Benchmarking SQL on MapReduce systems using large astronomy databases

Amin Mesmoudi, Mohand-Saïd Hacid, Farouk Toumani

Noname manuscript No.
(will be inserted by the editor)

Benchmarking SQL On MapReduce systems using large astronomy databases

Amin Mesmoudi · Mohand-Saïd Hacid · Farouk Toumani

Received: date / Accepted: date

Abstract In the era of bigdata, with a massive set of digital information of unprecedented volumes being collected and/or produced in several application domains, it becomes more and more difficult to manage and query large data repositories. In the framework of the PetaSky project (http://com.isima.fr/Petasky), we focus on the problem of managing scientific data in the field of cosmology. The data we consider are those of the LSST project (http://www.lsst.org/). The overall size of the database that will be produced is expected to exceed 60 PB [28]. In order to evaluate the performances of existing *SQL On MapReduce* data management systems, we conducted extensive experiments by using data and queries from the area of cosmology. The goal of this work is to report on the ability of such systems to support large scale declarative queries. We mainly investigated the impact of data partitioning, indexing and compression on query execution performances.

1 Introduction and Context

In the age of bigdata, a massive set of digital information of unprecedented volumes are being collected and/or produced in several application domains, making it increasingly difficult to manage and query large data repositories. While traditional database management systems (DBMS) have gain their reputation thanks to their support of SQL, a declarative, simple and efficient language, they turn out to be less effective when it comes to manage very large data repositories [32]. As a consequence, there has been recently an increasing trend towards using ad hoc

This work is partially supported by Centre National de la Recherche Scientifique-CNRS, under the project Petasky-Mastodons (http://com.isima.fr/Petasky)

A. Mesmoudi and M-S Hacid
Université de Lyon, CNRS
Université Lyon 1, LIRIS, UMR5205, F-69622, France
E-mail: {amin.mesmoudi,mshacid}@liris.cnrs.fr

F. Toumani
Blaise Pascal University, CNRS
LIMOS - UMR CNRS 6158 E-mail: ftoumani@isima.fr

tools instead of DBMS in several data management applications. Being aware of this fact, database research and development community and DBMS vendors [14] have been conducting intensive efforts towards the development of advanced tools intended to be embodied in the advanced generations of DBMS products in order to support big data management. Several application domains, like social networks [31], astronomy [28] and Web [13] express a clear need for new approaches for data management and analysis. For instance, Google provided the basis of MapReduce [13], a new computing paradigm designed to enable efficient processing of huge amounts of data on a multitude of machines in a cluster. MapReduce provides a simple programming framework that enables harnessing the power of very large data centers, while hiding low level programming details related to parallelization, fault tolerance, and load balancing. When using such a framework, users have only to specify two functions, namely Map and Reduce, and the system (*e.g.*, Hadoop-http://hadoop.apache.org/) schedules[1] a parallel execution of these functions over the available nodes of a given cluster. The MapReduce approach has been proven particularly efficient to implement one pass algorithms (*i.e.*, algorithms that require only a single scan of data) [19].

As part of the Petasky project (http://com.isima.fr/Petasky), the work presented in this paper addresses several challenges related to performance evaluation of MapReduce based systems in the context of a specific application related to analysis and management of scientific data in the field of cosmology. Petasky uses as application context, *i.e.*, data, queries and system requirements, LSST (Large Synoptic Survey Telescope- http://lsst.in2p3.fr/projet-LSST.html), a project which targets the construction of a telescope of a new generation with a light-gathering power among the largest in the world. Data management and analysis is recognized as one of the most challenging aspects of the LSST [28], as more than 30 TB (TeraBytes) of complex data (3.2 Gigapixel images, uncertain data, multi-scale data) must be processed and stored each night to produce the largest non-proprietary data set in the world. The LSST data management group (`http://www.slac.stanford.edu/`) recommends the use of an open Source system based on a shared nothing architecture. Two main factors motivated such a recommendation: *(i)* integration of personalized optimization and adhoc features is facilitated, and *(ii)* support of declarative queries. Taking into account these recommendations, our main objective was to evaluate the capabilities of some systems to manage LSST data, the expected data volume at the end of the acquisition process is 60 PB (PetaBytes), using as infrastructure a cluster of hundreds of commodity servers. We focused our study on MapReduce systems for the following reasons:

- the programming model underlying such approaches enables easy development of scalable parallel applications, even for programmers without experience with distributed systems (this because the system hides the details of parallelization, fault-tolerance, locality optimization, and load balancing),
- a large variety of problems can be expressed as MapReduce computations,
- a MapReduce framework is able to scale to large clusters with thousands of nodes.

Previous works (*e.g.* [25]) on benchmarking have stressed the technological superiority of parallel DBMS compared to Hadoop, which is mainly due to a lack of

[1] By distributing and coordinating the execution of jobs.

sophisticated optimization mechanisms (*e.g.*, compression, buffering, indexing) in Hadoop. Recently, new systems (*e.g.*, HadoopDB [9,12], Hive [33]) have been proposed to strenghten MapReduce technologies with a declarative query language, in order to facilitate the expression of repeated complex tasks, as well as with sophisticated optimisation techniques. These systems are based on the idea of generating, from an SQL(-Like) query, a set of MapReduce jobs whose execution and resources management will be supported by a MapReduce framework. In this paper, we specifically report on the capability of two such systems, namely Hive and HadoopDB, to accommodate LSST data management requirements.

To achieve our goal, we designed and set up a benchmark to evaluate the performance of SQL On MapReduce data management systems. We used data sets and queries from the area of cosmology, provided by the LSST project. We extensively investigated two systems as use cases: Hive, an HDFS based system and HadoopDB, a Hadoop/PostgreSQL based system. In contrast to existing benchmarks (*e.g.*, TeraSort [4], Pavlo *et al.* [25], Intel HiBench [17]), our benchmark considers real structured data and real queries which makes evaluation of the systems relevant. We also defined additional queries (*e.g.*, distance join,...) which can be very useful to evaluate other kinds of complex computations over structured data. In this context, we conducted extensive experiments to evaluate the two SQL On MapReduce systems to manage (*e.g.*, storage, loading) LSST data on one hand and their capabilities (*i.e.*, indexing, compression, buffering, and partitioning) to optimize some categories of queries on the other hand. In contrast to existing experiments we do not consider only execution time but also loading time. We investigated different configurations obtained by varying the values of the parameters we consider:

- Infrastructure: we used different sizes for the clusters, one with **25** nodes and another with **50** nodes.
- Data: we use different datasets with sizes of **250 GB**, **500 GB**, **1 TB** and **2 TB**.
- Partitioning: we used two partitioning schemas and we compared execution time for both schemas.
- Indexing: we run our queries on both indexed and non-indexed data. For Hive, we used different attributes as indexes.
- Selectivity: we considered queries with different selectivities (*i.e.*, the number of records that satisfy the query).

Additional information and technical details about this benchmark are available at: http://com.isima.fr/Petasky/hive-vs-hadoopdb.

The rest of this paper is organized as follows: In Section 2, we briefly review basic notions related to MapReduce based systems and we highlight the need for a new benchmark for SQL on MapReduce solutions. In Section 3, we present the main features of the datasets used in our experiments. In Section 4, we describe the experiments and we discuss the outcomes in Section 5. We conclude in Section 6.

2 Background and Related Work

In this section, we review basic notions underlying a MapReduce framework and its extensions to support SQL queries and we discuss existing MapReduce benchmarks.

We start by introducing a running example that will be used through the paper to illustrate the investigated issues.

2.1 A running example

We use an example from the field of astronomical observations [28]. We consider a simple relational database schema which includes two tables. The first table, called Source, is used to store detected astronomical objects extracted from acquired images. This table includes attributes that represent the *ID* of the observation (*SourceID*), the *ID* of the detected object (*ObjectID*), the world coordinates: Right Ascension (*RA*) and Declination (*Decl*) and science exposure *ID* (*ExposureID*). As query, we use a slightly adapted query[2] from the LSST workload. This query, given below, returns appearance frequency of detected astronomical objects in the database:

```
SELECT ObjectID,
count(SourceID) as freq                    (q)
FROM Source
GROUP BY objectID
```

Table *1a* shows some sample data of the relation Source while table *1b* shows an extract of the expected results when the previous query is executed over table *1a*.

SourceID	ObjectID	RA	DECL	ExposureID
1	1	10	-3	1
5	2	15	-4	1
9	2	20	-5	5
13	4	25	-7	1
2	1	40	-7	2
6	2	45	-8	2
10	3	50	-9	2
14	4	55	-3	3
...	...	...	...	...

(a) The Source table

ObjectID	freq
1	2
2	3
3	2
4	2
...	...

(b) Results of the query

Table 1: Examples of content of the table Source and the result returned by the query (q)

2.2 MapReduce

MapReduce is a new programming model that is intended to be used to facilitate the development of scalable parallel computations on large server clusters [13]. A MapReduce framework enables to perform computations through two simple programming primitives: Map and Reduce functions. A Map function takes as input a data set in form of a set of key/value pairs and, for every pair $< k, v >$ of the input

[2] The query number 019 available at https://dev.lsstcorp.org/trac/wiki/db/queries/019.

returns zero or more intermediate key-value pairs $< k'; v' >$. The map outputs are then processed by a reduce function which implements an application specific business logic. A reduce function takes as input a key-list pair $< k'; list(v') >$, where k' is an intermediate key and $list(v')$ is the list of all the intermediate values associated with the key k', and returns as final result zero or more key-value pairs $< k'', v'' >$. Several instantiations of the map and reduce functions can operate simultaneously on different nodes of the cluster. A Map's intermediate results are stored in the local file system and sorted by the keys. After all the Map tasks are completed, the MapReduce engine notifies the Reduce tasks (which are also processes that are referred to as Reducers) to start their execution. The Reducers pull the output files from the Map tasks in parallel and merge-sort the files obtained from the Map tasks to combine the key/value pairs into a set of new key/value pairs $< k'; list(v') >$.

We shall illustrate now a MapReduce process using our running example. Figure 1 shows the sequence of data processing steps. First, we assume that data is partitioned over nodes of the cluster. An approach to use MapReduce framework to process our sample query is to define Map and Reduce functions as follows:

- Map: for each record assigned to a Mapper, one generates a (key, value) where key is the value of the attribute *ObjectID* and the value is 1.
- Reduce: for each received input $< Key, list < Value >>$, one applies the addition on $list < value >$, *i.e.*, $list < freq >$.

The system starts by reading each record in the database and generates a set of $< ObjectID, 1 >$ by applying the Map function. Combiners perform a grouping on same *ObjectIDs* on $< ObjectID, freq >$ leading to $< ObjectID, list < freq >>$. Partitioners are then used to choose a Reducer recipient. By default, a hash function is used. In the Shuffle and Sort phases, a transfer of local key, $list < value >$ to Reducers is done. Finally, each Reducer applies the Reduce function on received pairs $< key, List < value >>$.

There are several implementations of the MapReduce framework. The most popular one is Hadoop. Indeed, Hadoop uses HDFS [29] to store and partition data on nodes of a cluster. Data is partitioned in chunks (by default each chunk has a size of 64 MB). Each Map task is assigned to a chunk. Hadoop also offers default functions that can be used as combiners and partitionners. These functions could easily be replaced by other functions defined by the user. Other implementations of MapReduce, *e.g.*, Themis and SailFish, exist. Themis [27] is a MapReduce implementation designed for small clusters where the probability of node failure is much lower. It achieves high efficiency by eliminating task-level fault tolerance. Compared to Hadoop, Sailfish [26] changes the transport layer between Mappers and Reducers. The authors in [26] tried to improve the disk subsystem performance for large scale MapReduce computations by aggregating intermediate data.

A new version of Hadoop, called YARN [35], enhances the power of a Hadoop framework by improving cluster utilization and providing support for workloads other than MapReduce. Tez [8] is another in-progress project which allows to speed up data processing across both small-scale, low-latency and large-scale, high-throughput workloads. This is achieved by eliminating unnecessary tasks, synchronization barriers, and reads from and write to HDFS. Other implementations of MapReduce on specialized hardware are proposed. For example, Mars [15] is a MapReduce framework on graphics processors.

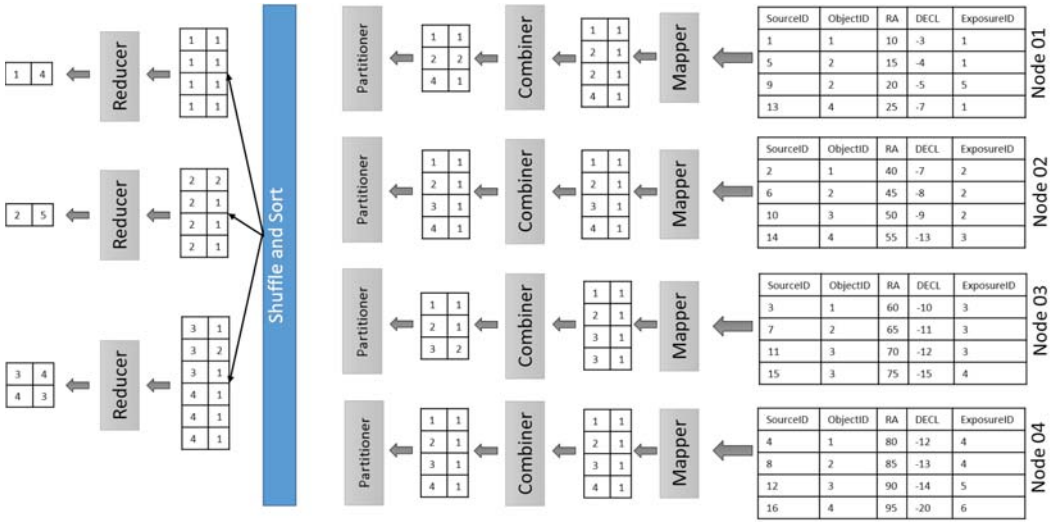

Fig. 1: Illustration of sample query processing using MapReduce

2.3 Relational operations on top of MapReduce

A lot of recent research work has been devoted to the implementation and optimization of relational operations using the MapReduce computation model. For example, *Select-Project* queries have been initially implemented using a single Map function. However, a drawback of such a solution comes from a need to achieve a full scan of all the data for each query, and this is a very costly task. To overcome such a limitation, main optimization techniques consist in the use of indexing and data compression. Instead of transferring all key/values pairs of chunks to Mappers, a filtering phase is applied to identify only relevant data to be transferred. With respect to data compression, a major proposal lies in the notion of RCFile [16]. In this case, less data is transferred from disk to RAM. Besides *Select-Project*, implementations of other relational operators have been proposed in the literature. Implementation of $GROUPBY$ (see figure 1) queries is discussed in section 2.2 while *Sorting* ($ORDERBY$) will be discussed in section 4.6.4.

The join operation is very expensive if one considers its implementation using the MapReduce model [10]. Indeed, in the case of non constrained inputs, many records that do not contribute to the final results will be transferred to Reducers. Some sophisticated implementations have been proposed to overcome the limitations of the naive approach. For example, Afrati et *al.* propose to control the number of Reducers to take advantage of the available resources for simple equi-join [10] and Multi-way equi-join [11].

Another relevant (non classical) operator for the LSST queries is similarity join. Indeed, this operation allows to group together records that have a distance (computed from some attributes) less or equal than a given threshold. Several approaches (*e.g.*,[30, 24, 23, 20, 18]) have been proposed to compute some kinds of similarity joins using the MapReduce model. For example, if one wants to group

together observations with a distance less than 15 degrees, one can specify the following query:

```
SELECT S1.objectID, S2.objectID
FROM Source as S1,Source as S2
WHERE
AngDist(S1.DECL,S2.DECL,S1.RA,S2.RA) <15
```

Where *AngDist* is the angular distance[3] between two observations (sources), *i.e.*:

```
AngDist(S1.DECL,S2.DECL,S1.RA,S2.RA) =
sin(S1.DECL)sin(S2.DECL)
+ cos(S1.DECL)cos(S2.DECL)cos(S1.RA-S2.RA)
```

The attributes *S.Ra* and *S.DECL* record the right ascension and declination, in degrees, related to a given observation *S*.

2.4 SQL on MapReduce

Several frameworks have been proposed to extend MapReduce models with SQL capabilities. We distinguish between two types of systems. In the first type, the implementation of the MapReduce framework does not change. For instance, Hive proposes to transform SQL queries into a set of MapReduce Jobs. HadoopDB follows the same principle but it uses an existing RDBMS to store the data in each worker, instead of the HDFS which is used by Hive. This allows to exploit indexing, buffering and compression capabilities offered by DBMSs.

The second type of systems requires a substantial change in the MapReduce framework. Essentially, this type of tools is based on the idea of loading all the data into the memory. For example, Shark [36] is based on the Spark[5] framework. Impala [2] is a Hive-compatible SQL engine with its own MPP (Massively Parallel Processing) execution engine. Stinger [6] is another in-progress project based on Tez. Presto [3] is an open source distributed SQL developed by Facebook for interactive queries. Other tools are proposed such as BigSQL [1] which is based on Hadoop on one hand and PostgresQL on the other hand.

2.5 MapReduce benchmark

Several benchmarks for MapReduce frameworks exist. Historically, the first one is TeraSort[4] which is devoted to test MapReduce implementations using a large scale sort application. Hadoop offers a set of tools to benchmark several implementations of MapReduce frameworks. For example, TestDFSIO[7] is a tool devoted to test I/O capabilities of DFS systems used in MapReduce frameworks. Pavlo et *al.* [25] proposed a customized benchmark tailored for search engines where a comparison between Hadoop and two parallel DBMS (Vertica and DBMS-X) was proposed. The conclusion of that work highlights the inability of existing implementations (the version of 0.19 of Hadoop has been used) of the MapReduce model to effectively process data. Three tasks have been tested: selection, aggregation and join.

[3] http://www.astronomycafe.net/qadir/q1890.html

The maximum number of attributes of the tables used in the experiments is nine (9) and the considered join query involves two tables. Authors stressed the technological superiority of parallel DBMS compared to Hadoop, which is mainly due to a lack of sophisticated optimization mechanisms (*e.g.*, compression, buffering, indexing) in Hadoop.

In [9] the same benchmark was used. Authors compared Parallel DBMS, HadoopDB and Hadoop (as an implementation of MapReduce framework). As conclusion, authors noticed that a hybrid system (*i.e.*, HadoopDB) MapReduce and DBMS could achieve performances of parallel DBMSs. Authors in [12] extended experiments in [9] by comparing Hadapt (commercial version of HadoopDB), Hive and parallel DBMSs. We believe that this comparison is not complete. Indeed, authors tried to highlight the advantages of Hadapt compared to Hive and parallel DBMS without considering all cases.

Since then, several extensions have been proposed to optimize the performance of MapReduce programs. We review in the sequel the impact of existing techniques on performances. We also explain in details when one of the two tools (Hive, HadoopDB) outperforms the other.

Intel Hibench [17] suite contains nine (9) workloads, including micro benchmarks, HDFS benchmarks, web search benchmarks, machine learning benchmarks, and data analytics benchmarks. Intel Hibench includes a dataset which is used in Pavlo et *al.* [25] for analytic tasks. In this work, the authors were interested in measuring resources consumption (*e.g.*, RAM, DISK, CPU) for each execution step within a MapReduce program. BigDataBench [37] is a benchmark suite that proposes various datasets and workloads for various applications from internet services domain. The datasets cover Wikipedia Entries, Amazon Movie Reviews, Google Web Graph, Facebook Social Network, E-commerce Transaction Data, ProfSearch Person Resumes, CALDA Data and TPC-DS Web Data. All the considered datasets, but E-commerce transaction data which uses a relational schema, include unstructured or graph data and focus on basic operations (*e.g.*, grep, wordcount, sort) or analytical processing (k-mean, pagerank, ...). We view our benchmark as complementary to existing benchmarks, including Intel Hibench and BigDataBench, for several reasons. First, our benchmark focuses on an application domain, data management in the field of cosmology, which is not covered by these benchmarks. Datasets and query workloads in this field display some specific features related to the structure of the data, data volumes, complexity of queries (*e.g.*, using specific UDFs) which are not provided by Intel Hibench and BigDataBench benchmarks. Moreover, both Intel Hibench and BigDataBench target general MapReduce related applications while, in our work, we especially target SQL on MapReduce systems. We are interested in the implementation and the optimization of SQL operators using MapReduce model. Our datasets and workload allow us to conduct deep analysis on various aspects specifically related to this kind of systems. Indeed, we allow analysis of the behaviour of these systems with respect to selectivity, partitioning, infrastructure and (data volume) scalability. To the best of our knowledge the two benchmarks (*i.e.*, Hibench, BigDataBench) do not allow to investigate in-depth query evaluation and optimization techniques with scalable structured data.

Our benchmark rests on a real case study. We consider structured data from astronomy where a set of SQL queries are already defined. Our experiments target SQL On MapReduce systems. Our objective is to report on the capabilities of

Table	size	♯records	♯attributes
Object	109 TB	38 Billions	470
Source	3.6 PB	5 Trillions	125
Moving Object	5 GB	6 Million	100
Forced Source	1.1 PB	32 Trillions	7
CCD Exposure	0.6 TB	17 Billions	45

Table 2: Expected data characteristics

those systems to manage (*e.g.*, storage, loading) LSST data on one hand and their capabilities (*i.e.*, indexing, compression, buffering, and partitioning) to optimize some categories of queries on the other hand. In contrast to existing experiments we do not consider only execution time but also loading time. We consider different configurations for the different parameters: Infrastructure, Data, Partitioning, Indexing and Selectivity. Also, our benchmark will serve as a baseline to assess query evaluation and optimization techniques that will be proposed in the context of the Petasky project.

3 Data characteristics

We focus on the issue of data management in the field of cosmology. Data are stored using relational tables where information about exposures, objects and observations can be found. The database is organized in the following four categories (represented as tables):

1. Exposure: describes each exposure, including the start date/time of exposure, the used filter, the position and orientation of the field of view on the sky, and other environmental information.
2. Object: describes attributes of each astrophysical object, including the world coordinates.
3. Source: describes each detected source on each image, including its location (x, y) on the detector, world coordinates (RA, Decl), brightness, size, and shape.
4. Moving Object: describes attributes of moving (solar system) objects, including orbital elements, brightness, albedo and rotation period.
5. Forced Source: describes measurements computed using source records.

Table 2 shows, for some tables, the estimated size of expected data at the end of the observation (which will start in 2020 for a duration of 10 years), the number of attributes as well as the expected final number of tuples. It is worth noting that: (i) the size of the data varies from Gigabytes (*e.g.*, size of the table Moving Object) to a few Petabytes (*e.g.* size of the tables Source and ForcedSource), (ii) the number of records ranges from few millions (*e.g.*, for the table Moving Object) to trillions (*e.g.*, for the Source table), and (iii) the number of attributes varies from 7, for ForcedSource table, to 450 for the Object table.

In order to evaluate the capacity of existing systems to manage LSST data, the data management group of the LSST project has provided some datasets and a tool to generate other datasets which are compliant with the characteristics of real data. Table *3a* shows the size of the four datasets PT1.1, PT 1.2, Winter 13 and SDSS

Dataset	Size
PT 1.1	90 GB
PT 1.2	220 GB
Winter 13	3 TB
SDSS	3 TB

(a) Datasets

Table	♯Attributes	♯Records	Size
Source	107	162 x 10^6	118 GB
Object	227	4 x 10^6	7 GB
RefSrcMatch	10	189 x 10^6	1.7 GB
Science_Ccd Exposure_Metadata	6	41 x 10^6	16 GB

(b) PT 1.2 characteristics

Datasets	Source: ♯Record	Source: Size (GB)	Object: ♯Record	Object: Size (GB)	♯SourceID	♯ObjectID	♯DECL	♯RA	♯ScienceCcd ExposureID
D250GB	325 x 10^6	236	9 x 10^6	14	325 x 10^6	9 x 10^6	325 x 10^6	162 x 10^6	84 x 10^3
D500GB	650 x 10^6	472	18 x 10^6	28	650 x 10^6	18 x 10^6	650 x 10^6	162 x 10^6	84 x 10^3
D1TB	1.3 x 10^9	944	36 x 10^6	56	1.3 x 10^9	36 x 10^6	1.1 x 10^9	162 x 10^6	84 x 10^3
D2TB	2.6 x 10^9	1888	72 x 10^6	112	2.6 x 10^9	72 x 10^6	2.3 x 10^9	162 x 10^6	84 x 10^3

(c) Generated dataset

Table 3: Datasets characteristics

(Sloan Digital Sky Survey, http://www.sdss.org/). Only SDSS dataset is a real dataset collected within the SDSS project. We chose to use for our experiments the dataset PT1.2 whose characteristics are depicted at Table 3b. This dataset contains 22 tables stored as "CSV" files with a total size of 220 GB. At this stage, we exploited only 4 tables: The Source table that has 107 attributes and contains 180 millions of tuples (for a total size of 139 GB), the Object table which has 229 attributes and contains 5 millions of tuples (for a total size of 7 GB), the Science_Ccd_Exposure_Metadata table which has 6 attributes and contains 41 millions of tuples (for a total size of 16 GB), and the RefSrcMatch table that has 10 attributes and contains 189 millions of tuples (for a total size of 1.5 GB). To address the scalability issue, we generated[4], from PT 1.2 dataset, four (4) new datasets with a total size of 250 GB, 500 GB, 1 TB and 2 TB respectively. New tables are generated only for the tables Source and Object. For the other tables, we kept the same records. Table 3c shows the characteristics of the generated data. The four first columns show the number of records and the size of the tables Source and Object respectively. The other columns show the number of distinct values for five attributes of the table Source involved in our workload.

4 Experiments

4.1 Processing Infrastructure

Our experiments are performed over

a distributed system composed of **50 nodes**. Each node has **8 GB of RAM**, **300 GB** of hard disk storage capacity and **2 cores**. Network rate can reach **1**

[4] generated data comply with the characteristics of real data, we used an algorithm (`https://dev.lsstcorp.org/trac/wiki/db/Qserv/Duplication`) designed by LSST data management group.

GB/s. We set up two clusters of **25** and **50** nodes, respectively. According to hdparm[5], the hard disks deliver 113 MB/sec for buffered reads.

Using this distributed architecture, we deployed HadoopDB and Hive.

4.2 Hadoop

We used two versions of Hadoop, namely 0.19.1 and 1.1.1., in our experiments. Due to the incompatibility of HadoopDB with recent versions of Hadoop, HadoopDB is coupled with Hadoop 0.19.1, whereas Hive is coupled with Hadoop 1.1.1. Both versions run on Java 1.7. We deployed the system by using the configuration mentioned in HadoopDB original paper.[6] Data in HDFS (Hadoop Distributed File System) is stored using 256 MB data blocks instead of the default 64 MB. Each MapReduce job executor runs with a maximum heap size of 1024 MB.

4.3 Hive

Hive is based on Hadoop open Source that implements a MapReduce [13] framework. Hive proposes HiveQL, a SQL-like language, to specify analysis tasks. Data is stored using HDFS. From an SQL-like query, Hive enables generating a set of MapReduce tasks and, in addition, it schedules the execution of the generated tasks. In our experiments, we used Hive 0.11.

4.4 HadoopDB

HadoopDB is based on Hive and uses HiveQL. Existing DBMS are used to store data in the nodes of the cluster. An execution plan, generated by Hive, is processed to push more complex tasks, as for example SQL queries, to nodes in the Map phase.

HadoopDB uses an XML file[7] to store access information to data. In particular, information about chunks and sub-chunks, related to each table, is stored in this file. HadoopDB processes queries as follows. First, HadoopDB parses an input query by resorting to Hive's mechanisms. Second, the catalog is parsed and the query is reformulated with respect to chunks and sub chunks information in the catalog. Then, a set of MapReduce jobs are generated together with a predefined execution order. In the Map phase of HadoopDB job, a query is sent to the DBMS and responses are formatted in a <key, value> format. After the execution of the Map and Reduce phases is completed, HadoopDB checks whether there is another job to execute. If any, partial results are stored in HDFS. If there is no additional job to be executed, responses are transferred to the node running a Hive client. In our work, HadoopDB is coupled with postgresQL in each node of the cluster.

Recently, we conducted experimental studies by using centralized systems [21, 22]: Mysql, PostgreSQL and DBMS-X (a commercialized DBMS). We found that PostgreSQL outperforms Mysql and DBMS-X with respect to LSST data. Also, in [9], the authors noticed that PostgreSQL outperforms Mysql for analytic queries.

[5] A tool that gives the average disc speeds (http://en.wikipedia.org/wiki/Hdparm).

[6] the same configuration is used in [25], the popular Hadoop benchmark paper.

[7] Called Catalog.

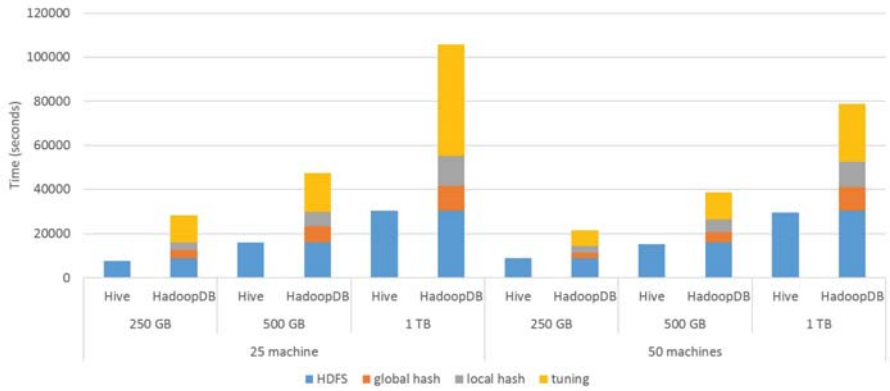

Fig. 2: Data loading: Hive vs. HadoopDB

This is why we believe that HadoopDB coupled with PostgreSQL outperforms HadoopDB coupled with Mysql. Consequently, we choose to discard Mysql and use only PostgreSQL with HadoopDB.

HadoopDB also offers a partitioning tool. This tool is used to minimize the intermediate results of jobs generated by Hive. The use of existing DBMS as data storage and access layer allows to exploit existing technologies such as indexing, buffering and compression.

4.5 Data loading and indexing

In this section, we analyze the data loading and indexing techniques of the two selected systems.

4.5.1 Hive

Data loading. The tables are loaded in the HDFS directly and stored as text files. Figure 2 shows the time needed to load data in the HDFS for the datasets used in our experiments. One can observe that the time needed for 25 nodes and 50 nodes is the same. The time needed to load datasets for a given cluster (*e.g.*, 50 machines) is linear compared to the size of the dataset. If we have 2X of data[8] we need in average 2X of time[9] to load data. Indeed, input operations (*i.e.*, reading data from local hard disks) induce the most important part of the loading time.

Data indexing. Hive supports only simple indexes with a single attribute. To deal with our query workload, we created 5 indexes: SourceID, ObjectID, RA, DECL and ScienceCCDExposureId. The time overhead due to creation of indexes is shown on figure 3, whereas the space overhead (*i.e.*, size of the indexes) is shown on figure 4. For each attribute, a set of <key, set of HDFS addresses> is stored in the HDFS. This structure is used to filter relevant records for a given query.

[8] In the rest of paper, we use the notation nX of data to denote the size n times.
[9] In the rest of paper, we use the notation nX of time to denote the time n times.

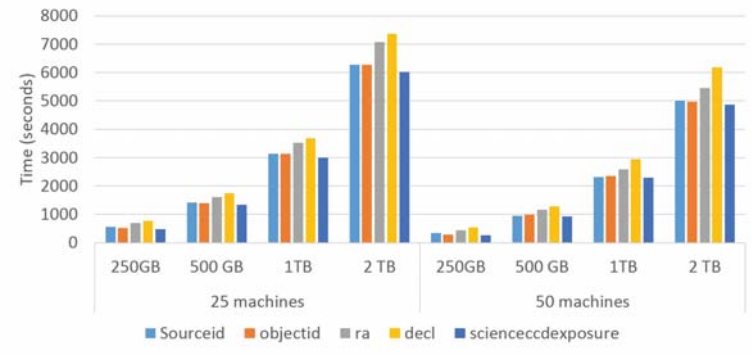

Fig. 3: Hive: Data indexing

The creation of the index is linear compared to the size of data. RA and DECL are both of type *double*, whereas SourceID, ObjectID and ScienceCCDExposure are of type *BigInt*. The Indexing of SourceID requires more time compared to ObjectID. SourceID is used as a primary key for the table Source, where the number of distinct values is bigger than the number of distinct values of the attribute ObjectID, which is used as a foreign key. As it can be observed on figure 4, SourceID needs more disk space and consequently more time to store the index. For the dataset with 2 TB, SourceID has 2,601,575,120 distinct values while ObjectID has 72,482,080 distinct values. The same rule applies to ObjectID and ScienceCCDExposure on one hand and DECL and RA on the other hand. The creation of indexes is achieved in two steps: in the first step, data is loaded in memory, and in the second step values are sorted and index information is stored on the disk. The performances of the first step depends on the size of the data. For example, we need 2X of time to index the attribute ScienceCCDExposure[10] of the dataset with 2 TB compared to the indexing of the same attribute in the dataset of 1 TB. The performances of the second step depends on the number of distinct values. For example, we need 2X of time to index the attribute DECL[11] in the dataset of 2 TB compared to the indexing of the same attribute in the dataset of 1 TB.

Regarding the speed up, we observe a gain of 15% in the indexing time if we use 2X machines.[12]

4.5.2 HadoopDB

Data loading. HadoopDB enables a user to customize the data partitioning by selecting a partitioning attribute that could be useful to speed up the processing of some queries. To achieve this task, HadoopDB proceeds in five steps: (i) data is loaded to the HDFS, (ii) a global hashing (partitioning) with respect to the number of nodes in the cluster is performed, (iii) each partition is loaded in a target node, (iv) a local hashing (partitioning) of chunks that can fit in memory is performed,

[10] the number of distinct values is the same for all the datasets.

[11] The number of distinct values double if we double the size of the dataset.

[12] In the rest of paper, we use the notation nX machines to denote the number of machines n times.

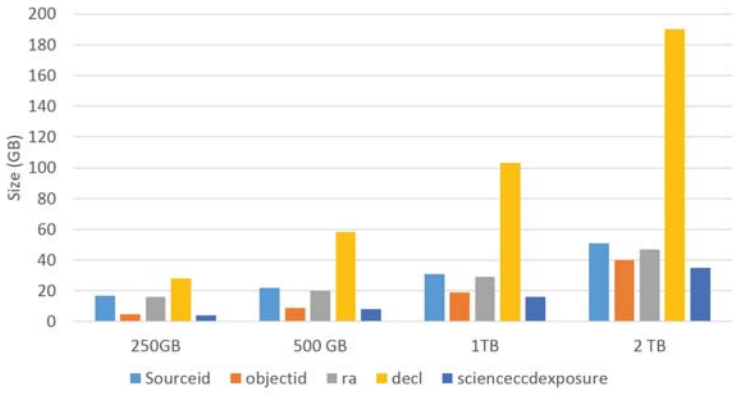

Fig. 4: Hive: Index size

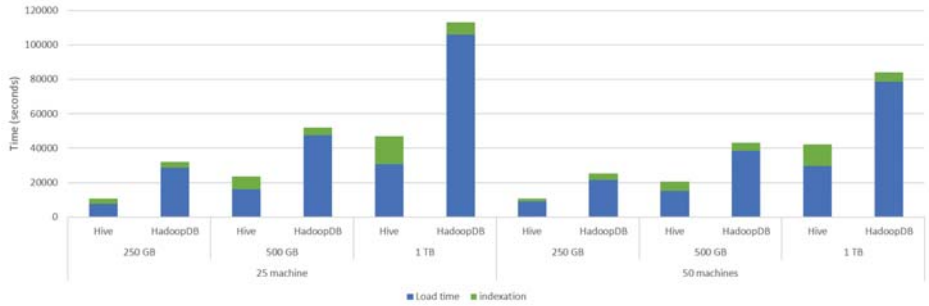

Fig. 5: Data indexing: Hive vs. HadoopDB

and (v) each chunk is loaded in a separate database. Figure 2 shows the evolution of loading time with respect to the size of datasets and the number of machines. The first step (HDFS loading) is linear with respect to the size of data due to the importance of the time needed to read local data. Global hash, local hash and DBMS tuning tasks can take benefits from parallelism in the usage of resources in the context of a shared nothing architecture.

In the same vein, when we move from a cluster of 25 nodes to a cluster of 50 nodes, time overhead (*i.e.*, loading time) drops by 25%. We also evaluated the cost of loading 2 TB of data, after the global hash and during data loading from the HDFS. As our experiments are performed over a cloud platform (virtual environment), we observed a level of failures. Indeed, reading and writing simultaneously up to 40 GB (in parallel on all nodes) leads to frequent failures in the storage platform (ceph[13]). Compared to Hive, where the time needed is equal to data loading in the HDFS, we observed that HadoopDB needs up to 300% of the time needed by Hive for 25 nodes and up to 200% for 50 nodes.

Data indexing. Indexes in HadoopDB are managed locally by the DBMS. For each chunk (stored in a separate database) a set of indexes is created. Unlike Hive, which

[13] `http://ceph.com/`

creates a global index for the data, HadoopDB leaves this task to the DBMS. An index can be used only to speed up query evaluation on local chunks. A comparison between time needed for index creation and for total data loading is shown on figure 5. The task of creating indexes takes benefit of parallelization and hence makes a better usage of the available resources. Moreover, we can take advantage of hardware resources to accelerate the creation of indexes (50 *vs* 25 nodes). HadoopDB requires only 6% of the total data loading time and 28% of the data volume.

4.6 Queries

In our experiments, we considered a query workload made of different types of queries: SELECTION, GROUP BY, ORDER BY and JOIN. We executed the queries over the three datasets, respectively, of sizes 250 GB, 500 GB and 1 TB. We used two clusters of different sizes: a cluster with 25 nodes and another cluster with 50 nodes. For each task, we evaluated[14] queries by considering two configurations: non-indexed data and indexed data.

number of records

Table 4 shows the selectivity of queries while table 5 shows the selectivity of the predicates used by the queries.

4.6.1 SELECTION Queries

Table 6 shows the set of SELECTION queries used in our experiments. The query Q_1 retrieves information about a given Source (detection of sky objects) while query Q_2 selects Sources (detection) and position (given by RA, DECL) of a given object. This query enables identifying moving objects. The query Q_3 retrieves Objects and their detection (source) for a given sky area (bounded by RA and DECL). Q_4 returns Detections (SourceID) and their positions from a given ScienceCDDExposure. The four queries target the Source Table.

Queries over non indexed data. From a SQL query, Hive and HadoopDB generate a MapReduce Job. Hive processes data SELECTION and PROJECTION within the Map task, whereas HadoopDB incorporates, in the Map task, database queries by using JDBC. A same query is sent to all Mappers. The Reduce function is not needed for the two systems because no additional processing is required after the Map task.

Figure 6 compares the execution time of the two systems (HadoopDB and Hive) using three (3) datasets with 250 GB, 500 GB and 1 TB and a configuration of 50 nodes. We first executed the queries on no indexed data.

Regarding the dataset of 250 GB, Hive outperforms HadoopDB. This is due to a latency of HadoopDB when it accesses to local DBMS using JDBC. Hive does not suffer from the same overhead since it directly accesses the data in the HDFS. However, HadoopDB outperforms Hive for datasets of 500 GB and 1 TB. Indeed,

[14] Additional results can be found in the project WebSite `http://com.isima.fr/Petasky/hive-vs-hadoopdb`

TASK	Query	D250GB	D500GB	D1TB	D2TB
SELECTION	Q_1	1	1	1	1
		$(3 * 10^{-7}\%)$	$(1.78 * 10^{-7}\%)$	$(7.69 * 10^{-8}\%)$	$(3.84 * 10^{-8}\%)$
	Q_2	43	43	43	43
		$(1.2 * 10^{-5}\%)$	$(7.67 * 10^{-6}\%)$	$(3.3 * 10^{-6}\%)$	$(1.65 * 10^{-6}\%)$
	Q_3	21	43	86	172
		$(6.6 * 10^{-6}\%)$	$(7.67 * 10^{-6}\%)$	$(6.61 * 10^{-6}\%)$	$(6.61 * 10^{-6}\%)$
	Q_4	$3.6 \text{ x } 10^3$	$7.3 \text{ x } 10^3$	$14.6 \text{ x } 10^3$	$29.2 \text{ x } 10^3$
		$(10^{-3}\%)$	$(10^{-3}\%)$	$(1.1 * 10^{-3}\%)$	$(1.1 * 10^{-3}\%)$
GROUP BY	Q_5	2	2	4	6
		$(6.15 * 10^{-7}\%)$	$(3.57 * 10^{-7}\%)$	$(3.07 * 10^{-7}\%)$	$(2.3 * 10^{-7}\%)$
	Q_6	$9 \text{ x } 10^6$	$18 \text{ x } 10^6$	$36 \text{ x } 10^6$	$58 \text{ x } 10^6$
		(2.76%)	(3.21%)	(2.76%)	(2.23%)
JOIN	Q_7	43	43	86	215
		$(1.32 * 10^{-5}\%)$	$(7.7 * 10^{-6}\%)$	$(6.61 * 10^{-6}\%)$	$(8.26 * 10^{-6}\%)$
	Q_8	$28 \text{ x } 10^3$	$28 \text{ x } 10^3$	$57 \text{ x } 10^3$	$127 \text{ x } 10^3$
		$(8 * 10^{-3}\%)$	$(5 * 10^{-3}\%)$	$(4 * 10^{-3}\%)$	$(4 * 10^{-3}\%)$
	Q_9	$7.7 \text{ x } 10^3$	$7.7 \text{ x } 10^3$	$7.7 \text{ x } 10^3$	$7.7 \text{ x } 10^3$
		$(2 * 10^{-3}\%)$	$(13 * 10^{-3}\%)$	$(6 * 10^{-4}\%)$	$(3 * 10^{-4}\%)$
ORDER BY	Q_{10}	43	43	86	86
		$(1.32 * 10^{-5}\%)$	$(7.67 * 10^{-6}\%)$	$(6.61 * 10^{-6}\%)$	$(3.3 * 10^{-6}\%)$
	Q_{11}	$28 \text{ x } 10^3$	$28 \text{ x } 10^3$	$57 \text{ x } 10^3$	$114 \text{ x } 10^3$
		$(8 * 10^{-3}\%)$	$(5 * 10^{-3}\%)$	$(4 * 10^{-4}\%)$	$(4 * 10^{-3}\%)$

Table 4: Selectivity of queries: For each query and for each data volume, the first line stands for the number of results and the second line stands for the percentage of results w.r.t the total number of records contained in the database

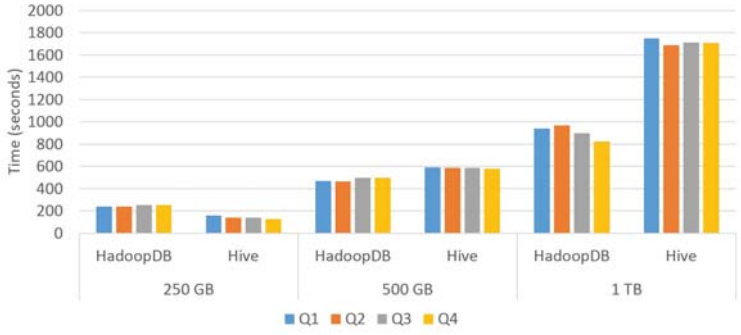

Fig. 6: SELECTION tasks without index

Predicate	D250GB	D500GB	D1TB	D2TB
SourceID =id	1	1	1	1
	$(3.07 * 10^{-7}\%)$	$(1.7 * 10^{-7}\%)$	$(7.69 * 10^{-8}\%)$	$(3.84 * 10^{-8}\%)$
ObjectID =id	43	43	43	43
	$(1.32 * 10^{-2}\%)$	$(7.67 * 10^{-6}\%)$	$(3.3 * 10^{-6}\%)$	$(1.65 * 10^{-6}\%)$
$2 < DECL < 2.05$	1.6 x 10^6	3.3 x 10^6	6.6 x 10^6	13.2 x 10^6
	(0.49%)	(0.28%)	(0.5%)	(0.5%)
$359.959 < ra < 359.96$	14 x 10^3	28 x 10^3	57 x 10^3	127 x 10^3
	$(4 * 10^{-3}\%)$	$(2.5 * 10^{-3}\%)$	$(4 * 10^{-3}\%)$	$(4 * 10^{-3}\%)$
$359.959 < ra < 359.96$ & $2 < decl < 2.05$	21	43	86	172
	$(6.46 * 10^{-6}\%)$	$(3.5 * 10^{-6}\%)$	$(6.6 * 10^{-6}\%)$	$(3.3 * 10^{-6}\%)$
ScienceCcdExposureId = id	3.6 x 10^3	7.3 x 10^3	14.6 x 10^3	29.2 x 10^3
	$(1.1 * 10^{-3}\%)$	$(6.4 * 10^{-3}\%)$	$(1.1 * 10^{-3}\%)$	$(1.1 * 10^{-3}\%)$

Table 5: Selectivity of predicates: For each predicates and for each data volume, the first line stands for the number of records satisfying the predicate and the second line stands for the those records w.r.t the total number of records contained in the database

Query	SQL syntax
Q_1	SELECT * FROM Source WHERE Sourceid=29785473054213321;
Q_2	SELECT Sourceid, ra,decl FROM Source WHERE Objectid=402386896042823;
Q_3	SELECT Sourceid, Objectid FROM Source WHERE ra > 359.959 and ra < 359.96 and decl < 2.05 and decl > 2;
Q_4	SELECT Sourceid, ra,decl FROM Source WHERE scienceccdexposureid=454490250461;

Table 6: SELECTION queries

the use of the buffer and data compression (for 1 GB raw data, only 540 MB of disk space is used to store this data) makes HadoopDB more efficient.

Regarding the scalability issue, we observe that if we double the data volume, the execution time increases in average by 250%. For example, the maximal time to query the dataset of 250 GB is 4 mins, whereas, the execution times observed for the same query over the datasets of 500 GB and 1 TB are respectively, 10 mins and 30 mins.

Queries over indexed data. Hive proceeds by filtering relevant parts of the index with respect to the condition expressed in a query. Only one index can be used to evaluate a query. The filtered part (keys, addresses) is then transferred to all the nodes of the cluster and only relevant HDFS blocks are processed. For example, query Q_3 contains two filter conditions, one on the attribute RA and the other on the attribute DECL. We evaluated query Q_3 twice. In the first evaluation, we used the index on RA. Due to its size, which exceeds 1 GB (size of heap memory)

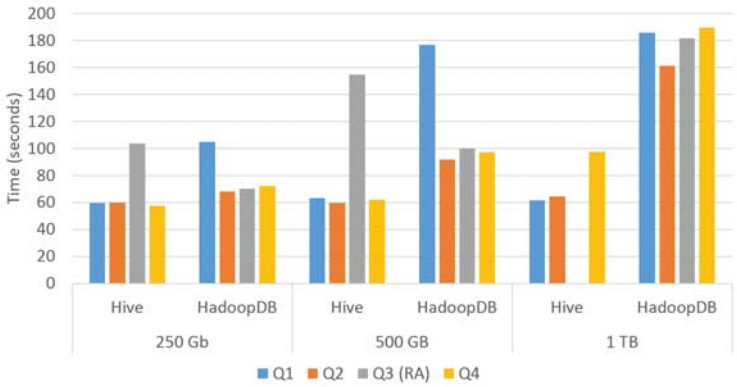

Fig. 7: SELECTION tasks with Index

for all the datasets, the workers choose to not use this index to evaluate query Q_3. In the second evaluation, we used an index on DECL. For the dataset of 1 TB, the size of the filtered part exceeds the size of the Heap memory.

In HadoopDB, indexes are automatically used by the local DBMS optimizer in each chunk.

Figure 7 shows the time needed to evaluate the queries using the indexes. One can observe that there is an improvement up to 70X for Hive, whereas, the improvement is only of 9X for HadoopDB.

Regarding data scalability, we need only 1.3X of time if we have 2X of data. For example, the maximal time to query the dataset of 250 GB is 2 mins, whereas, the datasets of 500 GB and 1 TB need 3 mins and 3.5 mins respectively.

Hive outperforms HadoopDB except for the query Q_3. In most cases, in particular for selective queries, a global index (proposed in Hive) leads to better performances than a local index (proposed in HadoopDB). However, the limit of this strategy is that when the size of the filtered part becomes larger, the nodes are no longer able to manage the indexes.

4.6.2 Aggregate Queries

Table 7 shows the two considered aggregate queries. Query Q_5 computes the number of observations (sources) of a given bounded area of the sky. Only a few tuples (< 10) are expected as a result for this query. The considered area is extended in the query Q_6, where the expected number of tuples to be returned by the query ranges from 9 to 58 millions of tuples.

Both Hive and HadoopDB use only one MapReduce Job to implement each of these queries. In the Map step, Hive performs data read, PROJECTION and filtering and as well as a partial GROUP BY. In the Reduce step, a global GROUP BY is performed. The number of Map tasks depends on the number of blocks of data in the HDFS. For example, for the 250 GB dataset, 887 Map tasks have been executed, whereas the number of Reduce tasks is determined by Hive using two parameters: the size of Map Input data and the maximum size of data handled by a Reducer (Hive.exec.reducers.bytes.per.reducer). For the dataset with 250 GB, we

Query	SQL syntax
Q_5	SELECT Objectid,count(sourceid) FROM Source WHERE ra > 359.959 and ra < 359.96 and decl < 2.05 and decl > 2 GROUP BY Objectid;
Q_6	SELECT Objectid,count(sourceid) FROM Source GROUP BY Objectid;

Table 7: Aggregation queries

have 236 Reduce tasks obtained from the size of the Source table (*i.e.*, 236 GB) and the size of data handled by a Reducer (*i.e.*, 1 GB).

For HadoopDB, within the Map step, the same query is sent to the local DBMS where PROJECTION, SELECTION and partial GROUP BY are performed. In the Reduce step, the global GROUP BY is performed. The number of Map tasks depends on the number of chunks (and subchunks) created in the data loading phase. For example, for the 250 GB dataset we have 250 Map tasks, whereas the number of Reduce tasks is set to one (1) by HadoopDB. For Hive, and because of its size, the index ObjectID cannot be used for the query Q_6. For example, for the 250 GB dataset, the size of the ObjectID index is equal to 5 GB. For Q_5, only RA can be used and only for the datasets with the sizes 250 GB and 500 GB. Indeed, the sizes of the filtered part of indexes of DECL and ObjectID exceed the heap memory dedicated to the Mapper ($>$ 1 GB).

A comparison of the execution time of queries Q_5 and Q_6 is shown on figure 8. We can observe that, for selective queries (*e.g.*, query Q_5), HadoopDB outperforms Hive for both indexed and non indexed data. Indeed, due to the few tuples ($<$ 10) transferred from Mappers to Reducers, the time needed to handle the Map tasks represents the most important part of the total time needed to process the whole query. The ability of HadoopDB, unlike Hive, to process compressed data (*e.g.*, for the dataset with 1 TB, 554 GB of compressed data for HadoopDB and 944 GB of non-compressed data for Hive) in the Map phase makes HadoopDB more efficient. The difference in performances of HadoopDB compared to Hive can reach up to 15X, with a maximum of 35 mins of processing time for Hive and 12.5 mins for HadoopDB.

Due to the size of data transferred from Mappers to Reducers (*e.g.*, 9 GB transferred in the case of the dataset of 1 TB), the time needed to handle the Reduce tasks impacts the global time needed to perform the whole query. Indeed, for non-selective queries (*e.g.*, query Q_6), Hive parallelizes the processing of GROUP BY in the Reduce phase by using all the nodes of the cluster as Reducers, whereas, for HadoopDB only one node is used as a Reducer. In the case of non-indexed data, Hive is significantly better than HadoopDB (with and without indexes). Indexes are useless in this case because all the records must be accessed (full scan). The difference in performances of Hive compared to HadoopDB can reach up to 2X, with a maximum of 36 mins for Hive and 1 hour 10 mins for HadoopDB.

Regarding data scalability, we need in average only 2X of time if we have 2X of data. For example, the maximum time to query the dataset of 250 GB is 20 mins, whereas, the datasets with 500 GB and 1 TB need 40 mins and 1 hour 10 mins respectively.

In the case of non selective queries, the use of only one Reducer represents a bottleneck for HadoopDB. To optimize this kind of queries, HadoopDB proposes

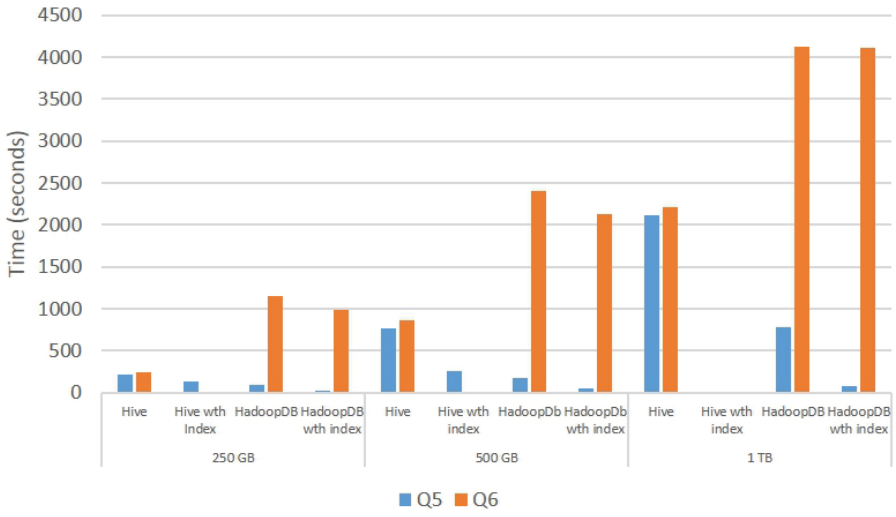

Fig. 8: GROUP BY tasks: Hive vs. HadoopDB

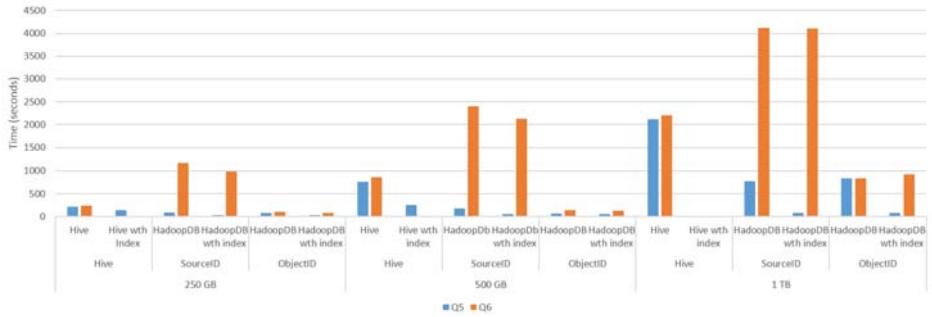

Fig. 9: GROUP BY optimization within HadoopDB vs. Hive

to modify the partitioning attribute. Indeed, our results were obtained while the Source table is partitioned using the SourceID attribute. When we change the partitioning attribute to ObjectID, we observe an improvement over the previous results, which can reach up to 5X. The results are shown in figure 9. The modification of the partitioning attribute allows: (i) to minimize the number of <key, value> transferred from Mappers to the Reducer and therefore the size of data to be processed by the Reducer, and (ii) to avoid the global GROUP BY in the Reducer side which was already done by the Mappers.

4.6.3 JOIN Queries

Table 8 shows the considered JOIN queries. Queries Q_7 and Q_5 express a JOIN between the tables Source and Object. The two queries retrieve information about objects and their observations (sources). In query Q_7, the sky area is constrained

Query	SQL syntax
Q_7	SELECT * FROM Source JOIN Object on (source.objectid=object.objectid) WHERE ra > 359.959 and ra < 359.96 and decl < 2.05 and decl > 2;
Q_8	SELECT * FROM Source JOIN Object on (source.objectid=object.objectid) WHERE ra > 359.959 and ra < 359.96;
Q_9	SELECT s.psfFlux, s.psfFluxSigma, sce.exposureType FROM Source s JOIN RefSrcMatch rsm ON (s.sourceId = rsm.sourceId) JOIN Science_Ccd_Exposure_Metadata sce ON (s.scienceCcdExposureId = sce.scienceCcdExposureId) WHERE s.ra > 359.959 and s.ra < 359.96 and s.decl < 2.05 and s.decl > 2 and s.filterId = 2 and rsm.refObjectId is not NULL;

Table 8: JOIN queries

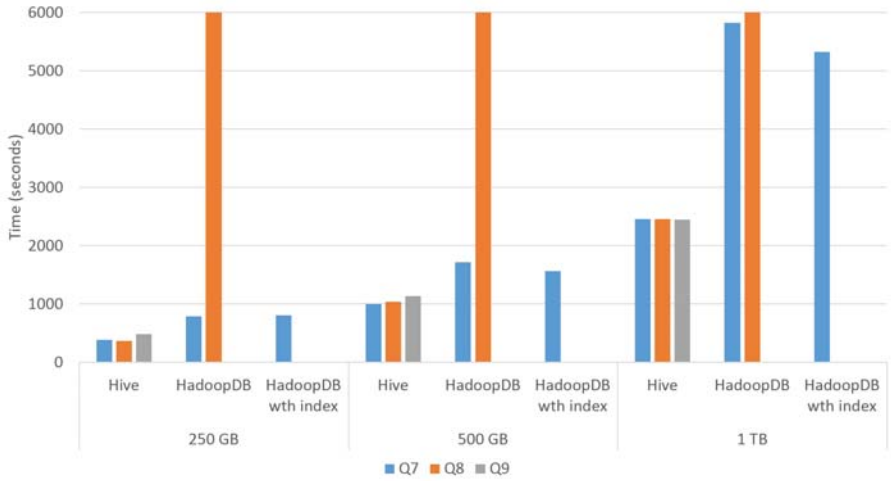

Fig. 10: JOIN tasks

by RA and DECL, whereas in query Q_8, the sky area is constrained only by RA. The sky area of query Q_7 is included in the sky area of query Q_8. Query Q_9 expresses a JOIN between three tables: Source, Science_Ccd_Exposure_Metadata and RefSrcMatch. This query retrieves the Sources, from a given sky area, and their related metadata obtained from Science_Ccd_Exposure_Metadata and RefSrcMatch.

HadoopDB does not support queries with Joins involving more than two tables, therefore the query Q_9 is not considered.

The number of MapReduce Jobs generated by Hive depends on the number of Joins in the query. For each JOIN, Hive creates a Job. Therefore, for queries Q_7 and Q_8, Hive generates only one (1) job while it generates two (2) jobs for query Q_9. In the case of two Joins or more, the result of each JOIN is used as an input for the next JOIN. Hive stores the partial results in the HDFS.

In the Map step, Hive performs data read, PROJECTION, filtering and tuples hashing (to group together tuples with same values of the JOIN key), while in the

Query	SQL syntax
♯10	SELECT Objectid,sourceid FROM Source WHERE ra > 359.959 and ra < 359.96 and decl < 2.05 and decl > 2 ORDER BY Objectid;
♯11	SELECT Objectid,sourceid FROM Source WHERE ra > 359.959 and ra < 359.96 ORDER BY Objectid;

Table 9: ORDER BY queries

Reduce step, a join is performed. The number of Map tasks depends on the number of blocks of data in the HDFS (*e.g.*, for the 250 GB dataset and queries Q_7 and Q_8, we have 938 Map tasks) whereas the number of Reduce tasks is set by Hive (*e.g.*, for the 250 GB dataset and queries Q_7 and Q_8, we have 252 Reduce tasks)

For HadoopDB, within the Map phase, the same Query is sent to the local DBMS where PROJECTION and SELECTION are performed. In the Reduce step, the JOIN is performed. The number of Map tasks depends on the number of chunks (and subchunks) created in the data loading phase (*e.g.*, for the 250 GB dataset we have 250 Map tasks) whereas the number of Reduce tasks is set to one (1) by HadoopDB.

Figure 10 shows the execution time of the considered join queries. Hive outperforms HadoopDB for both queries Q_7 and Q_8. Indeed, for the query Q_7 the difference of performance can reach 1.7X, whereas for the query Q_8, the difference can exceed 12X.

As noticed for the GROUP BY queries, when the size of the result returned by the Mappers is large, the one-Reduce strategy adopted by HadoopDB becomes problematic and represents a bottleneck.

4.6.4 ORDER BY Queries

Table 9 shows our queries for the ORDER BY task. These queries retrieve objects and their detections (sources) ordered by ObjectID. Query Q_{10}, considers a sky area bounded by RA and DECL, whereas query Q_{11} considers a larger sky area bounded only by RA. Both HadoopDB and Hive generate only one MapReduce Job from these queries. In the Map step, Hive performs data *READ, SELECTION, PROJECTION* operations and a partial ORDER BY, whereas HadoopDB sends the same query to all local DBMSs. The number of Map tasks depends on the number of data blocs in the HDFS (*e.g.*, for the dataset with 250 GB, we have 887 Map tasks). With HadoopDB, the number of tasks depends on the number of chunks created in the data loading stage. In the Reduce step, a global ORDER BY is performed by both systems using only one Reducer.

Due to their sizes, Hive can use indexes only for the datasets with 250 GB and 500 GB. For the 1 TB dataset, the size of the filtered part exceeds the size of heap memory used by the Mapper.

Figure 11 shows the evolution of execution time for HadoopDB and Hive over non indexed and indexed datasets. We observe that HadoopDB is significantly better than Hive with and without indexes.

With the index, a gain of 7X is reported for Hive and a gain of 11X is reported for HadoopDB. Furthermore, HadoopDB outperforms Hive with a factor that can exceed 2X.

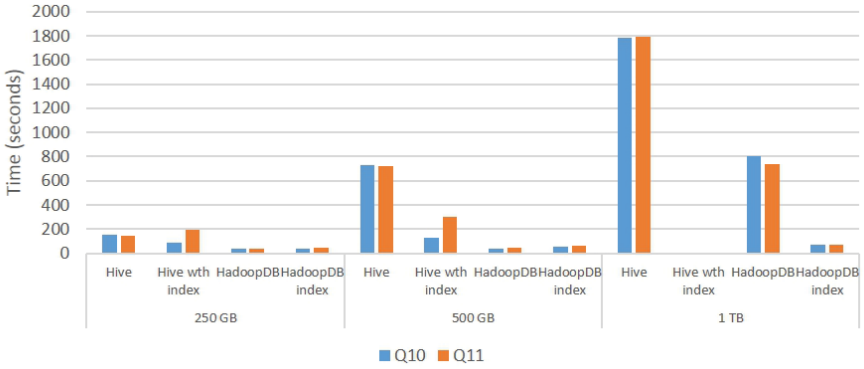

Fig. 11: ORDER BY tasks

5 Discussion

In this section, we first summarize the query evaluation mechanisms for SQL On MapReduce systems and the impact of different proposed optimization techniques on query evaluation. Then, we analyze the limits of these techniques with respect to LSST needs. We also make some proposals that could contribute to the design of a system able of meeting the LSST requirements in terms of data management.

In the LSST project, data is collected and exploited progressively. Indeed, an amount of 15 TB is expected to be collected each night. This data must be integrated with the already available one. In the sequel, we call such a data integration process: an *integration iteration*. Roughly speaking, to perform an *integration iteration* there is a need to process (index, compress, partition) the new data and integrate it with already available data.

We first start by analyzing the loading time. With respect to LSST data, both approaches proposed by Hive and HadoopDB are compatible with new data integration. However HadoopDB, is not efficient with respect to all kinds of queries, because when data is partitioned using a given attribute (*e.g.*, objectID), in the next integration iteration, there is no guarantee that all the records with the same partitioning key are in the same local database. A partitioning is efficient with respect to new data but not for all the data stored in the database (old and new data). Therefore, it is questionable whether it is really worth to have a customized data partitioning strategy when data is collected and integrated progressively. The only method that enables to guarantee the optimality of query processing, is a redistribution (*i.e.*, repartitioning) of all the available data each night. This solution is clearly not feasible due to induced time overhead: data loading time increases exponentially with respect to data volume. Indeed, more than 200-300% of the time is spent in partitioning data for a final result which is not necessarily very relevant.

Regarding indexing strategies, and with respect to data integration iteration, the strategy of HadoopDB remains very consistent. If we assume that partitioning does not trigger any particular problem, the data indexing strategy becomes even more interesting. Indeed, indexes are created at chunk level. When an index

is created, the existing indexes are not impacted and hence there is no need to recalculate them. In contrast, the strategy of creating indexes in Hive is not compatible with integration constraints. Indeed, to ensure that an index will be used, it is necessary to recalculate it. Moreover, Hive is not stable in using indexes. The indexing mechanism of Hive should be revisited with respect to these constraints and extended with additional functionalities such as: multiple index (with multiple attributes), automatic selection of indexes, integration of indexes in additional operators (currently only the selection operator supports the use of indexes).

The index size is exponential with respect to the number of distinct values. We have to study new compressed structures. Indeed, 10% of space overhead for storing one index is not a realistic solution in particular in the case of tables with hundreds of attributes and which require several indexes. The size of indexes in HadoopDB is however very acceptable (3.2% of the overall size of a raw chunk). Moreover, HadoopDB supports multiple indexes, but the usage of indexes is however still limited to selection queries. The benefit of indexing in Hive is to provide a global view about data location with however a price to pay in terms of sizes of indexes and maintenance since there is a need to recalculate an index at each integration iteration step. Indexes proposed in HadoopDB do not require a large size and are calculated quickly which is compatible with integration iteration steps. Nevertheless, HadoopDB indexes do not provide a global view of the data.

We believe that a hierarchical indexing, with several levels, may offer a good compromise. Such an index structure can take advantages from the global view of data offered by Hive and the ease of use offered by HadoopDB.

Before comparing the execution time of the considered systems, we will first introduce some cost metrics which will be used to analyze the various query evaluation and optimization strategies of Hive and HadoopDB.

First of all, as mentioned previously, an SQL query is mapped to a set of MapReduce jobs. In each MapReduce job, we have five steps: (i) Data is read and transferred to Mappers as key/value pairs. We denote by CostRead_Data, the cost of this step. (ii) A Map function is applied on the data read and key/value pairs received by the Mapper. We denote this cost by Cost_Map. (iii) A network synchronization by sending key/value pairs generated by the mappers to Reducers. We denote by Cost_network, the cost of this step. (iv) The reduce function is applied. We denote this cost by Cost_reduce. (v) Partial results are written in the HDFS, these results will be, if necessary, used as input for the next job. We denote this cost by Cost_WriteHDFS. These five steps are not required for all the queries. Indeed, for JOIN queries, we need all the steps, while for SELECTION queries only steps (i) and (ii) are required.

The overall cost of a job is obtained by summing these five costs. We denote this cost by Cost_Job. Finally, we define the cost related to query processing as the sum of all costs of all jobs associated with the query. In the rest of this paper, we discuss the impact of each cost with respect to query types.

In the case of selection queries, and without considering indexes, only one job is generated both in the case of Hive and HadoopDB. In this case, the cost of a query is the same as the cost of its associated job. In the implementation of such a job, the two systems read data and applies the filtering associated with predicates in the query. Consequently the cost of such a job is: CostRead_Data + Cost_Map.

The CostRead_Data, in HadoopDB, is given by the time necessary to read data from DBMS using JDBC. For Hive, this cost is related to chunks reading from

the HDFS. As data in the DBMS is compressed, the CostRead_Data related to HadoopDB job is less than CostRead_Data related to Hive in most cases. The only different case is when CostRead_Data, for HadoopDB, is dominated by JDBC latencies.

Both tools propose to use indexes to minimize CostRead_Data. Indeed, in HadoopDB, indexes are used within the DBMS which allow to only access relevant disk pages by the DBMS. For Hive, we use an additional data structure that allows to locate only relevant HDFS chunks. In this vein, a filtering phase is used before applying the Map function. The Cost CostRead_Data is minimized using the index. We have seen that when Hive decides to use an index, the result is better because Hive has a global view of the data which enables to locate relevant data more efficiently compared to HadoopDB.

The drawback of Hive is when the filter part has an important size. Indeed, when reading data, Hive extracts relevant chunks and sub-chunks, from index, that satisfy predicates in the query. If the filtered part cannot fit into the memory of Mappers, Hive is not able to use indexes. To conclude, we need a more sophisticated optimizer that can decide when a usage of an index is relevant and which index to use. In addition, Hive indexing mechanism must be extended to enable complex indexes with multiple attributes.

Another important point, for LSST project, is the choice of attributes to index. As some queries are known, it will be very interesting to have an index recommendation system for a large number of attributes.

With respect to GROUP BY queries, both Hive and HaddopDB read data, apply the filter related to predicates in a query and then apply partial GROUP BY, in the Map side. They send then the data to the Reducer(s) where a global GROUP BY is applied. Only one job is used, therefore the cost of a query is equivalent to the cost of the associated job. The cost of such a job is then: CostRead_Data + Cost_Map + Cost_network + Cost_reduce. For selective queries, only a few (<10) tuples are transferred from Mappers to Reducers. Therefore, CostRead_Data and Cost_Map dominate the Job cost. By compressing data HadoopDB outperforms Hive.

With respect to non-selective queries, the data transferred from Mappers to Reducers impacts the performance of the generated jobs. As HadoopDB uses one Reducer to perform the global GROUP BY and Hive uses several Reducers, we have observed that even when an index is used in HadoopDB, Hive outperforms HadoopDB for non-selective queries. Indeed, Cost_reduce is shared between several machines, whereas for HadoopDB, due to the use of a single Reducer, this cost dominates the query processing cost and constitutes a bottleneck.

HadoopDB proposes to modify the partitioning attribute to deal with GROUP BY queries. Indeed, by changing the partitioning attribute, we are sure that most records having the same GROUP BY key are in the same node and hence there is no need to perform a global GROUP BY for the concerned records. In this case, the number of key/values transferred from Mappers to Reducers is minimized. We have observed a very significant gain in this case. The cost of network transfer and reduce phases are minimized using this technique. The drawback of this strategy comes from the fact that it requires to identify *a priori* the adequate partitioning attribute.

Determining a good partitioning strategy for LSST data is very challenging knowing that in LSST, several attributes are used as grouping attributes. It will

be interesting to study the problem of partitioning data starting from a set of predefined queries. Another issue is related to the identification of the adequate number of Reducers to use for each query. In current state of affairs, Hive leaves this choice to the user. Extending such a system in order to be able to automatically determine the optimal number of Reducers to use during query execution is indeed an interesting research issue.

Regarding JOIN queries, Hive generates, for each JOIN, a MapReduce job.

HadoopDB does not support joins of more than two tables and do not allow multiple Reducers to perform the reduce function. Indeed, for JOIN queries and to the best of our knowledge, there is no optimization technique implemented in the open source release of HadoopDB. Also indexes are not used to perform joins. The cost of JOIN queries is obtained by the sum of costs of the associated jobs. Intermediate results are stored in the HDFS. In this case, the order of joining the relations has an important impact on query performances. To the best of our knowledge, no relevant techniques for finding the best order of joining tables is proposed so far. We believe that the optimizer of Hive should be revisited to incorporate sophisticated scheduling technique based on statistics. Indeed, the order of evaluation of joins will significantly impact query performance. In the TEZ system, Beta-version of Hadoop, MapReduce tasks can be pipelined without going through the store in the HDFS.

Regarding ORDER BY queries, both approaches propose to handle these queries in one job. HadoopDB outperforms Hive because HadoopDB takes advantages of compression and indexing, handled by the DBMS and minimizes the cost of CostRead_Data. The ORDER BY in the Map phase is not mandatory because it can be performed by the DBMS when it reads the data. We believe that the MapReduce model is not suitable for handling JOIN and ORDER BY queries. We believe that this model is efficient for queries that need only one pass on the data (*e.g.*, Selection and GROUP BY).

Another issue is the optimization of UDF (User Defined Functions). Indeed, UDF are very frequently used in astrophysics applications. Optimization within the MapReduce framework is not an easy task. Within Hive, we implemented some UDFs and we successfully executed those queries in a declarative manner. Hive offers several kinds of UDF: UDFs that can be evaluated in the Map phase, in the Reduce phase and as a separated job. For HadoopDB, the integration of UDFs is achieved manually as a separate MapReduce Job. HadoopDB lacks mechanisms for evaluating queries declaratively.

With respect to query frequency, in LSST we expect a half million of queries per day. At any time, we expect that the system should handle 50 simple queries (that need a few seconds to be evaluated) and 20 complex query (that could require hours or days to be evaluated). The versions of Hadoop used in our experiments are not able to support execution of multi-jobs. Actually, another version of Hadoop, called Yarn, provides this ability. Therefore, it will be interesting to investigate global execution and optimization of sets of queries in a MapReduce framework.

6 Conclusion

We designed and set up a benchmark for structured data from astronomy where a set of SQL queries are already defined. Our experiments targeted two

SQL On MapReduce systems, Hive, a HDFS based system and HadoopDB, a Hadoop/Postgres based system. Our objective is to report on the capabilities of those systems to manage (*e.g.*, storage, loading) LSST data on one hand and their capabilities (*i.e.*, indexing, compression, buffering, and partitioning) to optimize some kinds of queries on the other hand. In contrast to existing experiments we did not consider only execution time but also loading time. We considered different configurations for the different parameters related to material, data, partitioning, indexing and selectivity. We reported on the impact of data partitioning, indexing and compression on query evaluation performances. We also highlighted the need for new techniques for query optimization in emerging systems.

There are many research directions that could be pursued to improve query processing in the case of LSST data. For example, it could be interesting to experiment another distribution model. Due to the cost of network communication between Mappers and Reducers, the BSP model [34] could be an interesting alternative. We also believe that the extension of the presented analysis to other categories of systems (*e.g.*, Hybrid In-Memory/Column-oriented DBMS) would be clearly an interesting future work. Another important issue is to automatically choose the parameters of the distribution framework (*i.e.*, Hadoop), the partitioning attribute and the index to be created and used, from a workload. We are currently investigating these issues.

References

1. BigSQL. `http://www.bigsql.org/se/`. [Online; accessed 21-October-2014].
2. Impala. `http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html`. [Online; accessed 21-October-2014].
3. Presto. `http://prestodb.io/`. [Online; accessed 21-October-2014].
4. Sort Benchmark. `http://sortbenchmark.org/`. [Online; accessed 21-October-2014].
5. Spark. `http://spark.apache.org/`. [Online; accessed 21-October-2014].
6. Stinger. `http://hortonworks.com/labs/stinger/`. [Online; accessed 21-October-2014].
7. TestDFSIO. `http://wise.ajou.ac.kr/mata/hadoop-benchmark-testdfsio/`. [Online; accessed 21-October-2014].
8. TEZ. http://tez.incubator.apache.org/. [Online; accessed 21-October-2014].
9. A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.
10. F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM, 2010.
11. F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *Knowledge and Data Engineering, IEEE Transactions on*, 23(9):1282–1298, 2011.
12. K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *Proceedings of the 2011 international conference on Management of data*, pages 1165–1176. ACM, 2011.
13. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
14. I. Gartner. Big data challenges. `http://www.adeptia.com/products/Gartner-Cool-Vendors-in-Integration-2010.pdf`. [Online; accessed 21-October-2014].
15. B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.
16. Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1199–1208. IEEE, 2011.

17. S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proceedings of the ICDE Workshops*, volume 2010, 2010.
18. Y. Kim and K. Shim. Parallel top-k similarity join algorithms using mapreduce. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 510–521. IEEE, 2012.
19. R. Lämmel. Google's mapreduce programming model—revisited. *Science of computer programming*, 70(1):1–30, 2008.
20. W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1016–1027, 2012.
21. A. Mesmoudi and M. Hacid. A comparison of systems to large-scale data access. In *Database Systems for Advanced Applications - 19th International Conference, DASFAA 2014, International Workshops: BDMA, DaMEN, SIM - 3 - , UnCrowd; Bali, Indonesia, April 21-24, 2014, Revised Selected Papers*, pages 161–175, 2014.
22. A. Mesmoudi and M.-S. Hacid. A test framework for large scale declarative queries: preliminary results. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 858–859. ACM, 2014.
23. A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proceedings of the VLDB Endowment*, 5(8):704–715, 2012.
24. A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 949–960. ACM, 2011.
25. A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178. ACM, 2009.
26. S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 4. ACM, 2012.
27. A. Rasmussen, M. Conley, G. Porter, R. Kapoor, A. Vahdat, et al. Themis: an i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 13. ACM, 2012.
28. R. Shaw. Lsst data challenge handbook. *Version2. 0 (Tucson: LSST Corp.)*, 2012.
29. K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.
30. Y. N. Silva and J. M. Reed. Exploiting mapreduce-based similarity joins. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 693–696. ACM, 2012.
31. C. Smith. Reinventing Social Media. `http://www.businessinsider.com/social-medias-big-data-future-2014-3`. [Online; accessed 21-October-2014].
32. M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.
33. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.
34. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
35. V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.
36. R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 international conference on Management of data*, pages 13–24. ACM, 2013.
37. Y. Zhu, J. Zhan, C. Weng, R. Nambiar, J. Zhang, X. Chen, and L. Wang. Bigop: Generating comprehensive big data workloads as a benchmarking framework. *arXiv preprint arXiv:1401.6628*, 2014.