Sharing Scan and Computation Tasks in Hadoop Hive

Serkan Ozal, Tansel Dokeroglu, Murat Ali Bayir, and Ahmet Cosar

Abstract—MapReduce is a popular way of executing a time-consuming analytical query as a batch of tasks on large scale data. In simultaneous execution of multiple queries, many opportunities can arise for sharing scan and/or computation tasks. Executing common tasks only once can reduce the total execution time remarkably. Therefore, we propose to use the Multiple Query Optimization (MQO) technique to improve the overall performance of Hadoop Hive, an open source SQL-based distributed warehouse system working on MapReduce. Our framework, SharedHive, transforms a set of HiveQL queries into a new global query that will produce all of the required outputs which can be executed in remarkably smaller total execution time. It is experimentally shown that SharedHive outperforms the conventional Hive with 20-90% reduction in the total execution time of correlated TPC-H queries.

Index Terms—Hadoop, Hive, Multi-query Optimization, Data Warehouse

1 INTRODUCTION

HADOOP [1] is a popular open source software framework that allows distributed processing of large scale data sets. It employs MapReduce paradigm [2] to divide the computation tasks into parts (Figure 1) that can be distributed to a cluster of commodity therefore, providing horizontal scalability [3], [4], [5], [6]. Hadoop Distributed File System (HDFS) is the underlying file system of Hadoop MapReduce. Because of its simplicity, scalability, fault-tolerance, and efficiency Hadoop has gained significant support from both industry and academia. However, it has some limitations on its interfaces and performance [7]. Querying the data with Hadoop as if 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 internal details of Map Reduce jobs to extract information from their data warehouses.

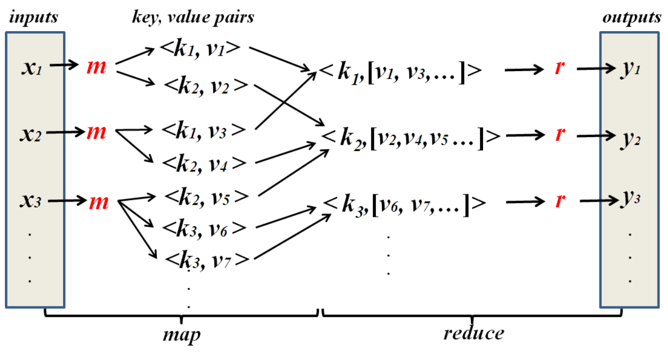

Fig. 1. MapReduce tasks.

- *Murat Ali Bayir is currently with Google Inc.*
- *Ahmet Cosar is a full-time faculty member in Middle East Technical University Ankara/TURKEY. see http://www.ceng.metu.edu.tr/ cosar/*

Hadoop Hive, an open source SQL-based distributed warehouse system is proposed to solve problems mentioned above by providing SQL like abstraction on top of Hadoop framework. Hive is a SQL-to-MapReduce translator and has an SQL dialect, HiveQL, for querying data stored in a cluster [8]. When users want to benefit from both MapReduce and SQL interface, mapping SQL statements to MapReduce tasks can become a very difficult job [9]. Hive does this work by translating queries to MapReduce jobs, thereby exploiting the scalability of Hadoop while presenting a familiar SQL abstraction [10]. 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 data is not updated frequently [4].

Because most data warehouse applications are implemented using SQLbased RDBMSs, Hive lowers the barrier for moving these applications to Hadoop, thus, people who already know SQL can use Hive easily. Similarly, Hive makes it easier for developers to port SQL-based applications to Hadoop. Since Hive is based on 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 that there has not been any study that incorporates the Multiple-query optimization (MQO) technique [11], [12], [13] for Hive to reduce the total execution time of the queries.

Studies concerning MQO on traditional warehouses have shown that it is an efficient technique that increases the performance of time-consuming decision support queries remarkably [2], [14], [15], [16]. 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 Hive query optimizer [10], [17], [18], [19]. Our contributions in this study can be listed as:

(1) MQO technique is used by an SQL-to-MapReduce translator for the first time.

(2) The execution plans of correlated HiveQL queries are analyzed, merged (with an optimization algorithm), and executed together.

(3) The developed model is introduced as a novel component for Hadoop Hive architecture.

In Section 2, we give brief information about the related work on MQO, other SQL-to-MapReduce translators that are similar to Hive, and recent query optimization studies on MapReduce framework. Section 3 explains traditional architecture of Hive and introduces our novel MQO component. Section 4 explains the process/algorithm of generating a global plan from correlated queries. Section 5 discusses the experiments conducted for evaluating SharedHive framework. Finally our concluding remarks are given in Section 6.

2 Related Work

MQO problem was introduced in 1980s and finding an optimal global query plan by using MQO was shown to be an NP-Hard problem [11], [20]. Since then, considerable amount of work was done on RDBMSs and data analysis applications [21], [22], [23]. 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 [24]. A proxy-based infrastructure for handling data intensive applications is proposed by Beynon [25]. This infrastructure was not as scalable as a collection of distributed cache servers available at multiple back-ends. Chen et al. considered the network layer of a data integration system and reduced the communication costs by a multiple query reconstruction algorithm [26]. IGNITE [27] and QPipe [28] 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 in [29]. Yasin et al. designed a cascade-style optimizer for Scope, Microsoft's system for massive data analysis [30].

In recent years, a significant amount of research and commercial activity has focused on integrating MapReduce and structured databases technologies. Mainly there are two approaches: Either adding MapReduce features to parallel database or adding databases technology to MapReduce. The second approach is more attractive because there exists no widely available open source parallel database system whereas MapReduce is available as an open source project. Furthermore, MapReduce is accompanied by a plethora of free tools as well as cluster availability and support. Hive [8], Pig [31], Scope [15], and HadoopDB [7], [32] are the projects that provide SQL abstractions (SQL-to-MapReduce translators) on top of MapReduce platform to familiarize the programmers with complex queries. SQL/MapReduce [33] and Greenplum [16] are recent projects that use MapReduce to process user-defined functions (UDF). Recently, there are interesting studies to apply MQO to MapReduce frameworks for unstructured data. MRShare [34] is one of these studies that 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 [35], to our knowledge there is no study like ours that is related to optimize the execution time of multi-queries on SQL-to-MapReduce translator tools.

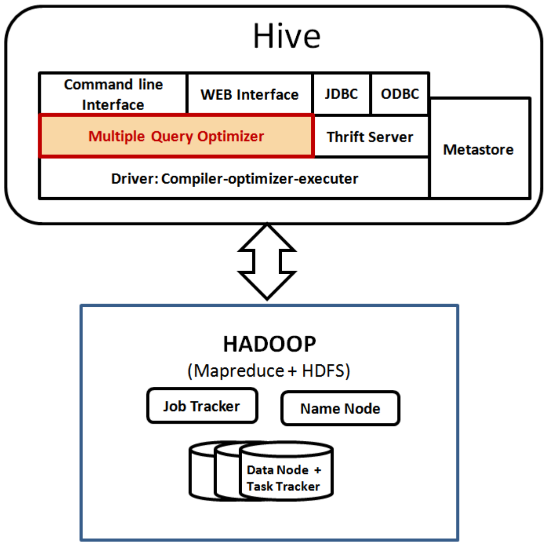

Fig. 2. SharedHive, A novel Hadoop Hive system architecture with MQO support.

3 Shared Hive System Architecture

In this Section, we give brief information about architecture of Shared Hive which is the modified version of Hadoop Hive with new MQO component. Figure 2 shows the architecture of SharedHive. 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 to answer the query. First, a directed acyclic graph is produced by HiveQL to define the MapReduce tasks to be executed. Next, the tasks are executed.

HiveQL statements are submitted via the Command Line Interface (CLI), the Web User Interface, the thrift, ODBC, or JDBC interfaces. Normally the query is directed to the driver component in a conventional Hive architecture. In the architecture we proposed, MQO component (located after the client interface) receives the incoming queries before the driver component. The set of the incoming queries are inspected, their common tasks (redundant join processes) are detected, and merged with a global HiveQL query that answers all the incoming queries. Details of this process is explained in Section 4. The driver component passes the global query to the Hive compiler that produces a logical plan using information in Metastore and optimize this plan using a single rule-based optimizer. The execution engine receives a directed acyclic graph of MapReduce tasks and associated HDFS tasks and executes it in accordance with the dependencies of the tasks.

The new MQO component does not require any big changes in the system architecture of Hadoop Hive. Therefore, it can be integrated easily by Hive without requiring many modifications. Other SQL-to-MapReduce translators can take advantage of this technique as well.

4 Multiple-query Optimization on HiveQL Queries

In this Section, we introduce our global query construction algorithm for HiveQL queries. The optimization algorithm constructs a global HiveQL query from a set of correlated queries and produces answers for all of the input queries as HDFS files for further processing.

Algorithm 1 Generating Global HiveQL Queries.

```
Input Q$_{org}$=HiveQL (q$_1$ ,..., q$_n$);
Output Q$_{opt}$=HiveQL (q$_1'$ ,..., q$_m'$), where (m<=n);

Q$_{opt}$:=create_HiveQL_list (); //optimized query list

for q$_i \in$ Q$_{org}$ do
  FROM_clause=get_FROM_clause(q$_i$);
  MAP[FROM_clause]:=q$_i \cup$ map[FROM_clause];
end for

for k$_i \in$ MAP.key do
  q$_{list}$:=MAP[k$_i$];
  if ($q_{list}.size()$==1) then
    add_query (Q$_{opt}$, q$_{list}[0]$)
  else
    new_query_node = create_HiveQL (k$_i$);
    for (q$_j \in q_{list}$) do
      (q$_j := q_j - k_i$);
      add_query (new_query_node, q$_j$);
    end for
    add_query_node (Q$_{opt}$, new_query_node);
  end if
end for
```

Symbols q_i and Q denote a single query and the set of the incoming queries respectively.

The MQO formulation used by SharedHive can be given formally as :

Input: a set of queries Q_1={q_1,...,q_n}.
Output: a set of modified queries Q_1'={q_1',...,q_m'}.

The total execution time of queries in Q_1' is less than Q_1.

$$\sum_{i=1}^{m} execution\_time(q_i') < \sum_{i=1}^{n} execution\_time(q_i)$$

If q$_m$ is the merged query of q_i and q_j then all of the tuples and columns required by both queries must be produced by q_m by preserving the attributes of the predicates of q_i and q$_j$.

The architecture of Hive cannot evaluate more than one HiveQL query and produces many jobs that run in parallel to complete the solution . The output that the global HiveQL query produced by our MQO component is compatible with Hive optimizer.

Hive allows many queries to share the same FROM clause. For example given

```
q1 :
INSERT OVERWRITE TABLE <q1_result>
  SELECT ... FROM <LINEITEM>
  WHERE <condition_1>;
q2 :
INSERT OVERWRITE TABLE <q2_result>
  SELECT ... FROM <LINEITEM>
  WHERE <condition_2>;
```

it is possible to merge q$_1$ and q_2 into q_m as a HiveQL statement :

```
 qm :
FROM <LINEITEM>
  INSERT OVERWRITE TABLE <query_1_result>
    SELECT ... <condition_1>
  INSERT OVERWRITE TABLE <query_2_result>
    SELECT ... <condition_2>;
```

In the merging process, first we generate the query execution plans of the input queries and classify each query execution plan according to the related tables in the FROM clause of HiveQL statements. FROM clause of HiveQL queries can have a single relation or more. Therefore, our algorithm finds a global plan to execute the queries by merging them. These query plans are inserted into a tree structure that maintains the similar queries of the parent query node. Children of this tree are the query plans that share the same tables with the parent query node. Therefore, the input relations are not scanned repeatedly while executing the child queries. This merged HiveQL query is passed to the query execution layer of Hive. The detailed explanation of the merging process is given in Algorithm 1.

The example below shows a merged global HiveQL query from two correlated TPC-H [36] queries (Q1 and

Q6). The output for each original query is written to separate files qr_1 and qr_2 after the execution of the merged query is completed which adds a minimal overhead to the global query execution time.

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
   SELECTL_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 Global Query for Q1 and 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;
```

At the last phase of execution, the "INSERT OVERWRITE TABLE" part of each merged HiveQL query is converted into its original form and the reducer tasks write the result of the query to its corresponding query result file, qr_i.

4.1 Processing Multiple Instances of the Same Query with Different Predicates

In this Section, we analyze the case where we can merge the queries that are the instances of the same query with different predicates. For example, TPC-H Q1, Pricing Report Query, provides a summary pricing report for all lineitems shipped as of a given date. The only varying parameter for Q1 is the starting day of the shipping. The conventional Hive system processes each one of these queries sequentially by generating unrelated distinct tasks. Although our proposed method does not optimize a given single HiveQL query, it can efficiently optimize a set of similar queries with different predicates (and with the same FROM clause). Instead of repeatedly scanning the same relations, we have introduced a simple task for each query in Hive that writes the results to a file when it detects a completed result for any of the queries. These tasks are parallelized by Hive infrastructure and by scanning the relation (lineitem) only once, the results of different queries can be obtained efficiently.
You can see below a global query for three Q1 queries with different predicates.

```
CREATE EXTERNAL TABLE LINEITEM
(L_ORDERKEY INT,...,L_COMMENT STRING)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '—' STORED AS TEXTFILE LOCATION
'/tpch/LINEITEM';

CRATE TABLE q1_pricing_summary_report_1
(L_RETURNFLAG STRING,..., COUNT_ORDER INT);

CREATE TABLE q1_pricing_summary_report_2
(L_RETURNFLAG STRING,..., COUNT_ORDER INT);

CREATE TABLE q1_pricing_summary_report_3
(L_RETURNFLAG STRING,..., COUNT_ORDER INT);

FROM LINEITEM

INSERT OVERWRITE TABLE q1_pricing_summary_report_1
   SELECT L_RETURNFLAG ,..., COUNT(*)
   WHERE L_SHIPDATE <= '1998-08-02'
   GROUP BY L_RETURNFLAG, L_LINESTATUS
   ORDER BY L_RETURNFLAG, L_LINESTATUS

INSERT OVERWRITE TABLE q1_pricing_summary_report_2
   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 q1_pricing_summary_report_3
   SELECT L_RETURNFLAG ,..., COUNT(*)
   WHERE L_SHIPDATE <= '1998-10-02'
   GROUP BY L_RETURNFLAG, L_LINESTATUS
   ORDER BY L_RETURNFLAG, L_LINESTATUS ;
```

4.2 Experimental Environment

We have run our queries 10 times and reported their averages. The observed variance in the experiments were negligible. We performed experiments with TPC-H queries adapted to HiveQL [37] and the number of submitted queries was increased up to 160 and no degradation was observed in the performance. Our experiments were performed on a High Performance Cluster (HPC) machine (using 20 nodes). All nodes having 2 CPUs and each CPU with 4 processors. Each node has 16 GB of main memory and has two 146 GB disks (mirrored using RAID-1), two 24 port Gigabit ethernet switches, and one 24 port high performance switch. The software installed on the HPC are; Scientific Linux v4.5 64-bit operating system, Lustre v1.6.4.2 parallel file system, Torque v2.3.6 resource manager, Maui v3.2.6 job scheduler, Hadoop 0.20.1, and Hive 0.9.0. The configuration of Hadoop and Hive used during the experiments is given in Table 1.

TABLE 1
Default Hadoop and Hive cluster settings.

Configuration	Default value
file buffer size	4 KB
blocksize	64 MB
replication	3
namenode.handler.count	10
datanode.handler.count	10
stream-buffer-size	4 KB
maximum map tasks	2
maximum reduce tasks	2
reduce tasks	1
parallel thread number	8
merge size per task	256 MB
reducers bytes per reducer	1 GB
number of nodes	20

4.3 Running HiveQL Queries with Hadoop

TPC-H queries [Q1-Q22] are run with Hadoop settings given in Table 1. The results are presented in Figure 3. Our aim was to observe the performance of the underlying architecture with increasing database instances for each query and to obtain the execution time of the selected queries. Table 2 gives the execution times and the increase percentages for the selected/correlated queries used in the experiments.

4.4 Experiments with Increasing Number of Nodes

In Figure 4, the results of the experiments with increasing number of nodes for Q1, Q6 (with 1 GB, 25 GB, and 50 GB database instance) are presented. Increasing the number of nodes affects the performance of the queries positively. The execution times are reduced by 12, 51, 68% in the database instances respectively when nodes are increased from 5 to 20 nodes. Table 3 shows the number of map and reduce tasks generated during the

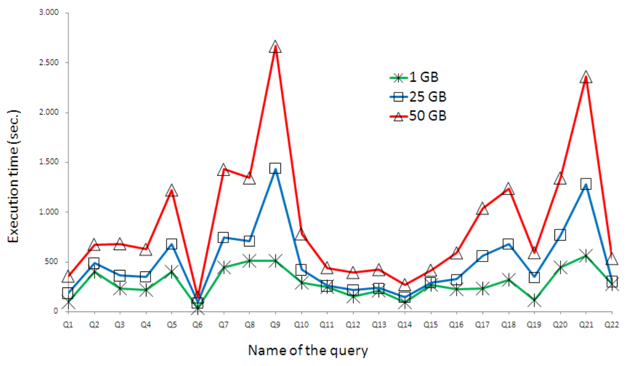

Fig. 3. TPC-H HiveQL query results with 1 GB, 25 GB, and 50 GB database instances.

TABLE 2
Execution times (sec.) of the TPC-H HiveQL queries with scales 1 GB, 25 GB, and 50 GB and their increase percentages in the execution times with respect to the previous scales.

Query	1 GB	25 GB	50 GB
Q1	96	233 (143%)	396 (70%)
Q3	233	511 (119%)	782 (53%)
Q6	34	141 (315%)	186 (32%)
Q14	91	189 (108%)	271 (43%)
Q18	316	980 (210%)	1,337 (36%)
Q19	113	532 (371%)	892 (68%)

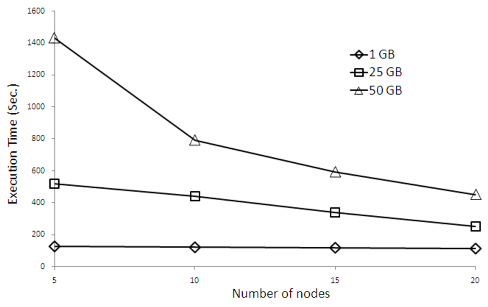

Fig. 4. The effect of increasing the number of nodes for Q1, Q6.

TABLE 3
Number of Map and Reduce Tasks with Increasing Number of Nodes.

Database Instances and Nodes	#map tasks	#reduce tasks
1 GB with 10 nodes	3	16
1 GB with 20 nodes	4	34
25 GB with 10 nodes	29	16
25 GB with 20 nodes	30	34
50 GB with 10 nodes	148	16
50 GB with 20 nodes	148	34

experiments. This experiment shows that to provide remarkable decrease in the total execution time of queries, Hive needs to add quite a number of nodes however; as it will be shown in the next experiments, SharedHive can provide the same performance with no additional nodes but exploiting the common tasks of the queries.

TABLE 4
Execution times (sec.) and improvement percentages of merged Q1 queries.

$\|Q\|$	1 GB	25 GB	50 GB
10	346 (63%)	1,549 (33%)	2,465 (37%)
20	590 (69%)	2,477 (46%)	4,148 (47%)
30	838 (70%)	3,255 (53%)	5,873 (50%)
40	1,082 (71%)	4,089 (56%)	7,519 (52%)
60	1,575 (72%)	6,148 (56%)	11,043 (53%)
80	2,066 (73%)	8,047 (56%)	14,705 (53%)

TABLE 5
Execution times (sec.) and improvement percentages of merged Q1, Q6 queries.

$\|Q\|$	1 GB	25 GB	50 GB
2	113 (13%)	251 (32%)	452 (22%)
20	403 (69%)	1,931 (48%)	3,712 (36%)
40	708 (72%)	3,524 (52%)	6,845 (41%)
60	979 (74%)	4,845 (56%)	9,287 (46%)
80	1,206 (76%)	5,988 (59%)	11,470 (50%)
120	1,848 (76%)	8,407 (62%)	16,579 (52%)
160	2,356 (77%)	11,078 (62%)	21,047 (54%)

TABLE 6
Execution times (sec.) and improvement percentages of merged Q14, Q19 queries.

$\|Q\|$	1 GB	25 GB	50 GB
2	161 (21%)	608 (15%)	979 (15%)
20	194 (90%)	852 (88%)	1,567 (86%)
40	226 (94%)	1,367 (90%)	2,249 (90%)
60	262 (95%)	1,834 (91%)	3,323 (90%)
80	298 (96%)	2,049 (92%)	4,032 (91%)
120	363 (97%)	2,870 (93%)	5,401 (92%)
160	427 (97%)	3,342 (94%)	6,390 (93%)

TABLE 7
Execution times (sec.) and improvement percentages of merged Q3, Q18 queries.

$\|Q\|$	1 GB	25 GB	50 GB
2	387 (29%)	1,313 (11%)	1,891 (10%)
20	1,176 (89%)	2,459(83%)	4,857 (17%)
40	2,062 (90%)	3,347 (88%)	5,806 (86%)
60	2,955 (91%)	4,314 (90%)	6,847 (89%)
80	3,845 (91%)	4,622 (92%)	7,831 (90%)
120	5,647 (91%)	5,406 (93%)	1,0194 (91%)
160	7,425 (91%)	6,891 (94%)	12,263 (92%)

4.5 Experiments with Increasing TPC-H Database Sizes

In this Section, correlated HiveQL queries are merged and run together to show the efficiency of SharedHive. Tables 4, 5, 6, 7 show the execution times and the achieved performance increases in the total execution times of the merged queries. Symbol $|Q|$ in these Tables denotes the number of queries given as input to SharedHive. At the first stage of the experiments, HiveQL queries are issued to Hive without optimization. Later, the same queries are merged by using SharedHive MQO component and executed again. The performance increases observed in our experiments are in the range of 20-90% depending on the amount of common tasks shared by the queries. Even with small number of merged queries, the total execution time can be reduced by as much as 50%. In case where no correlated queries can be found for improvement, the optimization time spent to analyze and merge the queries is so small that it does not have any adverse effect on the execution time of the queries.

4.6 Accuracy, Efficiency, and Scalability of SharedHive

The results of the queries found by SharedHive are always the same with the solutions of Hive and it can perform the execution of the selected/correlated queries in remarkable shorter times than conventional Hive. The execution time performance gains reaches up to 90% when the number of submitted queries is 160 correlated instances having different predicates. For repeatedly issued similar queries with different predicates, SharedHive shows excellent performance and it can handle a very large number of such queries in workloads. SharedHive benefits from HDFS therefore, its scalability performance is preserved and better performance can be obtained when additional nodes are introduced to the system.

5 Conclusion and Future Work

In this study, we proposed a multiple query optimization (MQO) framework, SharedHive, to improve the performance of conventional Hadoop Hive. To our knowledge, this is the first time that the performance of Hive is being improved with MQO techniques. In SharedHive, we detected and categorized sets of correlated TPC-H HiveQL queries and merged them into optimized HiveQL statements to run on Hadoop. With this approach, we showed that significant performance improvements can be achieved. Since SharedHive is designed as a new component on top of the existing Hive optimizer, it can be integrated into other SQL-to-MapReduce translators as well.

As future work, we are planning to apply MQO to the tasks in a single query. With this way, we intend to eliminate redundant tasks in queries and improve the overall performance of naïve rule-based query optimizer of Hive.

References

[1] Hadoop project. http://hadoop.apache.org/.
[2] Dean, J., and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.
[3] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. (2010). MapReduce online. In Proceedings of the 7th USENIX conference on Networked systems design and implementation.
[4] Stonebraker, M., Abadi, D., DeWitt, D. J., Madden, S., Paulson, E., Pavlo, A., and Rasin, A. (2010). MapReduce and parallel DBMSs: friends or foes. Communications of the ACM, 53(1), 64-71.
[5] DeWitt, D., and Stonebraker, M. (2008). MapReduce: A major step backwards. The Database Column,1.
[6] Lee, K. H., Lee, Y. J., Choi, H., Chung, Y. D., and Moon, B. (2012). Parallel data processing with MapReduce: a survey. ACM SIGMOD Record, 40(4), 11-20.
[7] Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., and Rasin, A. (2009). HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proceedings of the VLDB Endowment, 2(1), 922-933.
[8] Hive project. http://hadoop.apache.org/hive/.
[9] Ordonez, C., Song, I. Y., and 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 (pp. 67-68).
[10] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N.,..., and Murthy, R. (2010). Hive-a petabyte scale data warehouse using hadoop. ICDE,(pp. 996-1005).
[11] Sellis, T.K. (1988). Multiple-query optimization. ACM Transactions on Database Systems (TODS), 13(1), 23-52.
[12] Bayir, M. A., Toroslu, I. H., and Cosar, A. (2007). Genetic algorithm for the multiple-query optimization problem. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, 37(1), 147-153.
[13] Cosar, A., Lim, E. P., and 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 (pp. 433-438).
[14] Zhou, J., Larson, P. A., Freytag, J. C., and Lehner, W. (2007). Efficient exploitation of similar subexpressions for query processing. In Proceedings of ACM SIGMOD (pp. 533-544).
[15] Chaiken, R., Jenkins, B., Larson, P. ., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. (2008). SCOPE: easy and efficient parallel processing of massive data sets. Proceedings of the VLDB Endowment, 1(2), 1265-1276.
[16] Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J. M., and Welton, C. (2009). MAD skills: new analysis practices for big data. VLDB, 2(2), 1481-1492.
[17] He, Y., Lee, R., Huai, Y., Shao, Z., Jain, N., Zhang, X., and Xu, Z. (2011). Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. ICDE (pp. 1199-1208).
[18] Lee, R., Luo, T., Huai, Y., Wang, F., He, Y., and Zhang, X. (2011). Ysmart: Yet another sql-to-mapreduce translator. ICDCS (pp. 25-36).
[19] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., ... and Woodford, D. (2012). Spanner: Google's Globally-Distributed Database. Proceedings of OSDI.
[20] Finkelstein, S. (1982). Common expression analysis in database applications. SIGMOD (pp. 235-245).
[21] Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S. (2000). Efficient and extensible algorithms for multi query optimization. SIGMOD Record (Vol. 29, No. 2, pp. 249-260).
[22] Andrade, H., Kurc, T., Sussman, A., and Saltz, J. (2002). Multiple query optimization for data analysis applications on clusters of SMPs. In Cluster Computing and the Grid (pp. 154-154).
[23] Giannikis, G., Alonso, G., and Kossmann, D. (2012). SharedDB: killing one thousand queries with one stone. Proceedings of the VLDB Endowment, 5(6), 526-537.
[24] Mehta, M. and DeWitt, D.J. (1995). Managing intra-operator parallelism in parallel database systems. VLDB (pp. 382-394).
[25] Beynon, M., Chang, C., Catalyurek, U., Kurc, T., Sussman, A., Andrade, H., ... and Saltz, J. (2002). Processing large-scale multidimensional data in parallel and distributed environments. Parallel Computing, 28(5), 827-859.
[26] Chen, G., Wu, Y., Liu, J., Yang, G., and Zheng, W. (2011). Optimization of sub-query processing in distributed data integration systems. Journal of Network and Computer Applications, 34(4), 1035-1042.
[27] Lee, R., Zhou, M., and Liao, H. (2007). Request Window: an approach to improve throughput of RDBMS-based data integration system by utilizing data sharing across concurrent distributed queries. VLDB (pp. 1219-1230).
[28] Harizopoulos, S., Shkapenyuk, V., and Ailamaki, A. (2005). QPipe: a simultaneously pipelined relational query engine. SIGMOD (pp. 383-394).
[29] Le, W., Kementsietsidis, A., Duan, S., and Li, F. (2012). Scalable multi-query optimization for SPARQL. ICDE (pp. 666-677).
[30] Silva, Y. N., Larson, P., and Zhou, J. (2012). Exploiting Common Subexpressions for Cloud Query Processing. ICDE (pp. 1337-1348).
[31] Apache Pig. http://wiki.apache.org/pig.
[32] Bajda-Pawlikowski, K., Abadi, D. J., Silberschatz, A., and Paulson, E. (2011). Efficient processing of data warehousing queries in a split execution environment. In Proceedings of international conference on Management of data (pp. 1165-1176).
[33] Friedman, E., Pawlowski, P., and Cieslewicz, J. (2009). SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. VLDB, 2(2), 1402-1413.
[34] Nykiel, T., Potamias, M., Mishra, C., Kollios, G., and Koudas, N. (2010). Mrshare: Sharing across multiple queries in mapreduce. VLDB, 3(1-2), 494-505.
[35] Gruenheid, A., Omiecinski, E., and Mark, L. (2011). Query optimization using column statistics in hive. In Proceedings of the 15th Symposium on International Database Engineering and Applications (pp. 97-105).
[36] TPC-H. http://www.tpc.org/tpch/.
[37] Running TPC-H queries on Hive. http://issues.apache.org/jira/browse/HIVE-600.

Serkan Ozal is currently an MS Student in Middle East Technical University Ankara/TURKEY. He received his BS in University of Hacettepe Ankara TURKEY in 2005 and 2009, respectively. His main research areas are Big Data Computing, NoSQL Databases, Map/Reduce, Hadoop, Hive, Distributed/Parallel Programming.

Tansel Dokeroglu is currently a Ph.D. student in Middle East Technical University of Ankara/TURKEY. He received his BS in Mechanical Engineering Department of Turkish Armed Forces Academy and MS degree from Computer Engineering Department of Middle East Technical University in 1991 and 2006, respectively. His main research areas are Parallel Programming, Genetic Algorithms, Relational Cloud Database Query Engines. He has published papers in VLDB Ph.D. Workshop, IEEE, ISCIS international conferences, WIVACE Evolutionary computation Workshop, and industrial journal.

Murat Ali Bayir is currently a member of technical staff at Google Inc. He got his Ph.D. degree from CSE Department of the SUNY at Buffalo in 2010. He received his BS and MS degrees from Computer Engineering Department of Middle East Technical University with minor degree in Math in 2003 and 2006, respectively. His main research areas are Data Mining, Mobile Computing, Graph Theory and Social Network applications. He has published papers in international conferences, workshops and journals including WWW, WISE, WoWMoM, IEEE Transactions on SMC, Discrete Applied Mathematics, and Elsevier Mobile and Pervasive Computing. His MS Thesis titled A New Reactive Method for Processing Web Usage Data has led to an industrial research project sponsored by National Science Foundation of Turkey.

Ahmet Cosar got his BS, MS, and PhD degrees, all in computer engineering, from Middle East Technical University (METU), Bilkent University, and University of Minnesota, respectively. He has been a faculty member in METU Computer Engineering department since 1996. His research interests are in distributed databases, data mining, e-commerce, and web-based software architectures. Dr. Cosar has also worked as a visiting faculty member in University of Sharjah (UAE) and Manas University (Kyrgyzstan) and also taught a course at American University of Central Asia.