Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

Improving the performance of Hadoop Hive by sharing scan and computation tasks

Tansel Dokeroglu[1*], Serkan Ozal[1], Murat Ali Bayir[2], Muhammet Serkan Cinar[3] and Ahmet Cosar[1]

Abstract

MapReduce is a popular programming model for executing time-consuming analytical queries as a batch of tasks on large scale data clusters. In environments where multiple queries with similar selection predicates, common tables, and join tasks arrive simultaneously, many opportunities can arise for sharing scan and/or join computation tasks. Executing common tasks only once can remarkably reduce the total execution time of a batch of queries. In this study, we propose a Multiple Query Optimization framework, SharedHive, to improve the overall performance of Hadoop Hive, an open source SQL-based data warehouse using MapReduce. SharedHive transforms a set of correlated HiveQL queries into a new set of *insert queries* that will produce all of the required outputs within a shorter execution time. It is experimentally shown that SharedHive achieves significant reductions in total execution times of TPC-H queries.

Keywords: Hadoop; Hive; Data warehouse; Multiple-query optimization

Introduction

Hadoop is a popular open source software framework that allows the distributed processing of large scale data sets [1]. It employs the MapReduce paradigm to divide the computation tasks into parts that can be distributed to a commodity cluster and therefore, provides horizontal scalability [2-9]. The MapReduce functions of Hadoop uses (*key*,*value*) pairs as data format. The input is retrieved in chunks from Hadoop Distributed File System (HDFS) and assigned to one of the *mappers* that will process data in parallel and produce the (k_1,v_1) pairs for the reduce step. Then, (k_1,v_1) pair goes through *shuffle* phase that assigns the same k_1 pairs to the same reducer. The reducers gather the pairs with the same k_1 values into groups and perform aggregation operations (see Figure 1). HDFS is the underlying file system of Hadoop. Due to its simplicity, scalability, fault-tolerance and efficiency Hadoop has gained significant support from both industry and academia; however, there are some limitations in terms of its interfaces and performance [10]. Querying the data with Hadoop as in a traditional RDBMS infrastructure is one of the most common problems that Hadoop users face. This affects a majority of users who are not familiar with the internal details of MapReduce jobs to extract information from their data warehouses.

Hadoop Hive is an open source SQL-based distributed warehouse system which is proposed to solve the problems mentioned above by providing an SQL-like abstraction on top of Hadoop framework. Hive is an SQL-to-MapReduce translator with an SQL dialect, HiveQL, for querying data stored in a cluster [11-13]. When users want to benefit from both MapReduce and SQL, mapping SQL statements to MapReduce tasks can become a very difficult job [14]. Hive does this work by translating queries to MapReduce jobs, thereby exploiting the scalability of Hadoop while presenting a familiar SQL abstraction [15]. These attributes of Hive make it a suitable tool for data warehouse applications where large scale data is analyzed, fast response times are not required, and there is no need to update data frequently [4].

Since most data warehouse applications are implemented using SQL-based RDBMSs, Hive lowers the barrier to moving these applications to Hadoop, thus, people who already know SQL can easily use Hive. Similarly, Hive makes it easier for developers to port SQL-based applications to Hadoop. Since Hive is based on a

*Correspondence: tansel@ceng.metu.edu.tr
[1]Middle East Technical University Computer Engineering Department, Cankaya, Ankara, Turkey
Full list of author information is available at the end of the article

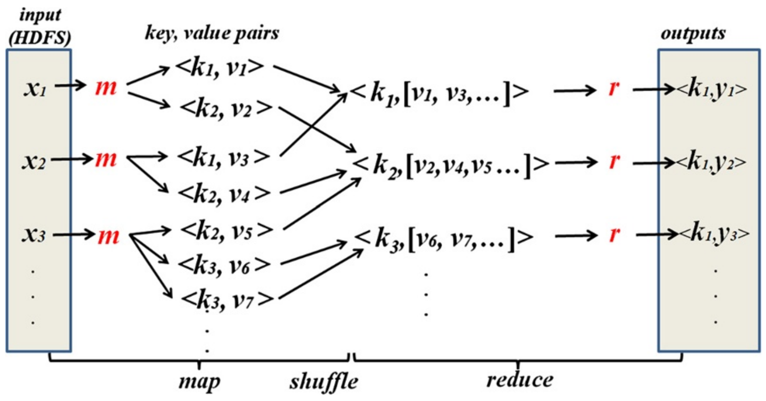
input (HDFS)
key, value pairs
outputs
x1 x2 x3
m
<k1, v1> <k2, v2> <k1, v3> <k2, v4> <k2, v5> <k3, v6> <k3, v7>
<k1,[v1, v3,…]> <k2,[v2,v4,v5…]> <k3,[v6, v7,…]>
r
<k1,y1> <k1,y2> <k1,y3>
map
shuffle
reduce

Figure 1 MapReduce tasks.

query-at-a-time model and processes each query independently, issuing multiple queries in close time interval decreases performance of Hive due to its execution model. From this perspective, it is important to note that there has been no study, to date, that incorporates the Multiple-query optimization (MQO) technique for Hive to reduce the total execution time of a batch of queries [16-18].

Studies concerning MQO for traditional warehouses have shown that it is an efficient technique that dramatically increases the performance of time-consuming decision support queries [2,19-21]. In order to improve the performance of Hadoop Hive in massively issued query environments, we propose SharedHive, which processes HiveQL queries as a batch and improves the total execution time by merging correlated queries before passing them to the Hive query optimizer [6,15,22]. By analyzing the common tasks of correlated HiveQL queries we merge them to a new set of *insert queries* with an optimization algorithm and execute as a batch. The developed model is introduced as a novel component for Hadoop Hive architecture.

In Related work Section, brief information is presented concerning the related work on MQO, SQL-to-MapReduce translators that are similar to Hive, and recent query optimization studies on MapReduce framework. SharedHive system architecture Section explains the traditional architecture of Hive and introduces our novel MQO component. The next Section (Sharing scan and computation tasks of HiveQL queries) explains the process of generating a set of merged *insert queries* from correlated queries. Experimental setup and results Section discusses the experiments conducted to evaluate the SharedHive framework for HiveQL queries that have different correlation levels. The Section before the conclusion presents the comparison of SharedHive with the other MapReduce-based MQO methods. Our concluding remarks are given in Conclusions and future work Section.

Related work

The MQO problem was introduced in the 1980s and finding an optimal global query plan using MQO was shown to be an NP-Hard problem [16,23]. Since then, a considerable amount of work has been undertaken on RDBMSs and data analysis applications [24-26]. Mehta and DeWitt considered CPU utilization, memory usage, and I/O load variables in a study during planning multiple queries to determine the degree of intra-operator parallelism in parallel databases to minimize the total execution time of declustered join methods [27]. A proxy-based infrastructure for handling data intensive applications has been proposed by Beynon [28]; however, this infrastructure was not as scalable as a collection of distributed cache servers available at multiple back-ends. A data integration system that reduces the communication costs by a multiple query reconstruction algorithm is proposed by [29]. IGNITE [30] and QPipe [31] are important studies that use the micro machine concept for query operators to reduce the total execution time of a set of queries. A novel MQO framework is proposed for the existing SPARQL query engines [32]. A cascade-style optimizer for Scope, Microsoft's system for massive data analysis, is designed in [33]. CoScan [34,35] shows how sharing scan operations can benefit multiple MapReduce queries.

In recent years, a significant amount of research and commercial activity has focused on integrating MapReduce and structured database technologies [36]. Mainly there are two approaches, either adding MapReduce features to a parallel database or adding database technologies to MapReduce. The second approach is more attractive because no widely available open source parallel database system exists, whereas MapReduce is available as an open source project. Furthermore, MapReduce is accompanied by a plethora of free tools as well as having cluster availability and support. Hive [11], Pig [37], Scope [20], and HadoopDB [10,38] are projects that provide SQL abstractions on top of MapReduce platform to familiarize the programmers with complex queries. SQL/MapReduce [39] and Greenplum [21] are recent projects that use MapReduce to process user-defined functions (UDF).

Recently, there have been interesting studies that apply MQO to MapReduce frameworks for unstructured data; for example MRShare [40] processes a batch of input queries as a single query. The optimal grouping of queries for execution is defined as an optimization problem based on MapReduce cost model. The experimental results reported for MRShare demonstrate its effectiveness. In spite of some initial MQO studies to reduce the execution time of MapReduce-based single queries [41], to our knowledge there is no study similar to ours that is related to the MQO of Hadoop Hive by using *insert query* statements.

SharedHive system architecture

In this section, we briefly present the architecture of SharedHive which is a modified version of Hadoop Hive with a new MQO component inserted on top of the *Driver* component of Hive (see Figure 2). Inputs to the *driver* which contains compiler, optimizer and executer are pre-processed by the added Multiple Query Optimizer component which analyzes incoming queries and produces a set of merged HiveQL *insert queries*. Finally, the remaining queries that don't have any correlation with others are appended at the end of the correlated query sets. The system catalog and relational database structure (relations, attributes, partitions, etc.) are stored and maintained by *Metastore*. Once a HiveQL statement is submitted, it is maintained by *Driver* which controls the execution of tasks in order to answer the query. Compiler parses the query string and transforms the parse tree to a logical plan. Optimizer performs several passes over the logical plan and rewrites it. The physical plan generator creates a physical plan from the logical plan.

HiveQL statements are submitted via the Command Line Interface (CLI), the Web User Interface or the thrift interface. Normally, the query is directed to the driver component in conventional Hive architecture. In SharedHive, the MQO component (located after the client interface) receives the incoming queries before the driver component. The set of incoming queries are inspected, their common tables and intermediate common joins are detected, and merged to obtain a new set of HiveQL queries that answer all the incoming queries. The details of this process are explained in the next Section.

The new MQO component passes the new set of merged queries to the compiler component of Hive driver that produces a logical plan using information from the Metastore and optimizes this plan using a single rule-based optimizer. The execution engine receives a directed acyclic graph (DAG) of MapReduce and associated HDFS tasks, then executes them in accordance with the dependencies of the tasks. The new MQO component does not require any major changes in the system architecture of Hadoop Hive and can be easily integrated into Hive.

Sharing scan and computation tasks of HiveQL queries

In order to benefit from the common scan/join tasks of the input queries and reduce the number (i.e. total amount) of redundant tasks, SharedHive merges *input queries* into a new set of HiveQL *insert queries* and produces answers to each query as a separate HDFS file.

The problem of merging a set of queries can be formally described as:

Input: A set of HiveQL queries $Q=\{q_1,...,q_n\}$.
Output: A set of merged HiveQL queries $Q^{'}=\{q_1^{'},...,q_m^{'}\}$, where $m \leq n$.

Rewrite/combine the given input queries in such a way that the total execution time of query set $Q^{'}$ is less than the total execution time of query set Q. If the execution time of query q_i is represented with t_i then

$$\sum_{i=1}^{m}(t_i^{'}) \leq \sum_{i=1}^{n}(t_i)$$

Given $q_i^{'}$ is the merged *insert query* corresponding to queries q_j and q_k then all of the output tuples and columns required by both queries must be produced by query $q_i^{'}$ preserving the predicate attributes of q_j and q_k.

The existing architecture of Hive produces several jobs that run in parallel to answer a query. The *insert queries* merged by SharedHive can combine the scan and/or intermediate join operations of the input queries in a new set of *insert queries* and gain performance increases by reducing the number of MapReduce tasks and the sizes of read/written HDFS files.

Unlike the traditional SQL statements, HiveQL join query statements are written in the FROM part of the query [15] such as

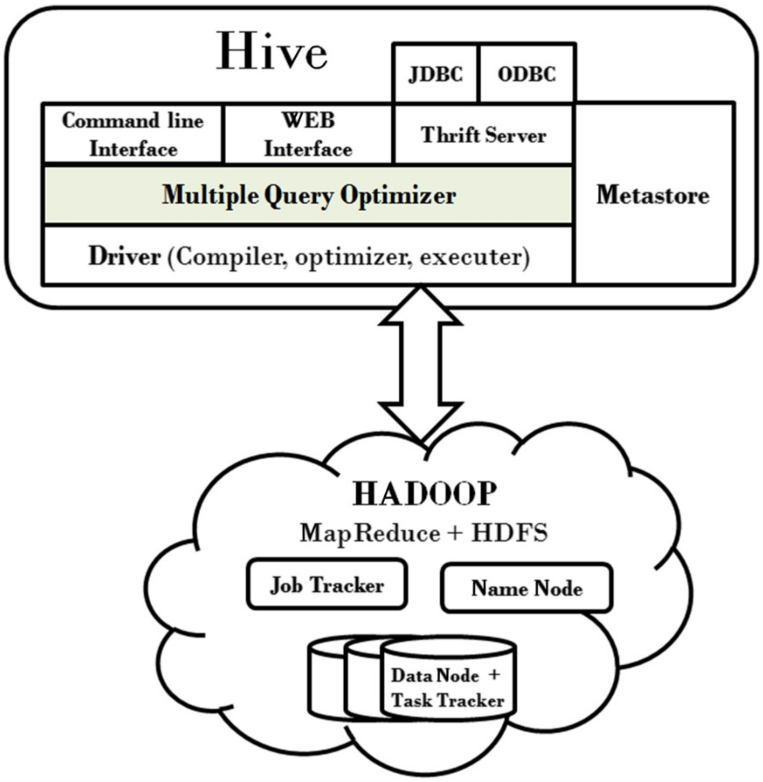

Figure 2 Architecture of SharedHive with newly added multiple query optimizer component.

SELECT SUM (L_EXTENDEDPRICE)
FROM LINEITEM L JOIN PART P ON P.P_PARTKEY = L.L_PARTKEY;

instead of
SELECT SUM (L_EXTENDEDPRICE)
FROM LINEITEM L, PART P
WHERE L.L_PARTKEY = P.P_PARTKEY;

The example below shows how a merged HiveQL *insert query* for TPC-H queries *Q1* and *Q6* is constructed.

Merging TPC-H Queries *Q1* and *Q6* :

Query *Q1*
CREATE EXTERNAL TABLE LINEITEM
(L_ORDERKEY INT ,..., L_COMMENT STRING)
CREATE TABLE q1_pricing_summary_report
(L_RETURNFLAG STRING ,..., COUNT_ORDER INT);
INSERT OVERWRITE TABLE q1_pricing_summary_report
  SELECT L_RETURNFLAG ,..., COUNT(*)
  FROM LINEITEM
  WHERE L_SHIPDATE ≤ '1998-09-02'
  GROUP BY L_RETURNFLAG, L_LINESTATUS
  ORDER BY L_RETURNFLAG, L_LINESTATUS;

Query *Q6*
CREATE EXTERNAL TABLE LINEITEM
(L_ORDERKEY INT ,..., L_COMMENT STRING)
CREATE TABLE q6_forecast_revenue_change (REVENUE DOUBLE);
INSERT OVERWRITE TABLE q6_forecast_revenue_change
  SELECT SUM(...) **AS** REVENUE
  FROM LINEITEM
  WHERE L_SHIPDATE ≥ '1994-01-01 **AND**
  L_SHIPDATE < '1995-01-01' **AND**
  L_DISCOUNT ≥ 0.05 **AND**
  L_DISCOUNT ≤ 0.07 **AND** L_QUANTITY < 24;

Merged *insert query* for (*Q1+Q6*)
CREATE EXTERNAL TABLE LINEITEM
(L_ORDERKEY INT ,..., L_COMMENT STRING)
CREATE TABLE q1_pricing_summary_report
(L_RETURNFLAG STRING ,..., COUNT_ORDER INT);
CREATE TABLE q6_forecast_revenue_change(REVENUE DOUBLE);
FROM LINEITEM
INSERT OVERWRITE TABLE q1_pricing_summary_report
  SELECT L_RETURNFLAG ,..., COUNT(*)
  WHERE L_SHIPDATE ≤ '1998-09-02'
  GROUP BY L_RETURNFLAG, L_LINESTATUS
  ORDER BY L_RETURNFLAG, L_LINESTATUS
INSERT OVERWRITE TABLE q6_forecast_revenue_change
  SELECT SUM(...) **AS** REVENUE
  WHERE L_SHIPDATE ≥ '1994-01-01
  AND L_SHIPDATE < '1995-01-01' **AND** L_DISCOUNT ≥ 0.05
  AND L_DISCOUNT ≤ 0.07 **AND** L_QUANTITY < 24;

The underlying SQL-to-Mapreduce translator of Hive uses *one operation to one job* model [22] and opens a new job for each operation (table scan, join, group by, etc.) in a SQL statement. Significant performance increases can be obtained by reducing the number of MapReduce tasks of these jobs. Figure 3 presents MapReduce tasks of merged *insert query* (*Q1+Q6*) that reduces the scan operations.

Algorithm 1: Generating set of merged HiveQL queries.

```
1  Input $Q_{in}$= $(q_1,...,q_n)$; // HiveQL queries
2  Output $Q_{out}$= $(q'_1,...,q'_m)$, where $(m<=n)$; // merged HiveQL queries
3  $Q_{out}$:={} //initial empty list of merged queries
4  //$q_{miq}$ is a merged insert query;
5  for $q_i \in Q_{in}$ do
6      if $\exists q_{miq} \in Q_{out}$ such that isFullyCorrelated($q_{miq}, q_i$) then
7          MergeWithFullyCorrelatedInsertQuery($q_{miq}, q_i$);
8          $Q_{in} = Q_{in}$ - $\{q_i\}$;
9      else if $\exists q_j \in \{Q_{in} - q_i\}$ such that isFullyCorrelated($q_i$,$q_j$) then
10         $Q_{out} = Q_{out} \cup$ MergedInsertQuery($q_i, q_j$);
11         $Q_{in} = Q_{in} - \{q_i, q_j\}$;
12
13     else if $\exists q_{miq} \in Q_{out}$ such that isPartiallyCorrelated($q_{miq}, q_i$) then
14         MergeWithQuery($q_{miq}, q_i$);
15         $Q_{in} = Q_{in} - \{q_i\}$;
16
17     else
18         if $\exists q_j \in \{Q_{in} - q_i\}$ such that isPartiallyCorrelated($q_i$,$q_j$) then
19             $Q_{out} = Q_{out} \cup$ MergedInsertQuery($q_i, q_j$);
20             $Q_{in} = Q_{in}$ - $\{q_i,q_j\}$;
21 MergeRemainingNonCorrelatedQueries($Q_{out}, Q_{in}$);
```

In the Appendix, three merged queries are presented to explain the merging process of HiveQL queries that share common input and output parts. The first one merges two *Q1* queries that have different selection predicates, the second one merges two fully-correlated queries, *Q14* and *Q19*, that share a common join operation and the third one merges two partially correlated queries *Q1* and *Q18* [42].

HiveQL statements have a preprocessing overhead for MapReduce tasks that will be executed to complete a query and this causes high latencies that could cause short running queries to take longer time on Hive [43]. In addition to the emerging opportunities of using common table scan and join operations, SharedHive intends to decrease the preprocessing period of uncorrelated query MapReduce tasks.

In the merging process of SharedHive, each query is classified according to the shared tables and/or join operations in the FROM clause of HiveQL statements. The input queries are inserted into a data structure that maintains the groups of similar queries according to the largest sharing opportunity they have with other queries.

While grouping the queries, the highest precedence is given to (a) queries with fully-correlated FROM expressions, (b) queries with partially-correlated FROM expressions and (c) queries that have no correlation with the other queries (which are appended to the end of the set of merged queries) (see Algorithm 1). With this approach, the common scan/join tasks in merged *insert queries* are not executed repeatedly [15]. After the merging process, the optimized set of *insert queries* are passed to the query execution layer of Hive.

Experimental setup and results

In this section, experimental setup and the performance evaluation of the merged HiveQL *insert queries* are presented. TPC-H is chosen as our benchmark database and related decision support queries because they process high amounts of data [44]. We believe this is a good match for our experiments since Hadoop is also designed to process large amounts of data. 11 query sets are prepared from standard TPC-H queries to experimentally analyze performance of SharedHive under different workload scenarios. These query sets define three correlation categories for merged queries (uncorrelated, partially correlated, and fully correlated). Uncorrelated queries have nothing in common, partially-correlated queries share at least one table and zero or more join operations. Fully-correlated queries have exactly the same list of the tables/joins (where conditions have different selection predicates). Table 1 gives the selected set of queries and their correlation levels. Query sets 8 and 9 use single queries that are submitted several times with different selection predicates. Query set 8 executes no join operation so that it presents the performance gains of SharedHive with intensive scan sharing, whereas query set 9 includes common join operations that require communication between datanodes. Query sets 10 and 11 include queries that produce several merged insert queries.

Three different TPC-H decision support databases with sizes 1GB, 100GB and 1TB are used. Similar experimental settings are used in previous studies [22,40].

The experiments are performed on a private Cloud server, 4U DELL PowerEdge R910 having 32 (64 with Hyper Threading) cores. Each core is Intel Xeon E7-4820 with 2.0GHz processing power. The server has 128GB DDR3 1600MHz virtualized memory and Broadcom Nextreme II 5709 1Gbps NICs. Operating system of the physical server is Windows Server 2012 Standard Edition. 20 Linux CentOS 6.4 virtual machines are installed on this server as guest operating systems. Each virtual machine has two 2.0GHz processors, 2GB RAM and 250GB disk storage. An additional master node is used

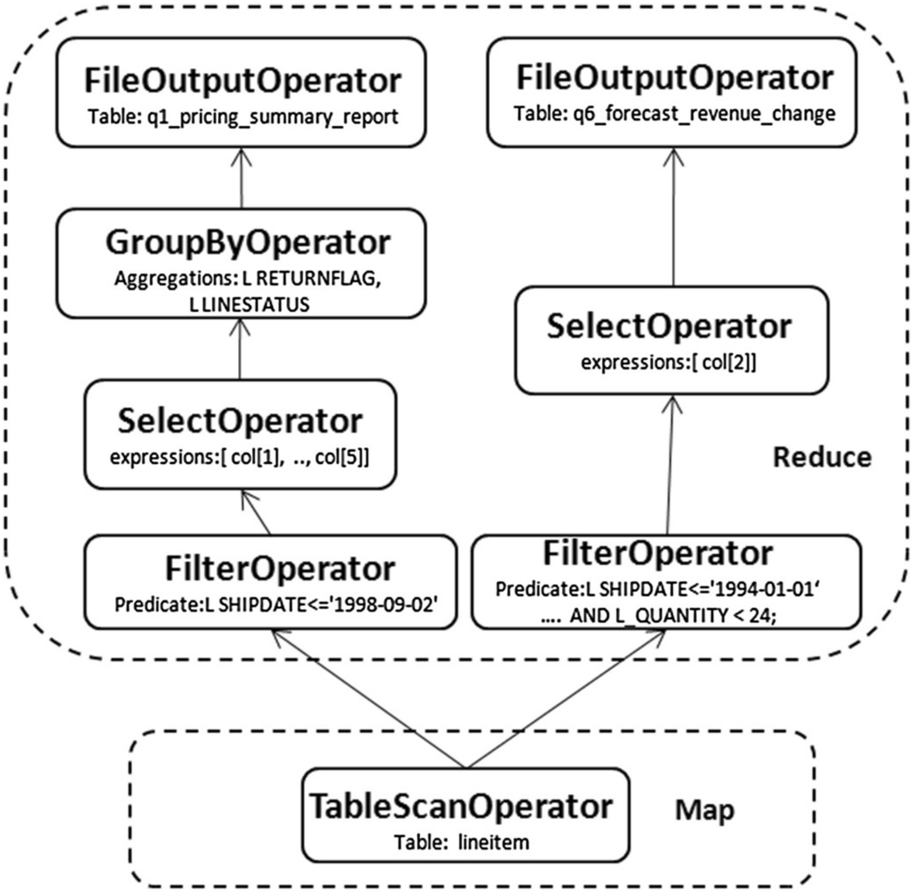
FileOutputOperator
Table: q1_pricing_summary_report
FileOutputOperator
Table: q6_forecast_revenue_change
GroupByOperator
Aggregations: L RETURNFLAG, L LINESTATUS
SelectOperator
expressions:[col[2]]
SelectOperator
expressions:[col[1], .., col[5]]
Reduce
FilterOperator
Predicate:L SHIPDATE<='1998-09-02'
FilterOperator
Predicate:L SHIPDATE<='1994-01-01' …. AND L_QUANTITY < 24;
TableScanOperator
Table: lineitem
Map

Figure 3 MapReduce tasks for merged insert query (*Q1+Q6*).

as NameNode/JobTracker (4 processors, 8GB RAM and 500GB disk storage). The latest stabilized versions of Hadoop, release 1.2.1 and Hive version 0.12.0 are used [1,11]. The splitsize of the files (HDFS block size) is 64MB, replication number is 2, maximum number of map tasks is 2, maximum number of reduce tasks is 2 and map output compression is disabled during the experiments.

Table 1 Sets of selected TPC-H queries and their correlation levels

Set number	TPC-H Query name	Correlation level
1	Q11,Q12	none
2	Q17,Q22	none
3	Q1,Q17	partial
4	Q1,Q18	partial
5	Q6,Q17	partial
6	Q1,Q6	full
7	Q14,Q19	full
8	Multiple *Q1s*	full
9	Multiple *Q3s*	full
10	Q1,Q14,Q18	mixed
11	Q1,Q3,Q11,Q14,Q17,Q19	mixed

In order to remove noise in performance measurements, the Cloud server is only dedicated to our experiment during the performance evaluation. Therefore, we believe that performance interference from external factors such as network congestion or OS-level contention on shared resources are minimized as much as possible. We observe that there were only negligible changes in the response time of the queries when we repeated our experiments three times.

Table 2 presents the response times of TPC-H queries (*Q1, Q3, Q6, Q11, Q12, Q14, Q17, Q18, Q19, Q22*) with 1GB, 100GB and 1TB database sizes. These results constitute baselines to compare the results of the merged HiveQL queries with single execution performance of Hive.

Tables 3 and 4 show the performance increases for the selected HiveQL query sets given in Table 1 that are merged and run to observe the effect of SharedHive on total response times. The percentage values show the

Table 2 Execution times (sec.) of single TPC-H queries

Query name	1GB	100GB	1TB
Q1	66	381	3,668
Q3	150	655	7,286
Q6	37	220	1,836
Q11	140	243	1,196
Q12	91	434	4,214
Q14	65	330	3,036
Q17	126	949	9,064
Q18	186	1,159	13,922
Q19	73	674	6,564
Q22	146	385	2,014

reduction of the response time. Significant performance increases can be seen easily.

Although uncorrelated queries have nothing in common, their total execution times are observed to reduce by 0.2%-6.9% due to the improvement in HIVE query preprocessing overheads. Merging uncorrelated queries does not increase the performance when the database size reaches terabyte scale. The reductions in total execution times of partially correlated queries is higher than uncorrelated query sets (between 1.5%-20.8%). The highest benefits are observed in the fully correlated query sets (between 9.9%-39.9%). For query set 8 (single *Q1* query submitted 8 times) the total query execution time is reduced from 26,716 to 3,985 seconds (85.1% reduction). The performance of mixed query sets depends on the correlation level of the queries they contain. Mixed query sets 10 and 11 execute their queries with 9.9% and 15.5% less execution times, respectively. During these experiments, the size of the intermediate tables that are written to the disks is considered carefully by SharedHive. If predicted overhead of writing intermediate results is larger than the expected improvement in response time, then queries are not merged. SharedHive is observed to reduce the number of MapReduce tasks and the sizes of read/written HDFS files as well. The results given in Table 5 present the effect of SharedHive for the number of MapReduce tasks and the sizes of read/written files of the given *insert queries* (having different correlation levels). As the correlation level of queries increases the number of MapReduce tasks and the sizes of read/written data also decreases substantially.

The optimization time of SharedHive on analyzing and merging the queries is observed to be small. This is because of the small number of input queries and executing Algorithm 1 on them requires only examination of their FROM clauses which are parsed to identify similar expressions and rewriting the merged HiveQL query. This optimization does not take more than a few milliseconds.

In the last phase of our experiments, SharedHive is run on five different cluster sizes to observe its scalability with increasing number of datanodes. First, the merged *insert query* (*Q14+Q19*) is executed on the cluster using three different database sizes (1GB, 10GB and 100GB). It is observed that increasing the number of datanodes in the cluster improves the performance of the merged query reducing execution times by 29%, 81% and 88% in the database instances when the number of datanodes is increased from 1 to 20 (see Figure 4).

MQO component of SharedHive is an extension to Hive and welcomes any performance increase that is achieved on the HDFS layer either due to increase in the number of datanodes or balanced distribution of data files.

Comparison with other MapReduce-based MQO systems

SharedHive can perform the execution of the selected/correlated queries in shorter times than Hive by reducing the number of MapReduce tasks and the sizes of the files read/written by the tasks. The correlation detection mechanism of SharedHive is simple and does not find the number common rows and/or columns of

Table 3 Execution times of sequential and merged queries in seconds

Query names	*Correlation*	Execution time (sec.) Hive/SharedHive (reduction %)		
		1GB	100GB	1TB
11,12 (set 1)	none	231/215 (6.9%)	676/666 (1.5%)	5,410/5,386 (0.4%)
17,22 (set 2)	none	272/262 (3.7%)	1,382/1.323 (4,3%)	11,078/11,060 (0.2%)
1,17 (set 3)	partial	192/185 (3.6%)	1,330/1,134 (14.7%)	12,300/11,386 (7.4%)
1,18 (set 4)	partial	252/248 (1.6%)	1,540/1,396 (9.4%)	17,499/16,324 (6.7%)
6,17 (set 5)	partial	163/160 (1.8%)	1,169/1,042 (10.9%)	10,430/8,636 (17.2%)
1,6 (set 6)	full	103/90 (12.6%)	601/436 (27.5%)	5,057/3,936 (22.2%)
14,19 (set 7)	full	138/83 (39.9%)	1,004/789 (21.9%)	8,989/7,178 (20.1%)
1,14,18 (set 10)	mixed	317/299 (5.7%)	1,870/1,689 (9.7%)	20,425/18,590 (9.0%)
1,3,11,14,17,19 (set 11)	mixed	620/524 (15.5%)	3,232/2,830 (12.4%)	30,069/26,024 (13.5%)

Table 4 Execution times of sequential and merged query sets (8 and 9) in seconds

Query name	*# of submitted queries*	Execution time (sec.) Hive/SharedHive (reduction %)		
		1GB	**100GB**	**1TB**
Q1 (set 8)	2	123/70 (44.9%)	756/430 (43.1%)	6,833/3,843 (43.8%)
	4	227/77 (66.1%)	1,435/442 (69.2%)	13,365/3,928 (70.6%)
	8	444/79 (82.2%)	2,894/458 (84.2%)	26,716/3,985 (85.1%)
Q3 (set 9)	2	246/176 (28.5%)	1,334/999 (25.1%)	15,546/14,316 (7.9%)
	4	470/307 (34.7%)	2,626/1,333 (49.2%)	27,712/14,645 (47.2%)
	8	957/503 (47.4%)	5,486/1,466 (73.3%)	56,112/15,615 (72.2%)

queries with complex algorithms as in [19,29]. The execution time performance gains are observed to be within the range of %1.5-85.1% in accordance with the correlation level of the queries. For repeatedly issued similar queries that have different predicates, SharedHive performs well. SharedHive benefits from underlying HDFS architecture therefore, its scalability is preserved and better performance is obtained when additional datanodes are introduced to Hadoop. The query results obtained by SharedHive have been compared with those of Hive and verified to be the same.

MRShare [40] is a recent MQO system developed for benefitting from multiple queries containing similar MapReduce tasks. It transforms a batch of queries into a new batch that will be executed more efficiently by merging jobs into groups and evaluating each group as a single query. MRShare optimizes queries that work on the same input table and does not consider sharing of join operations. However, SharedHive can merge queries containing joins into a new set of *insert queries*. MRShare shares scan tasks by creating a single job for multiple jobs and does not use temporary files (as it is done by SharedHive).

YSmart [22] is a correlation-aware MQO system similar to SharedHive. It detects and removes redundant MapReduce tasks of single complex queries but does not optimize multiple queries. The developers of YSmart present experimental results that significantly outperform conventional Hive for single queries. SharedHive does not provide any performance increase for single queries unless they are submitted several times (with different predicates). SharedHive works in the application layer of Hive by merging the query level operations, whereas MRShare and YSmart explore and eliminate redundant tasks in the MapReduce layer.

Apache Pig is the most mature MapReduce-based platform that supports a large number of sharing mechanisms among multiple queries [37]. Complex tasks consisting of multiple interrelated data transformations are explicitly encoded as data flow sequences; however, its query language, Pig Latin, is not compatible with standard SQL statements like SharedHive.

Conclusions and future work

In this study, we propose a multiple query optimization (MQO) framework, SharedHive, for improving the performance of MapReduce-based data warehouse Hadoop Hive queries. To our knowledge, this is the first work that aims at improving the performance of Hive with MQO techniques. In SharedHive, we detect common tasks of correlated TPC-H HiveQL queries and merge them into a new set of global Hive *insert queries*. With this approach, it has been experimentally shown that significant performance improvements can be achieved by reducing the number of MapReduce tasks and the total sizes of read/written files.

As future work, we plan to incorporate MQO functionality at MapReduce layer, similar to YSmart, into SharedHive. In this way, it will be possible to eliminate even more redundant MapReduce tasks in queries and improve

Table 5 Comparing the number of MapReduce tasks and the sizes of read/written HDFS files by Hive and SharedHive for different correlation level 100GB TPC-H data warehouse queries

Query set (correlation level)	**# Map tasks**		**# Reduce tasks**		**Read (GB)**		**Written (GB)**	
	Hive	**S.Hive**	**Hive**	**S.Hive**	**Hive**	**S.Hive**	**Hive**	**S.Hive**
11,12 (set 1) (none)	463	463	116	116	123	123	20	20
6,17 (set 5) (partial)	1,568	663	326	89	51,561	51,561	504,975	34,806
14,19 (set 7) (full)	668	334	280	84	356	171	168	2
1,3,11,14,17,19 (set 11) (mixed)	2,149	1,150	452	288	636	404	291	206

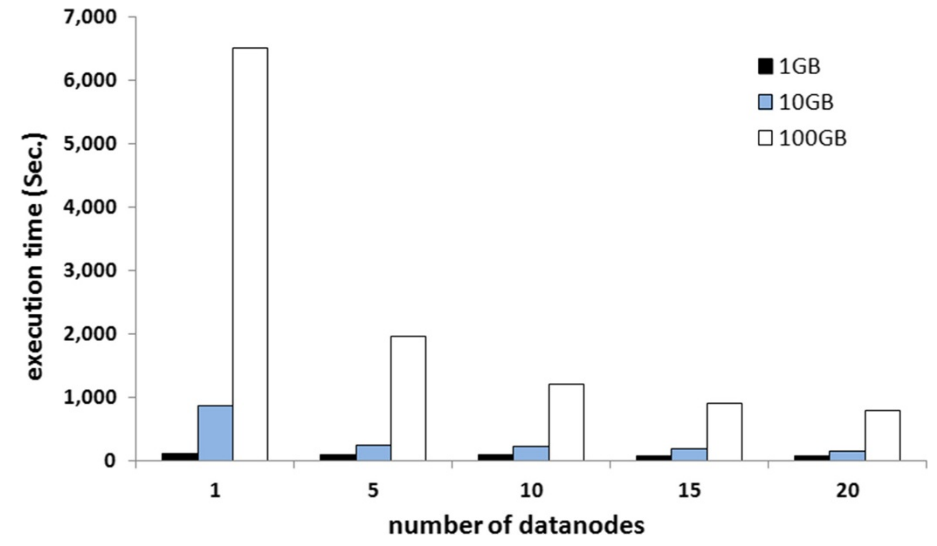

Figure 4 The effect of increasing number of datanodes for merged query (*Q14*+*Q19*) with 1GB, 10GB and 100GB database sizes.

the overall performance of naïve rule-based Hive query optimizer even further.

Appendix

A. Merging two *Q1* queries that have different select predicates

```
First query
    SELECT L_RETURNFLAG ,..., COUNT(*)
    FROM LINEITEM
    WHERE L_SHIPDATE ≤ '1998-09-04'
    GROUP BY L_RETURNFLAG, L_LINESTATUS
    ORDER BY L_RETURNFLAG, L_LINESTATUS;
Second query
    SELECT L_RETURNFLAG ,..., COUNT(*)
    FROM LINEITEM
    WHERE L_SHIPDATE > '1992-05-12'
    GROUP BY L_RETURNFLAG, L_LINESTATUS
    ORDER BY L_RETURNFLAG, L_LINESTATUS;

Merged Query
FROM
  (SELECT L_RETURNFLAG,...,COUNT(*), L_SHIPDATE
  FROM LINEITEM
  WHERE L_SHIPDATE ≤ '1998-09-04' OR L_SHIPDATE>
    '1992-05-12'
  GROUP BY L_RETURNFLAG, L_LINESTATUS,
    L_SHIPDATE
  ORDER BY L_RETURNFLAG, L_LINESTATUS,
    L_SHIPDATE) temp
INSERT OVERWRITE TABLE q1_pricing_summary_
report_1
  SELECT temp.L_RETURNFLAG ,..., COUNT(*)
  WHERE temp.L_SHIPDATE ≤ '1998-09-04'
  GROUP BY L_RETURNFLAG, L_LINESTATUS
  ORDER BY L_RETURNFLAG, L_LINESTATUS
INSERT OVERWRITE TABLE q1_pricing_summary_
report_2
  SELECT temp.L_RETURNFLAG ,..., COUNT(*)
  WHERE temp.L_SHIPDATE > '1992-05-12'
  GROUP BY L_RETURNFLAG, L_LINESTATUS
  ORDER BY L_RETURNFLAG, L_LINESTATUS;
```

B. Merging queries *Q14* and *Q19* (Fully correlated FROM clauses)

```
Query Q14
  SELECT ...
  FROM LINEITEM L JOIN PART P ON L.L_PARTKEY =
    P.P_PARTKEY
  AND L.L_SHIPDATE ≥ '1995-09-01' AND
    L.L_SHIPDATE <'1995-10-01';
Query Q19
  SELECT ...
  FROM
  LINEITEM L JOIN PART P ON L.L_PARTKEY =
    P.P_PARTKEY ;
  WHERE ... ;

Merged Query (Q14 + Q19)
  FROM LINEITEM L JOIN PART P ON L.L_PARTKEY =
    P.P_PARTKEY
INSERT OVERWRITE TABLE q14_promotion_effect
  SELECT ...
  WHERE L.L_SHIPDATE ≥ '1995-09-01' AND
    L.L_SHIPDATE < '1995-10-01'
  INSERT OVERWRITE TABLE q19_discounted_revenue
  SELECT ...
  WHERE ...;
```

C. Merging queries *Q1* and *Q18* (Partially correlated FROM clauses)

```
Query Q1
  SELECT L_RETURNFLAG ,..., COUNT(*)
  FROM LINEITEM
  WHERE L_SHIPDATE ≤ '1998-09-02'
  GROUP BY L_RETURNFLAG, L_LINESTATUS
  ORDER BY L_RETURNFLAG, L_LINESTATUS;

Query Q18
  INSERT OVERWRITE TABLE Q18_TMP
  SELECT L_ORDERKEY, SUM(L_QUANTITY) AS
    T_SUM_QUANTITY
  FROM LINEITEM
  WHERE L_SHIPDATE≤'1993-01-01'
  GROUP BY L_ORDERKEY;
```

```
INSERT OVERWRITE TABLE Q18_LARGE_VOLUME_
  CUSTOMER
SELECT C_NAME ,..., SUM(L_QUANTITY)
FROM CUSTOMER C JOIN ORDERS O
ON C.C_CUSKEY = O.O_CUSKEY JOIN Q18_TMP T ON
O.O_ORDERKEY = T.L_ORDERKEY AND T.T_SUM_
  QUANTITY > 300
JOIN LINEITEM L ON O.O_ORDERKEY =
  L.L_ORDERKEY
GROUP BY C_NAME, C_CUSKEY, O_ORDERKEY,
  O_ORDERDATE,O_TOTALPRICE
ORDER BY O_TOTALPRICE DESC, O_ORDERDATE;
```

Merged Query (*Q1* + *Q18*)

```
FROM LINEITEM

SELECT L_RETURNFLAG ,..., COUNT(*)
WHERE L_SHIPDATE ≤ '1998-09-02'
GROUP BY L_RETURNFLAG, L_LINESTATUS
ORDER BY L_RETURNFLAG, L_LINESTATUS

INSERT OVERWRITE TABLE Q18_TMP
SELECT L_ORDERKEY, SUM(L_QUANTITY) AS
  T_SUM_QUANTITY
WHERE L_SHIPDATE≤'1993-01-01'
GROUP BY L_ORDERKEY;

INSERT OVERWRITE TABLE Q18_LARGE_VOLUME_
  CUSTOMER
SELECT C_NAME ,..., SUM(L_QUANTITY)
FROM CUSTOMER C JOIN ORDERS O
ON C.C_CUSKEY = O.O_CUSKEY JOIN Q18_TMP T ON
O.O_ORDERKEY = T.L_ORDERKEY AND
  T.T_SUM_QUANTITY > 300
JOIN LINEITEM L ON O.O_ORDERKEY =
  L.L_ORDERKEY
GROUP BY C_NAME, C_CUSKEY, O_ORDERKEY,
  O_ORDERDATE, O_TOTALPRICE
ORDER BY O_TOTALPRICE DESC, O_ORDERDATE;
```

Abbreviations
HDFS: Hadoop distributed file system; RDBMS: Relational database management system; SQL: Structured query language; QL: Query language; MQO: Multiple-query optimization; UDF: User-defined functions; CLI: Command line interface; DAG: Directed acyclic graph.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
TD designed the algorithm for merging the (fully, partially correlated) queries of Hive and executed the experiments. SO implemented the new Multiple Query Optimization component and added it to Hive framework. MAB prepared the mathematical formulation of Multiple Query optimization process for Hive and drafted the manuscript. MSC prepared the Hadoop/Hive experimental setup. AC coordinated the whole study and prepared the related work section. All authors read and approved the final manuscript.

Acknowledgements
We sincerely thank all the researchers in our references section for the inspiration they provide.

Author details
[1]Middle East Technical University Computer Engineering Department, Cankaya, Ankara, Turkey. [2]Microsoft Research Redmond, One Microsoft Way, Redmond, Washington 98052, USA. [3]Hacettepe University Computer Engineering Department, Cankaya, Ankara, Turkey.

Received: 9 May 2014 Accepted: 20 June 2014
Published online: 29 July 2014

References
1. Apache Hadoop. http://hadoop.apache.org/. Last Accessed 1 February 2014
2. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
3. Condie T, Conway N, Alvaro P, Hellerstein JM, Elmeleegy K, Sears R (2010) MapReduce online In: Proceedings of the 7th, USENIX conference on Networked systems design and implementation, April 28–30, San Jose, California, pp 21–21
4. Stonebraker M, Abadi D, DeWitt DJ, Madden S, Paulson E, Pavlo A, Rasin A (2010) MapReduce and parallel DBMSs: friends or foes? Commun ACM 53(1):64–71
5. DeWitt D, Stonebraker M (2008) MapReduce: A major step backwards. The Database Column, 1
6. He Y, Lee R, Huai Y, Shao Z, Jain N, Zhang X, Xu Z (2011) Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems In: Proceedings of the 2011 IEEE 27th International, Conference on Data Engineering, April 11-16, pp 1199–1208
7. Kang U, Tsourakakis CE, Faloutsos C (2011) Pegasus: mining peta-scale graphs. Knowl Inf Syst 27(2):303–325
8. Grolinger K, Higashino WA, Tiwari A, Capretz MA (2013) Data management in cloud environments: NoSQL and NewSQL data stores. J Cloud Comput: Adv Syst Appl 2(1):22
9. Bayir MA, Toroslu IH, Cosar A, Fidan G (2009) Smart miner: a new framework for mining large scale web usage data In: Proceedings of the 18th international conference on World wide web. ACM, pp 161–170
10. Abouzeid A, Bajda-Pawlikowski K, Abadi D, Silberschatz A, Rasin A (2009) HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proc VLDB 2(1):922–933
11. Hadoop Hive project. http://hadoop.apache.org/hive/. Last Accessed 4 January 2014
12. Dai W, Bassiouni M (2013) An improved task assignment scheme for Hadoop running in the clouds. J Cloud Comput: Adv Syst Appl 2(1):1–16
13. Issa J, Figueira S (2012) Hadoop and memcached: performance and power characterization and analysis. J Cloud Comput: Adv Sys Appl 1(1):1–20
14. Ordonez C, Song IY, Garcia-Alvarado C (2010) Relational versus non-relational database systems for data warehousing In: Proceedings of the ACM 13th international workshop on Data warehousing and OLAP, October 30–30, Toronto, ON, Canada
15. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Zhang N, Murthy R (2010) Hive-a petabyte scale data warehouse using hadoop In: Proceedings of ICDE, pp 996–1005
16. Sellis TK (1988) Multiple-query optimization. ACM Trans Database Syst (TODS) 13(1):23–52
17. Bayir MA, Toroslu IH, Cosar A (2007) Genetic algorithm for the multiple-query optimization problem. IEEE Trans Syst Man Cybernet Part C Appl Rev 37(1):147–153
18. Cosar A, Lim EP, Srivastava J (1993) Multiple query optimization with depth-first branch-and-bound and dynamic query ordering In: Proceedings of the second international conference on, Information and knowledge management. ACM, pp 433–438
19. Zhou J, Larson PA, Freytag JC, Lehner W (2007) Efficient exploitation of similar subexpressions for query processing In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, pp 533–544
20. Chaiken R, Jenkins B n, Larson PÃ, Ramsey B, Shakibn D, Weaver S, Zhou J (2008) SCOPE: easy and efficient parallel processing of massive data sets. Proc VLDB 1(2):1265–1276
21. Cohen J, Dolan B, Dunlap M, Hellerstein JM, Welton C (2009) MAD skills: new analysis practices for big data. Proc VLDB 2(2):1481–1492
22. Lee R, Luo T, Huai Y, Wang F, He Y, Zhang X (2011) Ysmart: Yet another sql-to-mapreduce translator In: Distributed Computing Systems (ICDCS), 2011 31st International, Conference on. IEEE, pp 25–36
23. Finkelstein S (1982) Common expression analysis in database applications In: Proceedings of the 1982 ACM SIGMOD international conference on, Management of data. ACM, pp 235–245

24. Roy P, Seshadri S, Sudarshan S, Bhobe S (2000) Efficient and extensible algorithms for multi query optimization. ACM SIGMOD Rec 29(2):249–260
25. Giannikis G, Alonso G, Kossmann D (2012) SharedDB: killing one thousand queries with one stone. Prof VLDB 5(6):526–537
26. Chen F, Dunham MH (1998) Common subexpression processing in multiple-query processing. IEEE Trans Knowl Data Eng 10(3):493–499
27. Mehta M, DeWitt DJ (1995) Managing intra-operator parallelism in parallel database systems In: VLDB vol 95, pp 382–394
28. Beynon M, Chang C, Catalyurek U, Kurc T, Sussman A, Andrade H, Ferreira R, Saltz J (2002) Processing large-scale multi-dimensional data in parallel and distributed environments. Parallel Comput 28(5):827–859
29. Chen G, Wu Y, Liu J, Yang G, Zheng W (2011) Optimization of sub-query processing in distributed data integration systems. J Netw Comput Appl 34(4):1035–1042
30. Lee R, Zhou M, Liao H (2007) Request Window: an approach to improve throughput of RDBMS-based data integration system by utilizing data sharing across concurrent distributed queries In: Proceedings of the 33rd international conference on, Very large data bases. VLDB Endowment, pp 1219–1230
31. Harizopoulos S, Shkapenyuk V, Ailamaki A (2005) QPipe: a simultaneously pipelined relational query engine In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data. ACM, pp 383–394
32. Le W, Kementsietsidis A, Duan S, Li F (2012) Scalable multi-query optimization for SPARQL In: Data Engineering (ICDE), 2012 IEEE 28th International Conference on. IEEE, pp 666–677
33. Silva YN, Larson PA, Zhou J (2012) Exploiting common subexpressions for cloud query processing In: Data Engineering (ICDE), 2012 IEEE 28th International Conference on. IEEE, pp 1337–1348
34. Wang X, Olston C, Sarma AD, Burns R (2011) CoScan: cooperative scan sharing in the cloud In: Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, p 11
35. Wolf J, Balmin A, Rajan D, Hildrum K, Khandekar R, Parekh S, Wu K-L, Vernica R (2012) On the optimization of schedules for MapReduce workloads in the presence of shared scans. VLDB J 21(5):589–609
36. Ferrera P, De Prado I, Palacios E, Fernandez-Marquez JL, Serugendo GDM (2013) Tuple MapReduce and Pangool: an associated implementation. Knowl Inf Syst:1–27. doi:10.1007/s10115-013-0705-z
37. Apache Pig. http://pig.apache.org/. Last Accessed 1 February 2014
38. Bajda-Pawlikowski K, Abadi DJ, Silberschatz A, Paulson E (2011) Efficient processing of data warehousing queries in a split execution environment In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, pp 1165–1176
39. Friedman E, Pawlowski P, Cieslewicz J (2009) SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. Proc VLDB 2(2):1402–1413
40. Nykiel T, Potamias M, Mishra C, Kollios G, Koudas N (2010) MRShare: sharing across multiple queries in mapreduce. Proc VLDB 3(1–2):494–505
41. Gruenheid A, Omiecinski E, Mark L (2011) Query optimization using column statistics in hive In: Proceedings of the 15th, Symposium on International Database Engineering & Applications. ACM, pp 97–105
42. Chaudhuri S, Shim K (1994) Including group-by in query optimization In: VLDB vol. 94, pp 354–366
43. Jiang D, Tung AK, Chen G (2011) Map-join-reduce: toward scalable and efficient data analysis on large clusters. Knowl Data Eng IEEE Trans 23(9):1299–1311
44. Running TPC-H queries on Hive. http://issues.apache.org/jira/browse/HIVE-600. Last Accessed 1 January 2014

doi:10.1186/s13677-014-0012-6
Cite this article as: Dokeroglu *et al.*: **Improving the performance of Hadoop Hive by sharing scan and computation tasks.** *Journal of Cloud Computing: Advances, Systems and Applications* 2014 **3**:12.