloud%' )
*}
```

For defining map and filter expressions, the special variable `TUPLE` can be used, which refers to the entire tuple. The variables `KEY` and `VALUE` are thus simply aliases to `TUPLE[0]` and `TUPLE[1]` respectively.

To optimize this MFR subquery, the sequence is a subject to rewriting according to rules based on the algebraic properties of the MFR operators [9]. In the example above, since the `FILTER` predicate involves only the `KEY`, it can be swapped with the `REDUCE`, thus allowing the filter to be applied earlier in order to avoid unnecessary and expensive computation. The same rules apply for any pushed down predicates, including bind join conditions.

4.2 Optimizations

To provide an optimal execution of selective queries, we consider two optimization opportunities: bind join and MFR rewrite rules.

Bind join is a join method, in which the intermediate results of the outer relation (more precisely, the values of the join key) are passed to the subquery of the inner side, which uses these results to filter the data it returns. If the intermediate results are small and index is available on the join key at the inner side, bindings can significantly reduce the work done by the data store. [19]

To provide bind join as an efficient method for performing semi-joins across heterogeneous data stores, CloudMdsQL 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 (outer relation), is passed as a filter to the right-hand side (inner) 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
```

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 $b_1 \dots b_n$, 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 counterpart 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.

```
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 $b_1, \dots, b_n$ 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).

Using bind join can be subject to planning decision. 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 left-hand side `B` to be fully retrieved before initiating the execution of the right-hand side `A`. 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 subquery `A`. To take this decision, the optimizer needs at least to estimate the cardinality of the join keys of `B`, which can be easily solved if the data store exposes a cost model. However, if `B` is a native query or no cost information is available, the decision can still be taken, but at runtime: bind join is attempted and the retrieval of `B` initiated; then, if at some point the number of join keys exceeds a threshold, the execution falls back to an ordinary hash join. Nevertheless, the usage of bind join can be also explicitly requested by the user through the keyword `BIND` (e.g. `FROM b BIND JOIN a`).

MFR rewrite rules are used to optimize an MFR subquery after a selection pushdown takes place. The goal in general is to make filters take place as early as possible in the MFR sequence.

Rule #1 (name substitution): upon selection pushdown, an MFR `FILTER` is appended to the MFR sequence and the filter predicate expression is rewritten by substituting column names with references to dataset fields as per the mapping defined through the MFR expression. After this initial inclusion, other rules apply to determine whether it can be moved even farther. Example:

```
T1(a int, b int)@db1 ={* … *}
SELECT a, b FROM T1 WHERE a > b
```

is rewritten to:

```
T1(a int, b int)@db1 ={* … .FILTER(KEY > VALUE)*}
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.

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. The greatest advantage of these rules can be observed when Rule #2 is applicable, as it enables early filtering of the input to expensive REDUCE operators. MFR rewrites can be combined with bind join in the sense that when a bind join condition is pushed down the MFR subquery, it will be applied as early as possible, in many cases reducing significantly the work done by REDUCE operators on the way.

4.3 LeanXcale Architecture Overview

LeanXcale is a scalable distributed SQL database management system with OLTP and OLAP support and full ACID capabilities. It has three main subsystems: the query engine, the transactional engine, and the storage engine, all three distributed and highly scalable (i.e., to hundreds of nodes). The system applies the principles of Hybrid Transactional and Analytical Processing (HTAP) and addresses the hard problem of scaling out transactions in mixed operational and analytical workloads over big data, possibly coming from different data stores (HDFS, SQL, NoSQL, etc.). LeanXcale solves this problem through its patented technology for scalable transaction processing [21]. The transactional engine provides snapshot isolation by scaling out its components, the ensemble of which guarantees all ACID properties: local transaction managers (atomicity), conflict managers (isolation of writes), snapshot servers (isolation of reads), and transaction loggers (durability).

The LeanXcale database has derived its OLAP query engine from Apache Calcite [8], a Java-based open-source framework for SQL query processing and optimization. LeanXcale's distributed query engine (DQE) is designed to process OLAP workloads over the operational data, so that analytical queries are answered over real-time data. This 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 [13], 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.

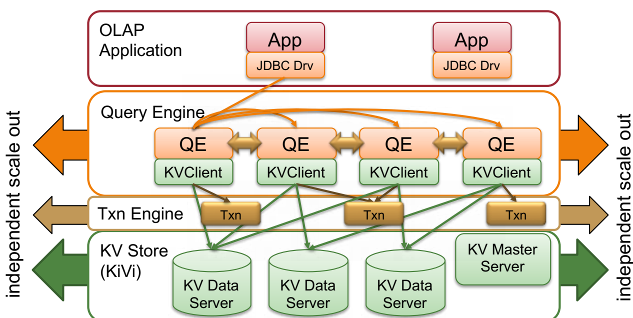

Fig. 1. DQE distributed architecture

Fig. 1 illustrates the architecture of LeanXcale's Distributed Query Engine (DQE). Applications connect to one of the multiple DQE instances 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 are 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 partition corresponds to a range of the primary/index key and it is the unit of distribution. 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). KiVi is relational in the sense that it has a relational schema and implements all relational operators but join, so any relational operator below a join can be pushed down to KiVi.

This architecture scales by allowing analytical queries to execute in parallel, based on the master-worker model using *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 TCP connection to the worker. Each TCP connection is initialized as a worker, creating a communication endpoint 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.

The master includes a state-of-the-art [31] query optimizer that transforms a query into a *parallel execution plan*. The transformation made by the optimizer 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 scheduling the obtained subsets to the different DQE instances. This scheduling is handled by a component in the master worker, named *DQE scheduler*. The generated parallel execution plan is broadcast to be processed by all workers. 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 [37] schema.

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

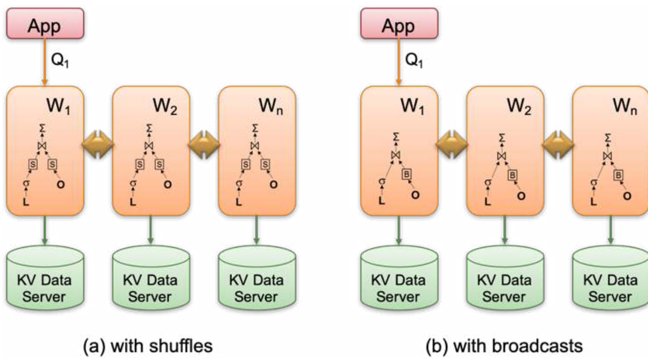

Fig. 2. Query processing in parallel mode

This query is transformed 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 (Fig. 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).

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 (Fig. 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.

5 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 data management clusters 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) and parallel data processing frameworks (such as Apache Spark, where data retrieval and/or transformation can be requested by means of Python or Scala scripting). 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 across a relational table, the result of a JavaScript subquery to MongoDB, and the result of an MFR/Scala subquery to Apache Spark, 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.

5.1 High Expressivity: Motivation

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.",
          keywords: ["data", "query", "cloud"]},
  {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,
           keywords 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,
                keywords:i.keywords});
    } );
    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.

In another example, a more sophisticated data transformation logic (such as a chain of user-defined transformations over Apache Spark RDDs) needs to be applied to unstructured data before processing by means of relational operators [9]. Let us consider the task of analyzing the logs of a scientific forum in order to identify the top experts for particular keywords, assuming that the most influencing user for a given keyword is the one who mentions the keyword most frequently in their posts. We assume that the forum application keeps log data about its posts in the non-tabular structure 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, link to the post, and username in the example) followed by a variable number of fields storing the keywords mentioned in the post.

```
2014-12-13, http://..., alice, storage, cloud
2014-12-22, http://..., bob, cloud, virtual, app
2014-12-24, http://..., alice, cloud
```

The unstructured log data needs to be transformed into the tabular dataset below, containing for each keyword the expert who mentioned it most frequently.

KW	**expert**	**frequency**
cloud	alice	2
storage	alice	1
virtual	bob	1
app	bob	1

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. This can be expressed with the following MFR subquery, with embedded Scala lambda functions to define custom transformation logic:

```
Experts(kw string, expert string)@spark = {*
  SCAN( TEXT, 'posts.txt', ',' )
  .MAP( tup=> (tup(2), tup.slice(3, tup.length)) )
  .FLAT_MAP( tup=> tup._2.map((_, tup._1)) )
  .MAP( TUPLE, 1 )
  .REDUCE( SUM )
  .MAP( KEY[0], (KEY[1], VALUE) )
  .REDUCE( (a, b) => if (b._2 > a._2) b else a )
```

```
  .MAP( KEY, VALUE[0] )
*}
```

In this sequence of operations, the first `MAP` takes a tuple (corresponding to a row from the input file) as an array of string values (`tup`) and maps the username (`tup(2)`) to the keywords subarray (`tup.slice(…)`). Then, the `FLAT_MAP` emits (keyword, user) pairs for each keyword. The following `MAP` and `REDUCE` count the frequencies of each such pair. Then, after grouping by keyword, the last `REDUCE` selects, for each keyword, the (user, frequency) pair that has the greatest value of frequency. The final `MAP` selects the keyword and username for the final projection of the returned relation.

Now, the named table `Experts` can be joined to `BookOrders`, for example the following way:

```
SELECT B.title, B.author, E.kw, E.expert
FROM BookOrders B, Experts E
WHERE E.kw IN B.keywords
```

Optimization. For optimal execution of this query, both *bind join* and *MFR rewrites* play their roles. The bind join condition (which involves only the `kw` column) can be pushed down the MFR sequence as a `FILTER` operator, in this case `FILTER(KEY IN (<set_of_B_keywords>))`. As per the MFR rewrite rules, this would take place immediately after the `FLAT_MAP` operator, significantly reducing at early stage the amount of data to be processed by the expensive `REDUCE` operators that follow. To build the bind join condition, the query engine flattens `B.keywords` and identifies the list of distinct values.

5.2 DataLake API

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.

Towards the design of a distributed wrapper architecture and its programming interface, we consider the outlined in Section 3 requirements for our polystore. In particular, we pay attention to the following desired capabilities:

- A DataLake wrapper must have multiple instances, each linked to a DQE worker.
- A wrapper instance must be able to execute a native subquery or script against a particular shard of the underlying data store cluster.
- The DQE scheduler must be able to retrieve (through one of the wrapper instances) a list of "shard entries", i.e. specifications of the available shards for an underlying dataset. These specifications must be opaque, as the DQE scheduler is agnostic to the specifics of the data store cluster.
- The scheduler must be able to assign shards to DQE workers and hence to the corresponding DataLake wrapper instances.

These requirements make the concept generic in the sense that our polystore can be easily extended to support any data management cluster as long as it provides means to connect directly to database shards and retrieve in parallel dataset partitions. In the subsequent subsections, we give details on how the process of parallel retrieval from MongoDB and HDFS datasets is mapped to the methods of the generic DataLake API. We also show that the same methods abstract well enough even the more sophisticated parallel data push model, necessary to support the parallel integration with Apache Spark, as introduced in Section 3.

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 (Fig. 3).

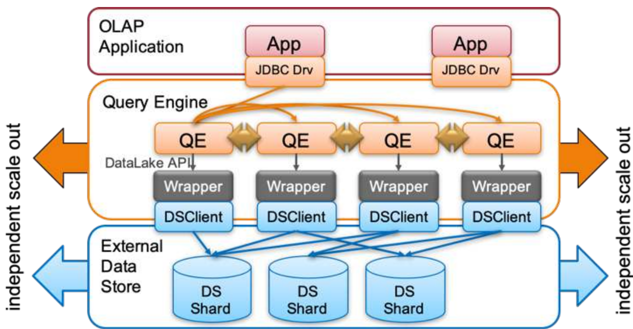

Fig. 3. Generic architecture extension for accessing external data stores

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 opaque 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.

5.3 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. 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, by means of 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 a cluster of equal numbers of: DQE workers, KVDS servers, and MongoDB shards. Thus, each DQE worker gets exactly one partition of both tables by connecting to one MongoDB shard (through a wrapper instance) and one KVDS (Fig. 4).

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`.

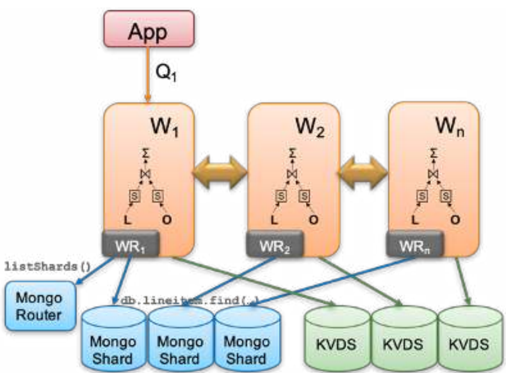

Fig. 4. Parallel join between sharded datasets: LeanXcale table and MongoDB collection.

5.4 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
```

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 (Fig. 5).

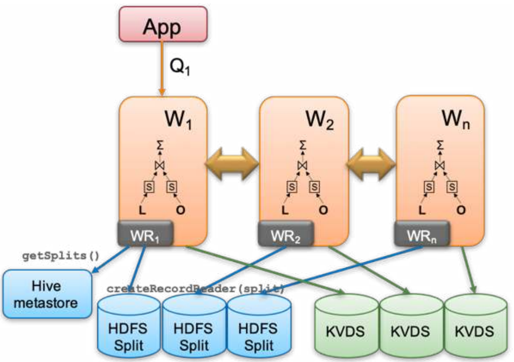

Fig. 5. Parallel join between LeanXcale and HDFS tables.

5.5 Implementation for Apache Spark

As stated in Section 3, the major challenge of supporting Apache Spark as an underlying data management cluster is to enable parallel data movement from Spark workers to DataLake wrappers. Since this necessitates that Spark workers find and connect to DataLake wrapper instances, it results in a different, more complex architecture of the distributed Spark wrapper. A discovery service is introduced through the special component *Spark Agent Registry* that keeps information about available Spark wrapper instances and dispatches them to the requesting Spark workers so that parallelism is fully exploited in moving data from a Spark RDD to the DQE. In order to make the wrapper instances collect in parallel partitions of the resulting RDD, the master wrapper ships an additional code together with the user defined script, that makes each RDD partition push its data directly to the assigned by the registry wrapper instance. This approach, explained in detail hereafter, differs from the typical retrieval of sharded data, where partitions of the underlying dataset can be directly accessed and pulled by wrapper instances.

As the wrapper processes MFR expressions wrapped in native subqueries, it implements a subquery processor. It parses and interprets a subquery written in MFR notation; then, uses an MFR planner to find optimization opportunities; and finally translates the resulting sequence of MFR operations to a sequence of Spark methods to be executed, expressed as Scala (in our focus for this paper) or Python script. The MFR planner decides where to position the pushed down filter operations to apply them as early as possible, using rules for reordering MFR operators that take into account their algebraic properties (see Section 4.2). This preprocessing takes place at the master wrapper instance.

To enable remote submission of the generated Scala script for Spark by the master wrapper, our setup relies on Apache Livy[2], which provides a REST service for easy submission of Spark jobs and Spark context management. Fig. 6 gives a high-level illustration of the processing of the query Q_1^{SL}, assuming a simple MFR subquery that

[2] https://livy.apache.org

reads the LINEITEM table as a text file from the Hadoop cluster, but this time through Spark.

```
Q1SL: LINEITEM(L_ORDERKEY int, …)@spark = {*
        {* SCAN(TEXT, 'lineitem.tbl', ',') *}
    SELECT count(*)
    FROM LINEITEM L, ORDERS O
    WHERE L_ORDERKEY = O_ORDERKEY
```

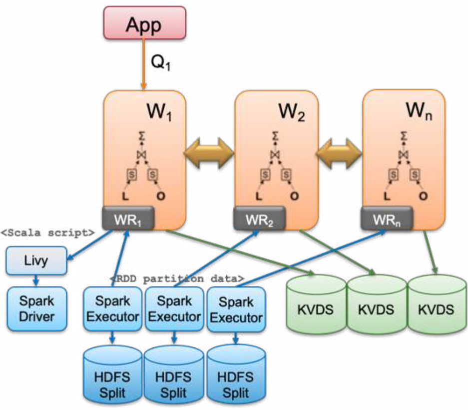

Fig. 6. Parallel join between LeanXcale and Spark.

Fig. 7 shows in detail the flow of operations for processing an MFR subquery by the distributed Spark wrapper. Each wrapper instance is composed of *PrepareScript* and *SparkAgent* components. *PrepareScript* is responsible for preparing the Scala script to be submitted as a Spark job and is active only at the master wrapper for a particular query. *SparkAgent* is the component, which accepts TCP connections from Spark executors to push RDD partition data. To initiate the subquery processing, the DQE sends the user MFR expression to the master wrapper through a call to the `init()` method. Then, the *PrepareScript* component of the master wrapper generates the Scala code that corresponds to the MFR query, to initialize a variable named `rdd`:

```
val rdd = sc.textFile( "lineitem.tbl" )
         .map( _.split(",") )
```

Next, the DQE calls the `listShards()` method of the master wrapper, which returns the number of expected partitions of the result RDD. To figure out this number, *PrepareScript* opens a Livy session, initializes the `rdd` variable using the above Scala statement, and then calls `rdd.getNumPartitions()`.

At this moment, the execution of the prepared Spark job gets initiated by calling through the same Livy session the following `foreachPartition` action function that makes each partition connect to an available wrapper instance and send its data:

```
rdd.foreachPartition { part =>
  val sock = connectSparkAgent()
  part.foreach( tup=> sock.write(serialize(tup)) )
  sock.close()
}
```

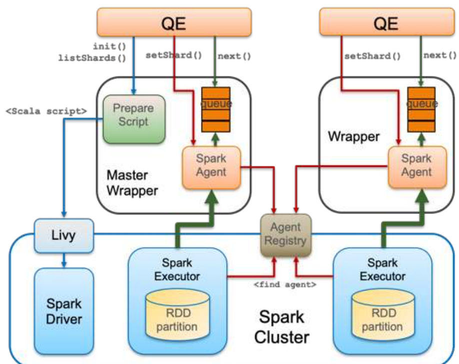

Fig. 7. Architecture of the distributed Spark wrapper.

In this code, `connectSparkAgent` is a function that the master wrapper preliminarily generates and defines in the Livy session. It requests from a common component, named *AgentRegistry*, the address of an available *SparkAgent* (waiting for such availability, if necessary) and makes a socket connection to it. `serialize` is another function that serializes each entry of the RDD partition to a byte array in a format that *SparkAgent* can interpret, which is then sent to the *SparkAgent* through the socket. This function is also generated by the master wrapper, once the type of the RDD entries is reported back through Livy after initialization of the `rdd` variable.

Upon a subsequent call of `setShard()` to a wrapper instance, the corresponding *SparkAgent* reports to the *AgentRegistry* its availability to receive partition data for this particular query. On the other hand, as described above, when processing a partition, each Spark executor finds and sends to an available Spark agent all tuples of the partition. When tuples are received and deserialized, *SparkAgent* buffers them to a queue, from where they are pulled by the query engine through calls of the `next()` method of the wrapper instance.

6 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[3]. Each node in the cluster runs on two Xeon E5-2630 v3 CPUs at 2.4GHz, 8 physical

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

cores per CPU (i.e., 16 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 32 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 and Spark workers are collocated at each node. The coordinating components for the Spark subsystem, Livy and AgentRegistry, are running on one of the nodes. 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.

We performed our experiments in three general groups of test cases, each having a distinct objective. The first group evaluates the scalability of the system in the context of straightforward SQL mappings with the underlying data stores. The second group adds higher expressivity to the subqueries, which cannot be easily achieved through trivial SQL mappings, while still assessing the scalability. The third group evaluates the benefit of performing bind join in the context of large-scale data and the same highly expressive subqueries. All the queries were run on a cluster of LeanXcale DQE instances, running the distributed wrappers for MongoDB, Hive, and Spark.

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. The choice of Spark SQL for a state-of-the-art representative to compare our work with is justified by the fact that it supports most of the features our approach targets and hereby evaluates, namely: (a) parallel MongoDB subqueries through the use of the MongoDB connector that also supports native MongoDB operators (e.g. aggregation pipelines), beyond the trivial SQL mappings; (b) parallel map/filter/reduce subqueries, done natively through Spark RDD transformations; (c) parallel joins and scalability. What Spark SQL is not capable of is bind join through SQL queries; to perform a bind join, one has to write a Spark program, which limits the use cases. Therefore, we stress on our advantage of supporting this powerful optimization technique.

6.1 General Scalability

The first group of test cases aims at generic evaluation of the performance and scalability of joins across any pair of the four involved data stores. The data used was based on the TPC-H benchmark schema [37], particularly for the tables LINEITEM, ORDERS, and CUSTOMER. All the generated datasets were: loaded in LeanXcale as relational tables; loaded in MongoDB as document collections; copied to the HDFS cluster as raw CSV files, to be accessed through Hive as tables and through Spark by means of scans expressed as simple MFR/Scala statements. 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. MFR subqueries to Spark are defined as single SCAN operators, translated to Scala commands. Intermediate result sets from MongoDB, HDFS, and Spark are retrieved in parallel, as described in Section 5.

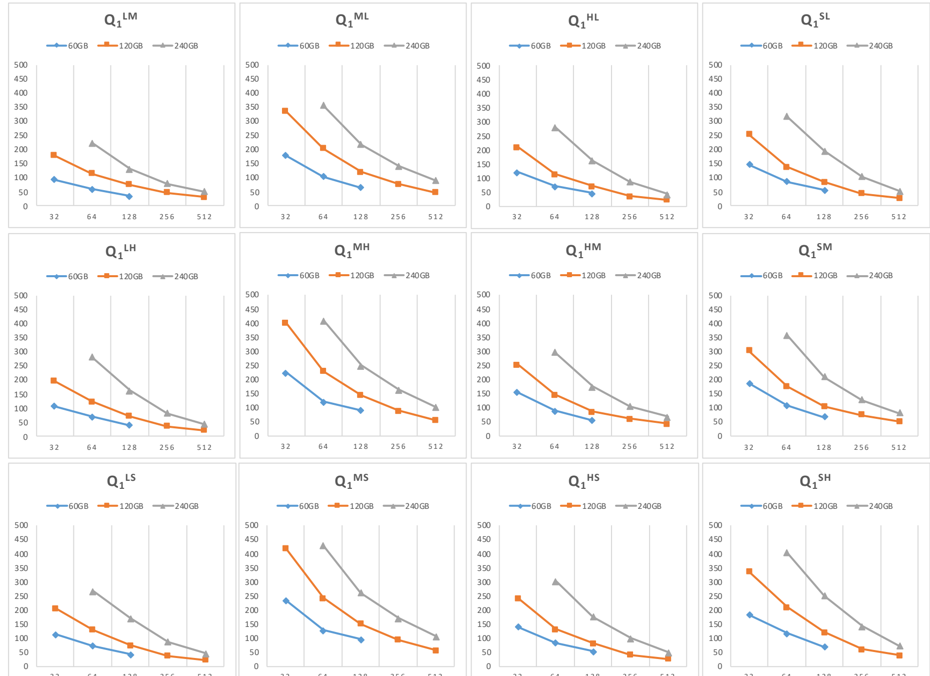

Fig. 8. Execution times (in seconds) of Q_1 queries on TPC-H data with different scales of data (60, 120, and 240 GB) and different levels of parallelism (32, 64, 128, 256, and 512 workers).

Fig. 8 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. For Spark result sets, this overhead is a bit higher, because of the additional

serialization/deserialization that takes place between Spark executors and SparkAgent instances. MongoDB subqueries show lowest performance as data retrieval passes through the embedded JavaScript interpreter at each worker.

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.

6.2 High Expressivity and Scalability

The second group of test cases aims at the evaluation of highly expressive JavaScript and MFR subqueries, such as the `BookOrders` and `Experts` examples from Section 5.1. The goal is to show that even with more sophisticated subqueries, scalability is not compromised.

To evaluate `BookOrders`, 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,
           keywords 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, keywords: i.keywords});
    } );
    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 in MongoDB, while Q_2^{ML} involves a join between MongoDB and 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
```

Fig. 9 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.

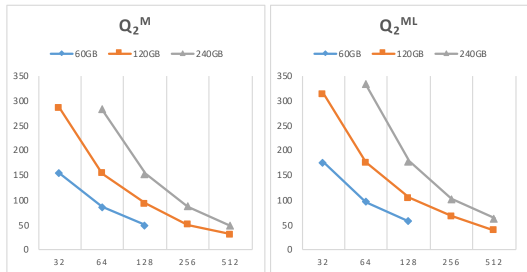

Fig. 9. Execution times (in seconds) of Q_2 queries on more sophisticated JavaScript MongoDB subqueries with scales of data from 60 to 240 GB and levels of parallelism from 32 to 512.

To evaluate the `Experts` subquery, we generated a log file for posts with the structure suggested in Section 5.1. The first and the second fields of each tuple are a timestamp and a URL; they do not have impact on the experimental results. The third 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 a set of 15,000 distinct keywords. Data have been generated in three scale factors: 60GB (400 million tuples), 120GB (800 million tuples), and 240GB (1.6 billion tuples). The intermediate datasets at the first shuffle, where (keyword, user) pairs are emitted, are about the same size respectively.

Similarly to the previous test case, we ran two queries varying the level of parallelism from 32 to 512. Query Q_2^S evaluates just the parallel execution of the `Experts` MFR query on Spark, while Q_2^{SL} involves a join with the `CUSTOMER` table from the LeanXcale data store:

```
Q2S: SELECT count(*) FROM Experts

Q2SL: SELECT count(*)
      FROM Experts E, CUSTOMER C
      WHERE E.expert = C.C_NAME
```

Fig. 10 shows the performance measurements of Q_2 queries that stress on the evaluation of the parallel processing of MFR/Scala queries against Spark, with and without join with a LeanXcale table. In general, the execution of these queries is much slower, as, at the Spark level, it involves shuffles of significant amounts of intermediate data. The results show good scalability and speedup with increase of the parallelism level, like for the Q_1 queries.

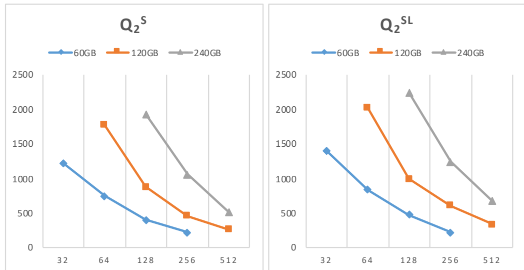

Fig. 10. Execution times (in seconds) of Q_2 queries on complex MFR/Scala queries to Spark with scales of data from 60 to 240 GB and levels of parallelism from 32 to 512.

6.3 Large Scale and Bind Joins

The third group of test cases evaluates the parallel polyglot query processing in the context of much larger data. Q_3 performs a join between a 600GB version of the MongoDB collection `Orders_Items` (containing ~770 million documents and ~3 billion order items) and a generated table `CLICKS` of size 1TB, containing ~6 billion click log records. To make the `CLICKS` dataset accessible by both Spark and LeanXcale, it is generated as an HDFS file.

```
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( ... 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, where 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 also the selectivity of the bind join. We ran experiments varying the selectivity factor `SF` between 0.02%, 0.2%, and 2%.

Comparison with Spark SQL. 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"}}, ...])
```

Fig. 11 shows the times for processing Q_3 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. Predicate selectivity does not affect significantly the query execution time, as full scans take place on both datasets anyway. Performance benefits are noticeable when using LeanXcale with bind join, where smaller values of the selectivity factor `SF` result in shorter lists of outer keys for the bind join condition and hence faster execution of the `BookOrders` subquery.

The last test case extends the Q_3 query by adding another join with the result of the `Experts` MFR subquery to Spark against the 240GB version of the generated posts log file.

Thus, Q_4 is defined as follows:

```
Q4: SELECT O.CUSTKEY, O.TITLE, C.URL, O.ORDERDATE, E.kw, E.expert
    FROM CLICKS C
      JOIN BookOrders O ON C.UID = O.CUSTKEY
      JOIN Experts E ON E.kw IN O.keywords
    WHERE C.CLICKDATE = O.ORDERDATE
      AND C.IPADDR BETWEEN a AND b
```

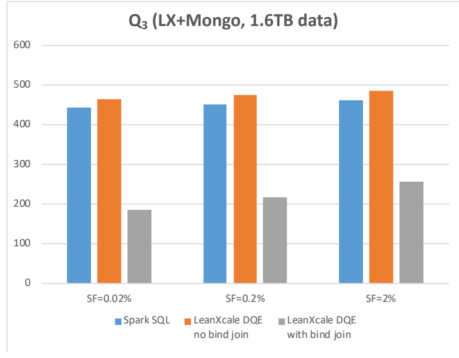

Fig. 11. Execution times (in seconds) of Q_3 queries joining an expressive JavaScript MongoDB subquery on a 600GB document collection with a 1TB click logs dataset. The level of parallelism was set to 512, i.e. 512 MongoDB shards, 512 LeanXcale DQE instances, and 512 Spark executors. To assess bind join, SF varied between 0.02%, 0.2%, and 2%.

Using bind join, the query executes as follows: first, the join between `CLICKS` at HDFS and `BookOrders` at MongoDB takes place, as in Q_3; then, after flattening `O.keywords` and identifying the list of distinct keywords, another bind join condition is pushed to the `Experts` subquery to Spark, as described in Section 5.1, to reduce the

amount of data processed by Spark transformations. To use the same mechanism for controlling the selectivity of the second join, the keywords for each book item in the `Orders_Items` MongoDB collection are generated in a way that a selectivity factor `SF` on the first join results in about the same `SF` on the second join.

Fig. 12 shows the times for processing Q_4 queries, involving 2 joins: with Spark SQL, with LeanXcale without using bind join, and with LeanXcale using bind join. Similarly to the previous test case, the performance evaluation shows that the ability for applying bind join that cannot be handled with Spark SQL gives our approach a significant advantage for selective queries. This is very useful in a wide range of industrial scenarios.

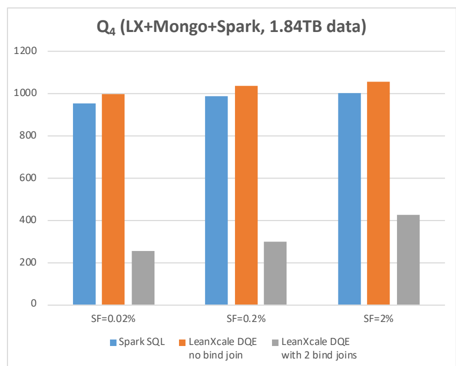

Fig. 12. Execution times (in seconds) of Q_4 queries joining the result of Q_3 queries (1TB LeanXcale table joining 600GB MongoDB collection) with an MFR/Scala/Spark subquery against 240GB HDFS file. The level of parallelism was set to 512, i.e. 512 MongoDB shards, 512 LeanXcale DQE instances, and 512 Spark executors. To assess bind join, SF varied between 0.02%, 0.2%, and 2%; this is applied at both joins.

7 Conclusion

In this paper, we introduced a parallel 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. The concept relies on an API that allows its generalization to multiple script engines and data stores. In our work, we focused on parallel joins across a partitioned relational table, the result of a parallel JavaScript subquery to MongoDB, and the result of a Spark/Scala script against an HDFS file.

Our experimental validation demonstrates the scalability of the query engine by measuring the performance of various join queries. In particular, even in the context of

sophisticated subqueries expressed as JavaScript or Scala code, 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 BigDataStack (grant 779747), project INFINITECH (grant 856632), project PolicyCLOUD (grant 870675), by the Madrid Regional Council, FSE and FEDER, project EDGEDATA (P2018/TCS-4499), CLOUDDB project TIN2016-80350-P (MINECO/FEDER, UE), and industrial doctorate grant for Pavlos Kranas (IND2017/TIC-7829).

Prof. Jose Pereira, Ricardo Vilaça, and Rui Gonçalves contributed to this work when they were 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, pp. 922-933 (2009)
2. D. Agrawal, S. Chawla, B. Contreras-Rojas, A. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, S. Thirumuruganathan, A. Troudi. RHEEM: enabling cross-platform data processing: may the big data be with you! *Proc. VLDB Endow.* 11, 11, pp. 1414–1427 (2018)
3. R. Alotaibi, D. Bursztyn, A. Deutsch, I. Manolescu, "Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue", in *ACM SIGMOD*, pp. 1660-1677 (2019)
4. Apache Drill – Schema-free SQL Query Engine for Hadoop, NoSQL and Cloud Storage, https://drill.apache.org/
5. Apache Impala, http://impala.apache.org/
6. 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", in *ACM SIGMOD*, pp. 1383-1394 (2015)
7. K. Awada, M. Eltabakh, C. Tang, M. Al-Kateb, S. Nair, G. Au, "Cost Estimation Across Heterogeneous SQL-Based Big Data Infrastructures in Teradata IntelliSphere", in *EDBT*, pp. 534-545 (2020)
8. E. Begoli, J. Camacho-Rodriguez, J. Hyde, M. Mior, D. Lemire, "Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources", in *ACM SIGMOD*, pp. 221-230 (2018)
9. C. Bondiombouy, B. Kolev, O. Levchenko, P. Valduriez, "Multistore big data integration with CloudMdsQL", *Transactions on Large-Scale Data and Knowledge-Centered Systems (TLDKS)*, pp. 48-74. Springer (2016)

10. C. Bondiombouy, P. Valduriez. Query Processing in Multistore Systems: an overview. *Int. Journal of Cloud Computing*, 5(4), pp. 309-346 (2016)
11. 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)
12. 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)
13. F. Darema, "The SPMD model: Past, present and future", in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 2131, Springer (2001)
14. S. Dasgupta, K. Coakley, A. Gupta, "Analytics-driven data ingestion and derivation in the AWESOME polystore", in *IEEE International Conference on Big Data*, pp. 2555-2564 (2016)
15. D. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, J. Gramling, "Split query processing in Polybase", in *ACM SIGMOD*, pp. 1255-1266 (2013)
16. 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)
17. 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)*, pp. 1-6 (2016)
18. I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, S. Hand. 2015. Musketeer: all for one, one for all in data processing systems. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15).* Article 2, pp. 1–16. ACM (2015)
19. L. Haas, D. Kossmann, E. Wimmers, J. Yang. Optimizing Queries across Diverse Data Sources. *Int. Conf. on Very Large Databases (VLDB)*, pp. 276-285 (1997)
20. H. Hacigümüs, J. Sankaranarayanan, J. Tatemura, J. LeFevre, N. Polyzotis, "Odyssey: a multi-store system for evolutionary analytics", *PVLDB*, vol. 6, pp. 1180-1181 (2013)
21. R. Jiménez-Peris, M. Patiño-Martinez, "System and method for highly scalable decentralized and low contention transactional processing", Filed at USPTO: 2011. European Patent #EP2780832, US Patent #US9,760,597 (2011)
22. Y. Khan, A. Zimmermann, A. Jha, D. Rebholz-Schuhmann, R. Sahay, "Querying web polystores", in *IEEE International Conference on Big Data* (2017)
23. B. Kolev, C. Bondiombouy, P. Valduriez, R. Jimenez-Peris, R. Pau, J. Pereira, "The CloudMdsQL multistore system", in *ACM SIGMOD*, pp. 2113-2116 (2016)
24. B. Kolev, O. Levchenko, E. Paciti, P. Valduriez, R. Vilaca, R. Goncalves, R. Jimenez-Peris, P. Kranas, "Parallel Polyglot Query Processing on Heterogeneous Cloud Data Stores with LeanXcale", in *IEEE International Conference on Big Data*, pp. 1756-1765 (2018)
25. B. Kolev, R. Pau, O. Levchenko, P. Valduriez, R. Jimenez-Peris, J. Pereira, "Benchmarking polystores: the CloudMdsQL experience", in *IEEE International Conference on Big Data*, pp. 2574-2579 (2016)
26. 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)
27. S. Kruse, Z. Kaoudi, B. Contreras-Rojas, S. Chawla, F. Naumann, J.-A. Quiané-Ruiz. RHEEMix in the data jungle: a cost-based optimizer for cross-platform systems. *The VLDB Journal* (2020). https://doi.org/10.1007/s00778-020-00612-x
28. J. LeFevre, J. Sankaranarayanan, H. Hacıgümüs, J. Tatemura, N. Polyzotis, M. Carey, "MISO: souping up big data query processing with a multistore system", in *ACM SIGMOD*, pp. 1591-1602 (2014)

29. Z. Minpeng, R. Tore, "Querying combined cloud-based and relational databases", in *Int. Conf. on Cloud and Service Computing (CSC)*, pp. 330-335 (2011)
30. 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)
31. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*, 4th ed. Springer, 700 pages (2020)
32. Presto – Distributed Query Engine for Big Data, https://prestodb.io/
33. A. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal, "Optimizing analytic data flows for multiple execution engines", in *ACM SIGMOD*, pp. 829-840 (2012)
34. M. Stonebraker, U. Cetintemel, "One size fits all: an idea whose time has come and gone", in *ICDE*, pp. 2-11 (2015)
35. 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)
36. A. Tomasic, L. Raschid, P. Valduriez, "Scaling access to heterogeneous data sources with DISCO", *IEEE Trans. On Knowledge and Data Engineering*, vol. 10, pp. 808-823 (1998)
37. TPC-H. http://www.tpc.org/tpch/
38. 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)
39. T. Yuanyuan, T. Zou, F. Özcan, R. Gonscalves, H. Pirahesh, "Joins for hybrid warehouses: exploiting massive parallelism in hadoop and enterprise data warehouses", in *EDBT/ICDT Conf.*, pp. 373-384 (2015)
40. J. Zhou, N. Bruno, M. Wu, P. Larson, R. Chaiken, D. Shakib. SCOPE: Parallel Databases Meet MapReduce. *PVLDB*, vol. 21, 611-636 (2012)