YSmart: Yet Another SQL-to-MapReduce Translator

Rubao Lee [#1], Tian Luo [#2], Yin Huai [#3], Fusheng Wang [$4], Yongqiang He [*5], Xiaodong Zhang [#6]

[#] *Department of Computer Science and Engineering, The Ohio State University*
{[1]liru, [2]luot, [3]huai, [6]zhang}@cse.ohio-state.edu

[$] *Center for Comprehensive Informatics, Emory University*
[4]Fusheng.Wang@emory.edu

[*] *Facebook Data Infrastructure Team*
[5]heyongqiang@fb.com

***Abstract*— MapReduce has become an effective approach to big data analytics in large cluster systems, where SQL-like queries play important roles to interface between users and systems. However, based on our Facebook daily operation results, certain types of queries are executed at an unacceptable low speed by Hive (a production SQL-to-MapReduce translator). In this paper, we demonstrate that existing SQL-to-MapReduce translators that operate in a one-operation-to-one-job mode and do not consider query correlations cannot generate high-performance MapReduce programs for certain queries, due to the mismatch between complex SQL structures and simple MapReduce framework. We propose and develop a system called *YSmart*, a correlation aware SQL-to-MapReduce translator. YSmart applies a set of rules to use the minimal number of MapReduce jobs to execute multiple correlated operations in a complex query. YSmart can significantly reduce redundant computations, I/O operations and network transfers compared to existing translators. We have implemented YSmart with intensive evaluation for complex queries on two Amazon EC2 clusters and one Facebook production cluster. The results show that YSmart can outperform Hive and Pig, two widely used SQL-to-MapReduce translators, by more than four times for query execution.**

I. Introduction

Large online stores and Web service providers must timely process an increasingly large amount of data represented by Web click-streams, user-generated contents, online transaction data, and others. To understand user behaviors and acquire useful information hidden in these huge data sets, extensive data processing applications are needed, such as Web-scale data mining, content pattern analysis (e.g. [1]), click-stream sessionization (e.g. [2]), and others. With the rapid advancement of network technologies, and the increasingly wide availability of low-cost and high-performance commodity computers and storage systems, large-scale distributed cluster systems can be conventionally and quickly built to support such applications [3]. MapReduce [4] is a distributed computing programming framework with unique merits of automatic job parallelism and fault-tolerance, which provides an effective solution to the data analysis challenge. As an open-source implementation of MapReduce, Hadoop has been widely used in practice.

The MapReduce framework requires that users implement their applications by coding their own *map* and *reduce* functions. Although this low-level hand coding offers a high flexibility in programming applications, it increases the difficulty in program debugging [5]. High-level declarative languages can greatly simplify the effort on developing applications in MapReduce without hand-coding programs [6]. Recently, several SQL-like declarative languages and their translators have been built and integrated with MapReduce to support these languages. These systems include Pig Latin/Pig [7], [8], SCOPE [9], and HiveQL/Hive [10]. In practice, these languages play a more important role than hand-coded programs. For example, more than 95% Hadoop jobs in Facebook are not hand-coded but generated by Hive.

These languages and translators have significantly improved the productivity of writing MapReduce programs. However, in practice, we have observed that auto-generated MapReduce programs for many queries are often extremely inefficient compared to hand-optimized programs by experienced programmers. Such inefficient SQL-to-MapReduce translations bring two critical problems in the Facebook production environment. First, auto-generated MapReduce jobs cause some queries to have unacceptably long execution times in some critical Facebook operations in the production environment. Second, for a large production cluster, the programs generated from inefficient SQL-to-MapReduce translations would create many unnecessary jobs, which is a serious waste of cluster resources. This motivates us to look into the bottlenecks in existing translators such as Hive, and develop more efficient SQL-to-MapReduce translator to generate highly optimized MapReduce programs for complex SQL queries.

The Performance Gap

To demonstrate the problem, we compared the performance between Hive-generated program and hand-coded MapReduce program for a click-stream query that represents a typical Facebook production workload. This query (Q-CSA) is used to answer *"what is the average number of pages a user visits between a page in category X and a page in category Y?"* based on a single click-stream table *CLICKS(user_id int, page_id int, category_id int, ts timestamp)*. It is a complex query that needs self-joins and multiple aggregations of the same table. Its SQL statement is shown in Fig. 1[1], and its execution plan tree is shown in Fig. 2(a). To demonstrate the

[1]This query is modified based on the SQL statement presented in paper [2] (page 1411) by replacing the non-SQL-standard "DISTINCT ON" clause with standard grouping and aggregation clauses. The semantics of the query is still the same.

performance gap, we also used a simple query (Q-AGG) that counts the number of clicks for each category. It only executes an aggregation with one pass of table scan on *CLICKS*.

```
SELECT avg(pageview_count) FROM
(SELECT c.uid,mp.ts1,(count(*)-2) AS pageview_count
 FROM clicks AS c,
   (SELECT uid,max(ts1) AS ts1,ts2
    FROM (SELECT c1.uid,c1.ts AS ts1,min(c2.ts) AS ts2
          FROM clicks AS c1,clicks AS c2
          WHERE c1.uid = c2.uid AND c1.ts < c2.ts
                AND c1.cid = X AND c2.cid = Y
          GROUP BY c1.uid,ts1) AS cp
   GROUP BY uid,ts2) AS mp
 WHERE c.uid=mp.uid AND c.ts>=mp.ts1 AND c.ts<=mp.ts2
 GROUP BY c.uid,mp.ts1) AS pageview_counts;
```

Fig. 1. The SQL statement for the clickstream analysis query (Q-CSA).

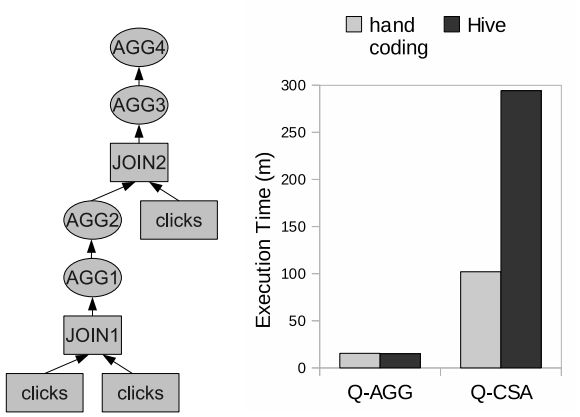
Fig. 2(b) shows the experiment results. For the simple Q-AGG query, Hive has comparable performance with our hand-coded program[2]. However, for query Q-CSA, the hand-coded MapReduce program outperforms Hive by almost three times. In fact, Hive generates a chain of MapReduce jobs according to the query plan, and each job is independently responsible for executing one operation in the plan tree. However, our hand-coded program, on the basis of query semantic analysis, uses only a single job to execute all the operations except the final aggregation (AGG4). This significantly reduces redundant computations and I/O operations in the MapReduce execution.

Translating SQL to MapReduce: Where Is the Bottleneck?

The above example shows that the source of inefficiency comes from the naive approach for translating SQL queries into MapReduce jobs. SQL-like declarative languages for MapReduce, such as Hive, use a subset of SQL language constructs. In practice, when translating a query expressed by such a language into MapReduce programs, existing translators take a one-operation-to-one-job approach. For a query plan tree, each operation in the tree is replaced by a pre-implemented MapReduce program, and the tree is finally translated into a chain of programs. For example, Hive generates six jobs to execute the six operations (JOIN1, AGG1, AGG2, JOIN2, AGG3, and AGG4) in the plan tree shown in Fig. 2(a). Such a translation approach is inefficient since it can cause redundant table scans (e.g., both JOIN1 and JOIN2 need to scan *CLICKS*) and unnecessary data transfers among multiple jobs. Thus, existing translators cannot generate high-performance MapReduce programs for two reasons. First, they cannot address the limitations of the simple MapReduce structure for a complex query. Second, they cannot utilize the unique opportunities provided by intra-query correlations in a complex query. We further give more specific explanations as follows.

a) Limitations of MapReduce for Complex Queries: A one-operation-to-one-job translation does not fully utilize MapReduce's flexible programming capabilities, instead, it is constrained by the structure and implementation of MapReduce in two ways. First, MapReduce requires materialization of intermediate results on local disks in order to deal with

[2]This is because Hive uses an optimized execution strategy for aggregations by maintaining an internal hash-aggregate map in the map phase of a job [11].



(a) Q-CSA plan. (b) Execution times.

Fig. 2. Comparison between Hive and hand-coded MapReduce programs.

node failures. Furthermore, temporary result of each step in a job chain must be uploaded to the global file system. This could cause extra overhead of disk I/O and network transfers. Second, the run-time system (e.g., Hadoop) is not aware whether concurrent jobs are correlated, thus it does not provide any mechanism to support intermediate data reusing between concurrent jobs. Due to the two limitations, MapReduce programs automatically translated in a one-operation-to-one-job approach may have low performance.

Indeed, an experienced programmer with the knowledge of database query engine can write efficient MapReduce programs, although not preferable, to execute a complex query, by analyzing and considering the intra-query correlations.

b) Intra-query Correlations: One typical type of complex queries in MapReduce is queries on multiple occurrences of the same table, including self-joins. Such queries are common in various data analysis applications. In traditional decision Support System (DSS) workloads characterized by TPC-H and TPC-DS, many queries are performed on multiple occurrences of the same table [12]. It is also very common to find such queries in spatial database systems [13] and other applications [14]. For Web data analysis, a query (e.g. Q-CSA) can have several times of self-join of the only table for storing click-stream [2]. More importantly, such type of queries are typical MapReduce workloads in Web-scale systems.

By considering intra-query correlations, SQL-to-MapReduce translations and executions can be automatically optimized to significantly improve performance through minimizing computation and I/O operations by merging correlated query operations. For example, in Q-CSA (in Fig. 2(a)), instead of three table scans of the same *CLICKS* table, JOIN1 only needs a single table scan for two instances of the same table, and AGG1, AGG2, JOIN2, AGG3 can be directly executed in the same job for JOIN1 without the need of additional jobs. Therefore, a single table scan of table *CLICKS* can support all the three instances in JOIN1 and JOIN2, and a single MapReduce job can execute all the five

operations from JOIN1 to AGG3 in the query execution plan.

Our Contribution: YSmart

Our goal is to build a correlation-aware SQL-to-MapReduce translator to optimize complex queries without modification to the MapReduce framework and the underlying system. YSmart is built on top of Hadoop as it is a widely used system and also used by Facebook. YSmart supports three types of intra-query correlations defined based on the key/value pair model of the MapReduce framework. After automatically detecting such correlations in a query, YSmart applies a set of rules to generate optimized MapReduce jobs, which are managed by the *Common MapReduce Framework* (CMF) in YSmart, so that it can use the minimal number of jobs to execute multiple correlated operations in the query. This provides significant query performance improvement by reducing redundant computations, unnecessary disk accesses, and network overhead.

We have conducted intensive experiments with both DSS workloads and click-stream analysis workloads on different scales of clusters: a small local cluster, two Amazon EC2 clusters, and a large production cluster in Facebook. The results show significant advantages of YSmart in terms of both performance and scalability over existing translators even with diverse configurations and unpredictable run-time dynamics.

The rest of this paper is organized as follows. Section II briefly introduces background knowledge; Section IV presents the definitions of intra-query correlations and their usages in YSmart; In Section V, we present how MapReduce jobs are generated in YSmart; The Common MapReduce Framework (CMF) is discussed in Section VI; Performance evaluation is presented in Section VII; Section VIII discusses related work, and Section IX concludes this paper.

II. Background

A. MapReduce and Hadoop

In the MapReduce framework, a computation is represented by a MapReduce job. A job has two phases: the map function phase and the reduce function phase. The underlying run-time system executes the functions in a way that it automatically partitions the output of the map function and copies it to the input of the reduce function. Furthermore, a complex computation process can be represented by a chain of jobs.

MapReduce does not allow arbitrary interfaces of the map and reduce function. Rather, their input and output must be based on key/value pairs. A map function accepts a key/value pair (k_1, v_1) and emits another key/value pair (k_2, v_2). After the map phase, the run-time system collects a list of values for each distinct key in the map output. Then, for each k_2, a reduce function accepts the input of (k_2, a list of (v_2)), and emits (k_3, a list of (v_3)). MapReduce allows users to define the format of a key/value pair. It can be a simple scalar value (e.g, an integer value or a string) or a complex composite object. In this way, it provides high flexibility to express computations and data processing operations in MapReduce jobs.

Hadoop is an open-source implementation to MapReduce designed for clusters of many nodes. It provides a *Hadoop Distributed File System (HDFS)* as the global file system running on a cluster. The execution of a MapReduce job in Hadoop has three steps. First, the JobTracker assigns a portion (e.g. a 64MB data chunk) of an input file on HDFS to a map task running on a node. Then the TaskTracker on the node extracts key/value pairs in the chunk, and invokes the map function for each pair. The map output, namely the intermediate result, is sorted and stored in local disks. Second, all the intermediate results on all nodes are transmitted into inputs of reduce functions. This step fetches the results via HTTP requests, partitions and groups the results according to their keys, and stores each partition to a node for reduce. Finally, each reduce function reads its input from its local disks, and outputs its result to HDFS via network.

B. Relational Operations in MapReduce

In order to evaluate an SQL query in MapReduce, the query must be represented into a single or a chain of MapReduce jobs. The critical issue is that each operation (e.g. selection, aggregation, join) must be implemented into a transformation between input key/value pairs and output key/value pairs. It is straightforward to implement selection and projection. For aggregation with grouping, the columns for grouping can be the keys for data partitioning in the map phase, and the aggregation is finished in the reduce phase. For join between two data sets, an efficient way is that each data set is partitioned by its columns for join condition, and the join is finished in the reduce phase. In this way, each key/value pair produced by a map function should have a tag to indicate the source of the pair so that the following reduce can know where an input pair comes from [11][15][16].

III. Correlation-aware MapReduce: An Overview

As we have introduced in the previous Section, a MapReduce job can efficiently execute a relational operation. However, using a chain of jobs to execute a complex SQL query with multiple operations could be inefficient, if the SQL-to-MapReduce translator does not consider possible intra-query correlations and works in a *one-operation-to-one-job* mode used by DBMSs. In a DBMS, when converting a logical query plan tree to the final physical plan, each logical operation is replaced with one pre-implemented physical operator [17]. For example, a join operation can be represented by a hash join operator. Eventually, in the physical plan, multiple physical operators are linked in an executable binary. We call the way of using one-to-one mapping from logical operations to physical operators as a one-operation-to-one-job translation mode.

However, the outcome can be very different if a SQL-to-MapReduce translator takes the same approach, because MapReduce does not have the same execution environment as that in a DBMS. A DBMS exploits a pipelined and iterator-based interconnection among multiple operators [18] that are in the same memory space. As the overhead of operator communications is very low, the physical plan can be executed efficiently. However, in a MapReduce environment, if each operator is represented by a MapReduce job, the efficiency of

the physical plan (a chain of jobs) can be low. MapReduce, with the merit of fault-tolerance in large-scale clusters, requires that intermediate map outputs be persistent on disks and reduce outputs be written to HDFS over the network. Under such a materialization policy, the way of executing multiple operations in a single job (many-to-one), if possible, could be a much more effective choice than the one-to-one translation.

YSmart is designed for translating a SQL query into MapReduce programs with specific considerations of intra-query correlations. YSmart batch-processes multiple correlated query operations within a query thus significantly reduces unnecessary computations, disk I/Os and network transfers. During job generation, YSmart applies a set of optimization rules to merge multiple jobs, which otherwise would have been run independently without YSmart, into a common job. It provides a Common MapReduce Framework (CMF) that allows multiple types of jobs, e.g., a join job and an aggregation job, to be executed in a common job. The CMF has low overhead on managing multiple merged jobs.

To achieve its goals, YSmart must address the following three issues (in the next three sections, respectively):

1) What types of correlations exist in a query and how can they affect query execution performance?
2) With the awareness of correlations, how to translate a query plan tree into efficient MapReduce programs?
3) How to design and implement the Common MapReduce Framework that need to merge different types of jobs with low overhead?

IV. Intra-query Correlations and their Optimization Principles

In this paper, we target SQL queries with following operations: selection, projection, aggregation (with or without grouping), sorting, and equi-join (inner join or left/right/full outer join). These operations are the most common and important for relational queries. We define intra-query correlations as possible relationships between join nodes or aggregation nodes, or both, in a query plan tree.

A. Types of Correlations and the Optimization Benefits

For an operation node in a query plan tree, YSmart introduces a property *Partition Key (PK)* to reflect how map output is partitioned in the operation execution with MapReduce's key/value pair model. Since a map function is to transform (k_1, v_1) to (k_2, v_2), the partition key actually represents k_2. The partition key of an equi-join is the set of columns used in the join condition. The partition key of an aggregation can be any non-empty subset from the set of columns used in grouping. For example, for a join operation $R(A, B) \bowtie S(A, C)$, the partition key is (A). For an aggregation operation on R with grouping attributes G_1 and G_2, the partition key can be (G_1), (G_2), or (G_1, G_2).

In a query plan tree, we define three correlations:

1) *Input Correlation*: Multiple nodes have input correlation (IC) if their input relation sets are not disjoint;
2) *Transit Correlation*: Multiple nodes have transit correlation (TC) if they have not only input correlation, but also the same partition key;
3) *Job Flow Correlation*: A node has job flow correlation (JFC) with one of its child nodes if it has the same partition key as that child node.

These definitions do not cover the correlation within a self-join of the same table, since such a correlation does not help reduce the number of jobs. We develop a special optimization for self-join as discussed in Section V-A.

If an aggregation node has multiple partition key candidates, YSmart has to determine which one is its partition key. Currently YSmart does not seek a solution based on execution cost estimations due to the lack of statistics information of data sets. Rather, YSmart uses a simple heuristic by selecting the one that can connect the maximal number of nodes that can have these correlations.

These correlations between nodes provide an opportunity so that the jobs for the nodes can be batch-processed to improve efficiency. First, if two nodes have input correlation, then the corresponding two jobs can share the same table scan during the map phase. This can either save disk reads if the map is local or save network transfers if the map is remote. Second, if two nodes have transit correlation, then there exists overlapped data between map outputs of the jobs. Thus, during a map-to-reduce transition, redundant disk I/O and network transfers can be avoided. Finally, if a node has a job flow correlation with one of its child nodes, then it is possible that the node actually can be directly evaluated in the reduce phase of the job for the child node. Specifically, in this case of exploiting job flow correlation, there are following scenarios:

1) An aggregation node with grouping can be directly executed in the reduce function of its only child node;
2) A join node J_1 has job flow correlation with only one of its child nodes C_1. Thus as long as the job of another child node of this join node C_2 has been completed, a single job is sufficient to execute both C_1 and J_1;
3) A join node J_1 has job flow correlation with two child nodes C_1 and C_2. Then, according to the correlation definitions, C_1 and C_2 must have both input correlation and transit correlation. Thus a single job is sufficient to execute both C_1 and C_2. Besides, J_1 can also be directly executed in the reduce phase of the job.

B. An Example of Correlation Query and Its Optimization

We take the query shown in Fig. 3 as an example to demonstrate the three types of correlations and their optimization benefits. The query is re-written from the original TPC-H Q17 (more details covered in Section VII.) As we can see from the query plan tree (Fig. 4), an aggregation node (AGG1) generates *inner*, a join node (JOIN1) generates *outer*, and a join node (JOIN2) joins *inner* and *outer*.

To illustrate correlations and their benefits, we show the generated MapReduce jobs without and with the awareness of correlations respectively. Without the awareness of correlations, a one-to-one translation will generate three MapReduce

```
SELECT sum(l_extendedprice) / 7.0 AS avg_yearly
FROM (SELECT l_partkey, 0.2* avg(l_quantity) AS t1
      FROM   lineitem
      GROUP BY l_partkey) AS inner,
     (SELECT l_partkey,l_quantity,l_extendedprice
      FROM   lineitem, part
      WHERE  p_partkey = l_partkey) AS outer
WHERE  outer.l_partkey = inner.l_partkey;
  AND  outer.l_quantity < inner.t1;
```

Fig. 3. A variation of TPC-H Q17.

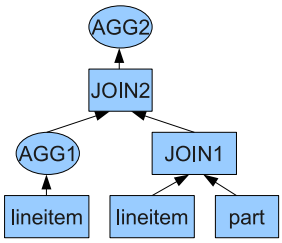


Fig. 4. The query plan tree for Q17.

jobs for the three nodes through a post-order tree traverse. Fig. 5 shows the three jobs: Job1 for AGG1, Job2 for JOIN1, and Job3 for JOIN2. In each job, the map function transforms an input record to a key/value pair. For example, Job1's map function transforms a $lineitem$ record to a key/value pair that uses column $l\_partkey$ as the key and column $l\_quantity$ as the value. The reduce function is the actual worker for aggregation or join. For example, Job1's reduce function executes aggregation on $l\_quantity$ for each unique input key ($l\_partkey$).

```
Job1: generate inner by group/agg on lineitem
Map:
 lineitem -> (k:l_partkey, v:l_quantity)
Reduce:
 calculate (0.2*avg(l_quantity)) for each (l_partkey)

Job2: generate outer by join lineitem and part
Map:
 lineitem -> (k: l_partkey,
              v:(l_quantity,l_extendedprice))
 part -> (k:p_partkey,v:null)
Reduce:
 join with the same partition (l_partkey=p_partkey)

Job3: join outer and inner
Map:
 outer-> (k:l_partkey, v:(l_quantity,l_extendedprice))
 inner-> (k:l_partkey, v:(0.2*avg(l_quantity)))
Reduce:
 join with the same partition of l_partkey
```

Fig. 5. A chain of jobs for the plan in Fig. 4. (We ignore the fourth job for evaluating the final aggregation AGG2)

We can determine the correlations among the nodes by looking into their corresponding MapReduce jobs. First, both AGG1 and JOIN1 need the input of the *lineitem* table, which means these two nodes have input correlation. Second, AGG1 and JOIN1 have the same partition key $l\_partkey$. This fact can be reflected by the map output key/value pairs in Job1 and Job2. Both jobs use $l\_partkey$ to partition their input table *lineitem*[3]. Based on correlation definitions, AGG1 and JOIN1 have transit correlation. Finally, as the parent node of AGG1 and JOIN1, JOIN2 has the same partition key $l\_partkey$ as all its child nodes. As shown in the map phase of Job3, $l\_partkey$ is used to partition *outer* and *inner*, thus JOIN2 has job flow correlation with both AGG1 and JOIN1.

By exploiting these correlations, instead of generating three independent jobs, YSmart only needs to use a single MapReduce job to execute all functionalities of AGG1, JOIN1, and JOIN2, as shown in Fig. 6. Such job merging has two advantages. First, by exploiting input correlation and transit correlation, AGG1 and JOIN1 can share a single scan of the *lineitem* table, and remove redundant map outputs. Second, JOIN2 can be directly executed in the reduce phase of the job. Therefore, the persistence and re-partitioning of intermediate tables *inner* and *outer* are actually avoided, which can significantly boost the performance of the query.

```
Job1: generate both inner and outer,
      and then join them
Map:
 lineitem -> (k: l_partkey,
              v:(l_quantity,l_extendedprice))
 part -> (k:p_partkey,v:null)
Reduce:
 get inner: aggregate l_quantity for each (l_partkey)
 get outer: join with (l_partkey=p_partkey)
 join inner and outer
```

Fig. 6. The optimized job by exploiting correlations.

Thus, the major task of YSmart is to translate a SQL query into efficient MapReduce jobs with the awareness of intra-query correlations. Next, we will discuss how YSmart translates such complex queries as jobs in Section V and then present the Common MapReduce Framework for executing merged jobs and generating final results in Section VI.

V. Job Generation in YSmart

The initial task of YSmart is to translate a SQL query into MapReduce jobs. We first present the primitive job types in YSmart, and then introduce how to merge these jobs.

A. Primitive Job Types

Based on the programming flexibility of MapReduce, YSmart provides four types of MapReduce jobs for different operations.

- A SELECTION-PROJECTION (SP) Job is used to execute a simple query with only selection and projection operations on a base relation;
- An AGGREGATION (AGG) job is used to execute aggregation and grouping on an input relation;
- A JOIN job is used to execute an equi-join (inner or left/right/full outer) of two input relations;
- A SORT job is used to execute a sorting operation.

If selection and projection operations come with a job on a physical table, these operations are executed by the job itself, but not executed by an individual job. For a JOIN job, in addition to the equi-join condition, other predicates, for example an "*IS NULL*" predicate after an outer join, are executed by the job itself without the need of additional jobs.

[3]Job2 uses $p\_partkey$ to partition the *part* table. The columns in the two sides of the equi-join predicate $l\_partkey = p\_partkey$ are just aliases of the same partition key.

A JOIN job for a self-join of the same table is optimized to use only a single table scan in the map phase. For each raw record, according to the select conditions of the two instances of the table, the mapper adds a tag in the output key/value pair to indicate which instance (or both) the pair belongs to.

With these primitive jobs, it is possible to provide a one-operator-to-one-job based translation from a query plan tree to MapReduce programs. By traversing a tree with post-order and replacing a node with its corresponding type of the job, a chain of MapReduce jobs can be generated with data dependence. YSmart, beyond this straightforward translation, is able to optimize jobs via job merging.

B. Job Merging

With the awareness of the three intra-query correlations, YSmart provides a set of rules to merge multiple jobs into a common job. The merging of jobs can either be at the map phase or at the reduce phase, performed in two different steps – the first step applies for input correlation and transit correlation, and the second step applies for job flow correlation.

Rule 1: If two jobs have input correlation and transit correlation, they will be merged into a common job. This is performed in the first step, where YSmart scans the chain of jobs generated from the above one-to-one translation. This process continues until there is no more input correlation and transit correlation between any jobs in the chain. After this step, YSmart will continue the second step to detect if there are jobs that can be merged in the reduce phase of a prior job.

Rule 2: An AGGREGATION job that has job flow correlation with its only preceding job will be merged into this preceding job.

Rule 3: For a JOIN job with job flow correlation with its two preceding jobs, the join operation will be merged into the reduce phase of the common job. In this case, there must be transit correlation between the two preceding jobs, and the two jobs have been merged into a common job in the first step. Based on this, the join operation can be put into the reduce phase of the common job.

Rule 4: For a JOIN job that has job flow correlation with only one of its two preceding jobs, merge the JOIN job with the preceding job with job flow correlation – which has to be executed later than the other one. For example, a JOIN job J_1 has job flow correlation with P_1 but not P_2. In this case, J_1 can be merged into P_1 only when P_2 was finished before P_1. In this case, YSmart needs to determine the sequence of executing two preceding jobs for a JOIN job. That is, the preceding job that has no job flow correlation with the JOIN job must be executed first. YSmart implements this rule when traversing the query plan tree with post-order. For a join node, its left child and right child can be exchanged in this case.

C. An Example of Job Merging

We take the query plans shown in Fig. 7 as an example to demonstrate the job merging process. The difference between the two plans is that the left child and right child of node JOIN2 are exchanged. We assume that 1) JOIN1 and AGG2 have input correlation and transit correlation, 2) JOIN2 has job flow correlation with JOIN1 but not AGG1, and 3) JOIN3 has job flow correlation with both JOIN2 and AGG2. In the figure, we show the job number for each node.

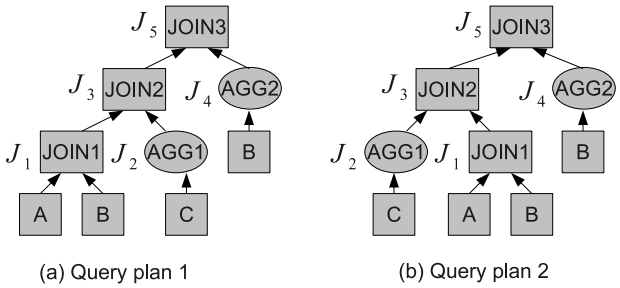


Fig. 7. Two query plan trees.

For the plan in Fig. 7 (a), a post-order traverse will generate five jobs in a sequence $\{J_1, J_2, J_3, J_4, J_5\}$. In the first step to use input correlation and transit correlation, J_1 and J_4 will be merged. Thus, the job sequence becomes $\{J_{1+4}, J_2, J_3, J_5\}$. In the second step to use job flow correlation, J_5 will be merged into J_3 since when J_3 begins J_4 has already finished in the merged job J_{1+4}. Thus, finally we get three jobs in a sequence $\{J_{1+4}, J_2, J_{3+5}\}$. However, since YSmart uses *Rule 4* to exchange J_1 and J_2, the plan can be automatically transformed to the plan in Fig. 7 (b).

For the plan in Fig. 7 (b), since J_2 is finished before J_1, the plan can be further optimized by maximally using job flow correlation. The initial job sequence is $\{J_2, J_1, J_3, J_4, J_5\}$. After the first step that merges J_1 and J_4, the sequence is $\{J_2, J_{1+4}, J_3, J_5\}$. At the second step, since J_2 has finished, J_3 can be directly executed in the job J_{1+4}. Furthermore, J_5 can also be merged into the job. Therefore, the final job sequence is $\{J_2, J_{1+4+3+5}\}$ with only two jobs.

VI. The Common MapReduce Framework

The Common MapReduce Framework (CMF) is the foundation of YSmart to use a common job to execute functionalities of multiple correlated jobs. CMF addresses two major requirements in optimizing and running translated jobs.

The first requirement is to provide a flexible framework to allow different types of MapReduce jobs, for example a JOIN job and an AGGREGATION job, to be plugged into a common job. Therefore, the map and reduce function of a common job must have the ability to execute multiple different codes belonging to independent jobs.

The second requirement is to execute multiple merged jobs in a common job with minimal overhead. Since a common job needs to manage all computations and input/output of its merged jobs, the common job needs to bookkeep necessary information to keep track of every piece of data and their corresponding jobs, and provides efficient data dispatching for merged jobs. Due to the intermediate materialization limitation of MapReduce, any additional information generated by the common job will be written to local disks and transferred over the network. Thus, CMF needs to minimize the bookkeeping information to minimize the overhead.

CMF provides a general template based approach to generate a common job that can merge a collection of correlated jobs. The template has the following structures. The *common mapper* executes operations (selection and/or projection operations) involved in the map functions of merged jobs. The *common reducer* executes all the operations (e.g. join or aggregation) involved in the reduce functions of merged jobs. The *post-job computation* is a subcomponent in the common reducer to execute further computations on the outputs of merged jobs.

A. Common Mapper

A common map function accepts a line (a record) in the raw data file as an input. Then it emits a common key/value pair that would contain all the required data for all the merged jobs. (The pair could be $null$ if nothing is selected.)

Since different merged jobs can have different projected columns, and different jobs can have different selection conditions, the common mapper needs to record which part should be dispatched to which query in the reduce phase. Such additional bookkeeping information can bring overhead caused by intermediate result materialization in MapReduce. To minimize the overhead, CMF takes the following approaches. First, the projection information is kept as a job-level configuration property since this information is fixed and record-independent for each job. Second, for each value in the output key/value pair, CMF adds a tag about which job should use this pair in the reduce phase. Since each tag is record-dependent, their aggregated size cannot be ignored if a large number of pairs are emitted by the common mapper. Therefore, in our implementation, a tag only records the IDs of jobs (if they exist) that should not see this pair in their reduce phases. This could support common cases with highly overlapped map outputs among jobs.

B. Common Reducer and Post-job Computations

A common reduce function does not limit what a merged reducer (i.e., the reduce function of a merged job) can do. The core task of the common reducer is to iterate the input list of values, and dispatch each value with projections into the corresponding reducers that need the value. CMF requires a merged reducer be implemented with three interfaces: (1) an *init* function, (2) a *next* function driven by each value, and (3) a *final* function that does computations for all received values. This approach has two advantages: It is general and allows any types of reducers to be merged in the common reducer; It is efficient since it only needs one pass of iterations on the list of values. The common reducer outputs each result of a merged reducer to the HDFS, and an additional tag is used for each output key/value pair to distinguish its source.

However, in the common reduce function, if another job (say J_a) has job flow correlation to these merged jobs, it can be instantly executed by a post-job computation step in the function, so that J_a would not be initiated as an independent MapReduce job. In this case, the results of the merged jobs would not be outputted, but are treated as temporary results and consumed by J_a. Thus, the common reducer only outputs the results of J_a. (See Algorithm 1 for the workflow).

Algorithm 1: the Common Reduce Function

```
input: key, a list of values
foreach merged Reducer R do R.init(key);
while there are left values do
    cur_val = get_current_value();
    foreach merged Reducer R do
        if R can see cur_val (according to the tag) then
            do projection on cur_val and get p_cur_val
            R.next(key, p_cur_val);
foreach merged Reducer R do R.final(key);
if there are no post-job computations then
    foreach merged Reducer R do output R.get_result();
else
    execute post-job computations;
    output final result;
```

VII. Evaluation

To demonstrate the performance and scalability of YSmart, we provide comprehensive study of YSmart versus the most recent version of Hive [10] and Pig [8], two widely-used translators from SQL-like queries to MapReduce programs.

A. Workloads and Analysis

1) Workloads: We used two types of workloads. The first workload consists of Q17, Q18, and Q21 from the TPC-H benchmark which has been widely used in performance evaluation for complex DSS workloads. The original queries have nested sub-queries. Since the MapReduce structure does not support iterative jobs and nested parallelism [19], these queries have to be "flattened" so that they can be expressed by MapReduce programs. In our work, we took the first-aggregation-then-join algorithm [20] to flatten the three queries. The second workload comes from a Web click-stream-analysis workload. The query Q-CSA has been introduced in the Introduction Section.

The codes for running three TPC-H queries on Hive can be found in an open report[4]. For YSmart, we modified the Hive queries (they are flattened by first-aggregation-then-join) to standard SQL statements. For Pig, we tried our best to write highly efficient queries according to available features of the Pig Latin language [7]. For example, we used multi-way join and the SPLIT operator whenever possible. Fig. 8 shows query plans of Q18 and Q21 (Q17 in Fig. 4, Q-CSA in Fig. 2(a)).

2) Analysis of query execution: Next we explain how the four queries are executed in YSmart. The three TPC-H queries have similar situations. First, as the analysis in Section IV-B, for Q17 (Fig. 4), YSmart can generate one MapReduce job to

[4]http://issues.apache.org/jira/secure/attachment/12416257/TPC-H_on_Hive_2009-08-11.pdf

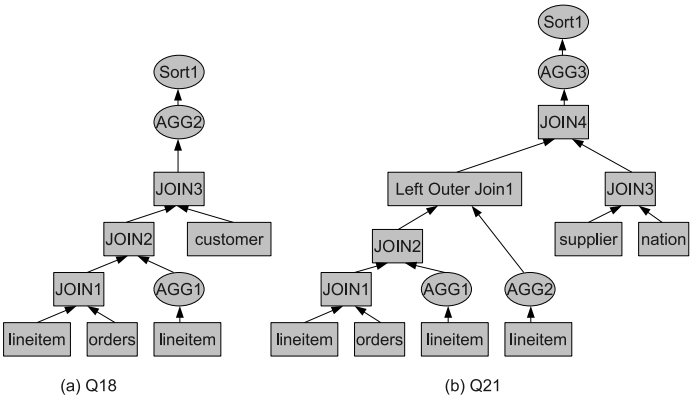


Fig. 8. Query plan trees for Q18 and Q21.

execute all the operations in the sub-tree of JOIN2. Second, for Q18 (Fig. 8(a)), JOIN1, AGG1, and JOIN2, which have the same PK (*l_orderkey*), can be executed by a single job. In the job, the map phase is used to partition the input tables *lineitem* and *orders*, and the reduce phase is used to execute the three operations. Third, similar to Q18, for Q21 (Fig. 8(b)), all the five operations in the sub-tree of "Left Outer Join1" have the same PK (*l_orderkey*), and can be executed by a single job. Real SQL code for this sub-tree is included in Appendix.

The execution of Q-CSA (Fig. 2(a)) is similar to the three TPC-H queries. YSmart can generate one job to execute all the operations in the sub-tree of AGG3. There are a special situation for this query. As aggregation nodes, both AGG1 and AGG2 have multiple candidate PKs since their Group-By clauses have more than one column. For example, the PK of AGG1 (i.e. *group by uid, ts1*) can be (*uid*), (*ts1*), or (*uid, ts1*). YSmart determines *uid* as the PK so that AGG1 can have job flow correlation with JOIN1 since JOIN1's PK is *uid*. The same choice is for AGG2. Finally, YSmart determines that all the five operations (JOIN1, AGG1, AGG2, JOIN2, and AGG3) have correlations so that they can be executed by one job.

After having optimized the above sub-trees in the whole plan trees, YSmart cannot provide any further optimizations for the rest operations that have no available correlations. They will be executed in consequent jobs which are generated in the same way as a one-operator-to-one-job translation. Note that as shown by the following experimental results, these consequent jobs are lightly-weighted. Thus YSmart's effort is the most critical for improving performance of the whole query.

B. Experimental Settings

We have conducted comprehensive evaluation on three types of clusters:

1. A small-scale cluster with only two nodes connected by a Gigabit Ethernet. Each node comes with a quad-core Intel Xeon X3220 processor (2.4 GHz), 4GB of RAM, a 500GB hard disk, running Fedora Linux 11. One node is used to run JobTracker, and another node is used to run TaskTracker. The TaskTracker is configured to provide 4 task slots. The Hadoop version is 0.19.2 (map output compression is disabled.)

2. Two middle-scale clusters provided by Amazon EC2 commercial cloud service. These two clusters have 11 nodes and 101 nodes, respectively. Each node is a default small instance comes with 1.7 GB of memory, 1 EC2 Compute Unit (1 virtual core), 160 GB of local instance storage, 32-bit platform[5]. One node is selected from each cluster for JobTracker. We use the Cloudera Distribution AMI for Hadoop[6]. It provides scripts to automatically configure Hadoop, Hive, and Pig. We use its default configuration for our experiments.

3. A large-scale production cluster in Facebook. In this cluster, 747 nodes are assigned to perform our experiments. Each node has 8 cores, 32GB memory, and 12 disks of 1TB. The used Hadoop version is Hadoop 0.20.

C. On Small-scale Cluster: YSmart vs Hand-coded Program

This small execution environment allows us to make detailed measurement in an isolated mode. We used a 10GB TPC-H data set for TPC-H queries, and a large 20GB data set for Q-CSA. In this subsection, we compare performance of YSmart and hand-coded MapReduce program for the most complex query (Q21). Then, we compare performance between YSmart, Hive, and Pig for all the four queries in the next subsection.

We made detailed tests to compare YSmart and hand-coded programs for Q21. We only tested the execution of the sub-tree "Left Outer Join 1" (see Fig. 8 (b)), since it is the dominated part for the whole query execution of Q21.

In order to understand how each type of correlations can be beneficial to query execution performance, we test the following cases:

1. Without applying any correlations, the sub-tree is translated in a one-operator-to-one-job approach into five jobs, corresponding to JOIN1, AGG1, JOIN2, AGG2, and Left Outer Join1 respectively.

2. Only applying input correlation and transit correlation (ignoring job flow correlation), the sub-tree is translated into three jobs. Job1 is to batch-process JOIN1, AGG1, and AGG2. Job2 and Job3 are for JOIN2 and Left Outer Join1, respectively. For Job1, since we do not applying job flow correlation, there are no post-job computations. Its common reduce function is only used to execute the functionalities of the three merged operations (JOIN1, AGG1, and AGG2), and their own output key/value pairs will be written to the HDFS and be read again by Job2 and Job3.

3. By considering all correlations, YSmart translates the sub-tree into only one job. That means the three jobs in the above case are combined in a way that Job2 and Job3 are executed in the reduce phase of Job1.

4. We also used a hand-optimized MapReduce program to execute the sub-tree on the basis of query semantic analysis. Its major difference from YSmart is that, in its reduce function, it does not need to execute multiple operations in a strict way as indicated by the query plan tree. For example, as shown in the query plan tree and the SQL code (Appendix), if JOIN1 ($orders \bowtie lineitem$) has no output, then the sub-tree (i.e. Left

[5]http://aws.amazon.com/ec2/

[6]http://archive.cloudera.com/docs/_getting_started.html

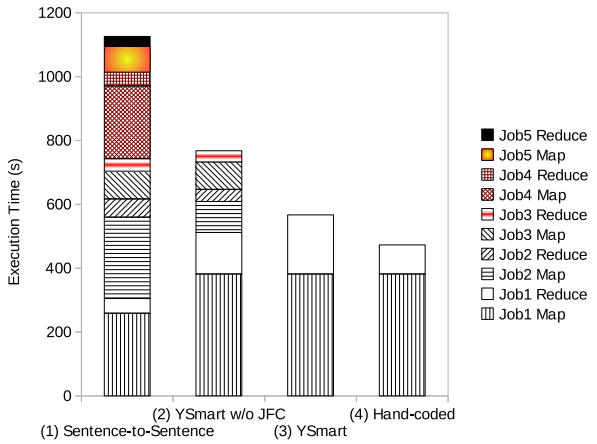


Fig. 9. Breakdown of job finishing times of Q21

Outer Join1) will certainly have no output. Thus, the existence of such type of short-paths makes it unnecessary to execute any further computations in the tree. For example, in the reduce function, if there is no input key/value pairs from *orders*, due to the selection condition $o\_orderstatus =' F'$ that is executed in the map phase, the function returns immediately since the function will certainly have no output.

Fig. 9 shows the results. Each bar shows the execution time of the map/reduce phase for each job. We ignored the time between two jobs (at most 5 seconds in our results). We have the following four observations:

First, a one-operator-to-one-job translation has the worst performance, due to its unawareness of intra-query correlations. For its total execution time (1140s), the map phases of Job1, Job2, and Job4, each of which needs a table scan on *lineitem*, take 65% of the total time (742s).

Second, when ignoring job flow correlation and only using input correlation and transit correlation, the total execution time is 773s (167% speedup over that of one-operator-to-one-job translation). It only executes one pass of scan on *lineitem* in the map phase of Job1 (387s).

Third, when using all correlations, YSmart can further decrease the total execution time to 561s (203% speedup over that of one-operator-to-one-job translation). The reduce phase (185s) is slower than the one (130s) in Job1 of the above case without job flow correlation, because it executes more lines of codes which have to be executed by two additional jobs.

Finally, by the hand-coded program, the query execution time is only 479s. YSmart is only 17% slower. As shown in the figure, the major difference between YSmart and the hand-coded program is YSmart's reduce phase (185s) is longer than that in the hand-coded program (91s).

These results show the importance of correlation awareness during SQL-to-MapReduce translations. YSmart's performance is very close to the hand-optimized program.

D. On Small Cluster: YSmart vs Hive, Pig, and DBMS

Next we show how YSmart can outperform Hive and Pig in our experiment. In this experiment, we also included PostgreSQL 8.4 on the TaskTracker node to execute these queries. Our goal is to simulate a parallel DBMS on the basis of the single-threaded PostgreSQL engine. Because the node has 4 computing cores, we assume a parallel DBMS can achieve an ideal 400% speedup. Therefore, we set the data set size (2.5GB for TPC-H and 5GB for Q-CSA) for PostgreSQL as 1/4 of the original size. Furthermore, we try our best to optimize performance of PostgreSQL with index building, query plan arrangement and buffer pool warm-up. Fig. 10 shows the job execution times for the four systems: YSmart, Hive, Pig and PostgreSQL. Due to page limit, we omit breakdowns for map/reduce phases.

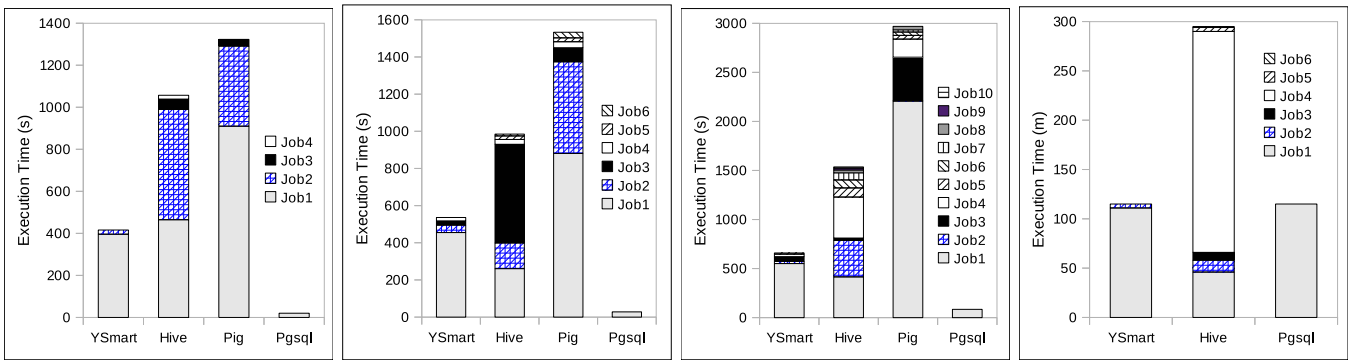
We first examine the total execution times. The results consistently show the performance advantages of YSmart over Hive and Pig. For the four queries YSmart's speedup over Hive (the consistent winner between it and Pig for all the four queries) is 258%, 190%, 252%, and 266% respectively. We notice that Pig cannot finish Q-CSA with the 20GB data set because it would generate much larger intermediate results than the capacity of our test disk.

With dynamical job composition, YSmart executes much less number of jobs than those of Hive and Pig using one-operator-to-one-job translations. For example, for Q-CSA, YSmart executes two jobs, while Hive executes six jobs with the strict operators as in the query plan shown in Fig 2(a). For Q17 by Hive, there are four jobs, and the detailed job execution breakdowns show that most of the times are spent on the jobs to scan the raw table *lineitem*. Each of the first two jobs involves a time-consuming full scan on the largest *lineitem* table. However, YSmart avoids the second pass of table scan on *lineitem*, and reduces redundant disk I/O and network transfers between a map-reduce transition.

There are two distinct observations when comparing YSmart and the ideal parallel PostgreSQL. First, for the three TPC-H queries that represent traditional data warehouse workloads, the database solution shows much better performance than the MapReduce solutions including YSmart. However, for Q-CSA that represents typical web click-stream analysis workloads, the database solution does not have significant performance advantage. Moreover, with query-correlation-awareness, YSmart can generate highly-efficient MapReduce programs that have almost the same execution time as the DBMS (note that it is normalized with 1/4 data set, i.e., 5GB).

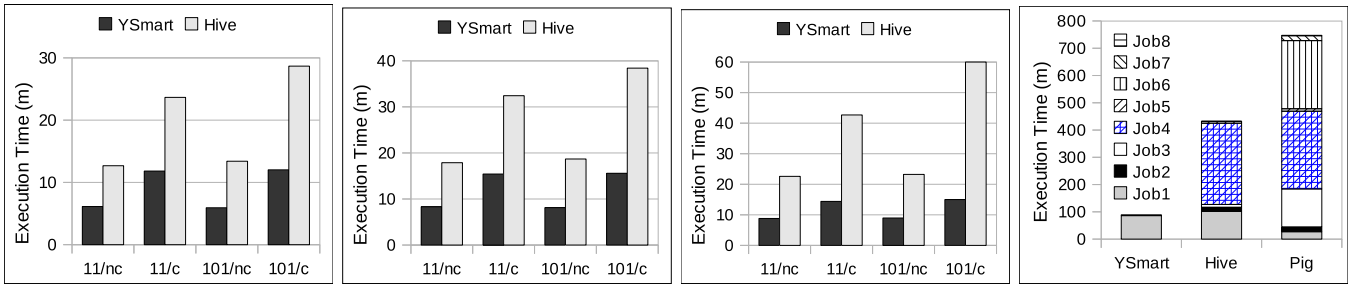
E. Results on Amazon EC2

In this section, we show YSmart's performance in two Amazon EC2 clusters with 11 nodes and 101 nodes, respectively. We conduct two groups of experiments. The first group is for the three TPC-H queries executed by YSmart and Hive. We selected different data set sizes for the two clusters respectively (10GB and 100GB), so each worker node can process one GB of data. Different from the above local cluster, query executions on the two clusters will generate a lot of data transfers via network. Therefore, we measured both the execution times by enabling map output compression (with the default configuration by the Cloudera Distribution AMI) and disabling compression. The second group is for Q-



(a) Q17 (b) Q18 (c) Q21 (d) Q-CSA

Fig. 10. Execution Breakdowns of job execution times (pgsql for the ideal parallel PostgreSQL).



(a) Q17 (b) Q18 (c) Q21 (d) Q-CSA

Fig. 11. Query execution times on Amazon EC2 11-node and 101-node clusters (c for map output compression, nc for no compression). Results for Q-CSA are only on the 11-node cluster (no compression).

CSA executed by YSmart, Hive, and Pig respectively. For this group, we only use the 11-node cluster and disable map output compression. We selected a 20GB data set for the query.

Fig. 11 (a - c) show performance comparisons between YSmart and Hive, with and without compression. Here we omit detailed job execution breakdowns since they are very similar to the ones in previously presented experiments. One special case is that Hive with compression cannot finish Q21 on the 101-node cluster in one hour, and here for drawing, we use one hour as the query execution time. Fig. 11 (d) shows performance comparisons for Q-CSA among YSmart, Hive and Pig, with detailed job execution time breakdowns. Next we summarize the three major conclusions drawn from our experiments.

First, YSmart outperforms Hive in all cases. For the TPC-H queries, without map output compression, YSmart's maximal speedup over Hive is 297% for Q21 on the 101-node cluster. For Q-CSA, YSmart has a 487% speedup over Hive and a 840% speedup over Pig on the 11-node cluster.

Second, both YSmart and Hive show nearly linear speedup from the 11-node cluster to the 101-node one. In particular, query execution times by YSmart are almost unchanged when comparing the same case between the two clusters.

Third, map output compression does not provide performance improvement, but significantly degrades performance of YSmart and Hive in all cases. For example, the execution time of Q17 in YSmart on the 101-node without compression is 5.93 minutes. However, it is increased to 12.02 minutes with compression, although the size of reduce input can be compressed from 11.09GB to 3.87GB. It reflects that, in this isolated cluster, it is not beneficial for performance to trade-off between the cost of compression/decompression and network transfer times. Note that, YSmart outperforms Hive regardless if compression is enabled, because YSmart can reduce the size of map output via merging correlated MapReduce jobs.

F. Results on Facebook's Cluster

In order to further test the scalability of YSmart, we conduct experiments on a physical cluster with 747 nodes, each of which has 8 cores, in Facebook with 1TB data set. Map output compression is not enabled. Since this is a production cluster, there are also other jobs running on it. In order to compare the performance between YSmart and Hive, for each query, we concurrently execute three YSmart instances and three Hive instances. In our tests, we find there are many unexpected dynamics in this large-scale production cluster. Moreover, our results are much more complicated than what we collect from the previous isolated cluster environments.

1) Q17: Among the three YSmart instances and three Hive instances, YSmart can outperform Hive with a maximal speedup of 310% and a minimal speedup of 230%. We show the execution time phases of the six instances in Fig. 12. The performance differences between YSmart and Hive, from the perspective of total query execution times, are similar to those at our local server and Amazon EC2 virtual clusters. However, the time breakdowns, when compared with those in Fig. 10

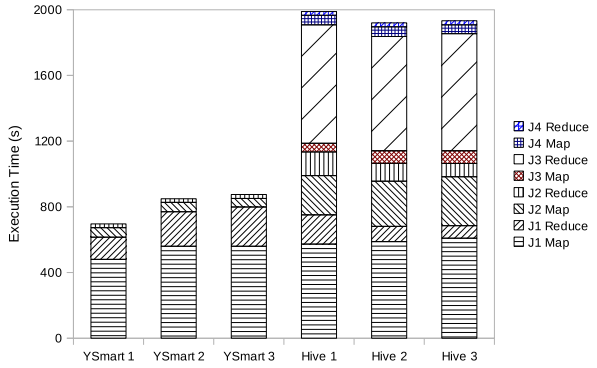


Fig. 12. Execution times of six Q17 instances on Facebook's cluster.

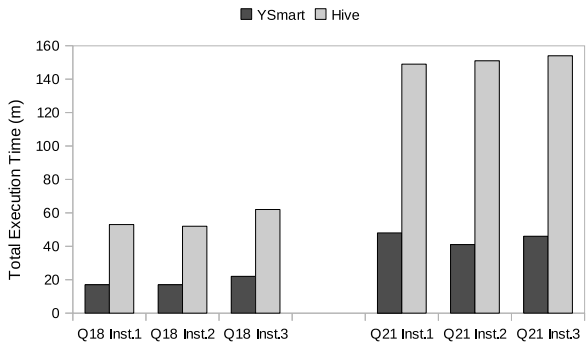


Fig. 13. Execution times of Q18 and Q21 on Facebook's cluster.

(a), show significant differences between the results in this experiment and the previous results.

For Hive, Job3 used to execute JOIN2 with the inputs from JOIN1 and AGG1 (see Fig. 4) has a notably long execution time. In the first instance of Hive (the bar for "Hive 1" in the figure), it can even take 38.9% of the whole query execution time (only 4.5% in Fig. 10 (a)). Furthermore, its reduce phase (721s) is much longer than its map phase (53s). Its fast map phase is a result of small input data sets. However, its slow reduce phase is unexpected. We believe this is because Hive cannot efficiently execute join with temporarily-generated inputs. This unexpected situation further confirms the necessary effort of reducing the number of jobs if jobs can be dynamically composed, as done by YSmart. In addition, we also find that the time between two jobs is small (at most 50s) in this experiment.

2) Q18 and Q21: Fig. 13 shows the total execution times for the two queries. We calculate the average execution times of three instances for each case. The average speedups of YSmart over Hive are 298% and 336%, respectively.

We are not able to complete the executions of the two queries on the same day as Q17 in the above section. When comparing the results for the two queries with the above results for Q17, we find a noticeable uncertain effect on this large-scale production cluster. The two queries are significantly slower than Q17, executed by YSmart or Hive. Especially for Q21, its average execution times are 3.46 times larger than that of Q17 by YSmart, and even 4.88 times larger than that of Q17 by Hive. These ratios are much higher than those in isolated clusters. For example, on the isolated Amazon EC2 cluster with 101 nodes, for YSmart, Q21 is at most 1.5 times slower than Q17. This reflects unexpected dynamics due to resource contentions of co-running workloads.

Despite the existence of such high dynamics, YSmart outperforms Hive significantly. Moreover, its speedups in this experiment are higher than in the experiments conducted on the isolated clusters. On Amazon EC2 without compression, YSmart's speedup over Hive for Q21 is at most 259% (Fig. 11(c)), while the average speedup is 336% in this experiment. One important reason is that with highly unexpected dynamics, the time interval between two sequential jobs can be very large due to job scheduling. In this experiments, we observe that the maximal interval is 5.4 minutes between the first two jobs of one Q21 instance by Hive. Because Hive executes more jobs than YSmart, it causes higher scheduling cost.

VIII. Related Work

In database systems, co-operative scan [21][22] and multi-query optimization [23][12] use shared table scans to reduce redundant computations and disk accesses. However, optimizing query execution in the MapReduce environment is more challenging due to MapReduce's two unique characteristics. First, data sharing must be maximized under the constraint of the MapReduce programming model that is based on key/value pairs. Second, the number of jobs must be minimized because of MapReduce's materialization mechanism for intermediate results and final results. Therefore, YSmart must consider all possible intra-query correlations during the translation from SQL to MapReduce.

Much work has been done recently on improving query performance in MapReduce. The first category is on enhancing the MapReduce model or extending the run-time system Hadoop. MapReduce Online [24] allows pipelined job interconnections to avoid intermediate result materialization. A *PACT* model [25] extends the MapReduce concept for complex relational operations. The HaLoop [26] framework is used to support iterative data processing workloads. These projects do not focus on SQL-to-MapReduce translation and optimization.

The second category is on improving query performance without modification of the underlying MapReduce model. Our work falls into this category. Hadoop++ [27] injects optimized UDFs into Hadoop to improve query execution performance. RCFile [28] provides a column-wise data storage structure to improve I/O performance in MapReduce-based warehouse systems. Researchers studied scheduling shared scans of large files in MapReduce [29]. MRShare [30] takes a cost model approach to optimizing both map input and output sharing in MapReduce. Since the job flow correlation is not considered, MRShare will not support batch-processing jobs that have data dependency, thus the number of jobs for executing a complex query is not always minimized. A recent work introduced an approach to optimizing joins in MapReduce [31], however, it did not consider a general correlation-exploiting mechanism for various operations. Another recent work presented a query optimization solution that can avoid

high-cost data re-partitioning when executing a complex query plan in the SCOPE system [32]. YSmart aims at providing a generic framework on translating a complex SQL query into optimized MapReduce jobs by exploiting various correlations.

IX. Conclusion

Execution of complex queries with high efficiency and high performance is critically desirable for big data analytics applications. Our solution YSmart aims at providing a generic framework to translate SQL queries into optimized MapReduce jobs, and executing them efficiently on large-scale distributed cluster systems. Our extensive experimental evaluations with various workloads in different platforms have shown the effectiveness and scalability of YSmart. YSmart will be merged into the Hive system as a patch, and will also be an independent SQL-to-MapReduce translator.

X. Acknowledgments

This work is supported in part by the US National Science Foundation under grants CCF072380 and CCF0913050, the National Cancer Institute, National Institutes of Health under contract No. HHSN261200800001E, and the National Library of Medicine under grant R01LM009239.

References

[1] L. Guo, E. Tan, S. Chen, X. Zhang, and Y. E. Zhao, "Analyzing patterns of user content generation in online social networks," in *KDD*, 2009.

[2] E. Friedman, P. M. Pawlowski, and J. Cieslewicz, "SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions," *PVLDB*, vol. 2, no. 2, pp. 1402–1413, 2009.

[3] D. J. DeWitt, E. Paulson, E. Robinson, J. F. Naughton, J. Royalty, S. Shankar, and A. Krioukov, "Clustera: an integrated computation and data management system," *PVLDB*, vol. 1, no. 1, pp. 28–41, 2008.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[5] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Visual, log-based causal tracing for performance debugging of mapreduce systems," in *ICDCS*, 2010, pp. 795–806.

[6] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "MapReduce and parallel DBMSs: friends or foes?" *Commun. ACM*, vol. 53, no. 1, pp. 64–71, 2010.

[7] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*, 2008.

[8] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a highlevel dataflow system on top of MapReduce: The Pig experience," *PVLDB*, 2009.

[9] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: easy and efficient parallel processing of massive data sets," *PVLDB*, vol. 1, no. 2, pp. 1265–1276, 2008.

[10] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - a warehousing solution over a Map-Reduce framework," *PVLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.

[11] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *PVLDB*, 2009.

[12] Y. Cao, G. C. Das, C. Y. Chan, and K.-L. Tan, "Optimizing complex queries with multiple relation instances," in *SIGMOD Conference*, 2008.

[13] S. Shekhar and S. Chawla, *Spatial Databases: A Tour*. Prentice Hall, 2003.

[14] Q. Zou, H. Wang, R. Soulé, M. Hirzel, H. Andrade, B. Gedik, and K.-L. Wu, "From a stream of relational queries to distributed stream processing," *PVLDB*, vol. 3, no. 2, pp. 1394–1405, 2010.

[15] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *SIGMOD Conference*, 2007, pp. 1029–1040.

[16] 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 *SIGMOD Conference*, 2009, pp. 165–178.

[17] S. Chaudhuri, "An overview of query optimization in relational systems," in *PODS*, 1998, pp. 34–43.

[18] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–170, 1993.

[19] B. Hindman, A. Konwinski, M. Zaharia, and I. Stoica, "A common substrate for cluster computing," in *HotCloud*, 2009.

[20] W. Kim, "On optimizing an SQL-like nested query," *ACM Trans. Database Syst.*, vol. 7, no. 3, pp. 443–469, 1982.

[21] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki, "QPipe: A simultaneously pipelined relational query engine," in *SIGMOD*, 2005.

[22] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz, "Cooperative scans: Dynamic bandwidth sharing in a dbms," in *VLDB*, 2007, pp. 723–734.

[23] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, 1988.

[24] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," in *NSDI*, 2010.

[25] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: a programming model and execution framework for web-scale analytical processing," in *ACM SoCC*, 2010, pp. 119–130.

[26] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," in *VLDB*, 2010.

[27] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)," *PVLDB*, vol. 3, no. 1, pp. 518–529, 2010.

[28] 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 *ICDE*, 2011.

[29] P. Agrawal, D. Kifer, and C. Olston, "Scheduling shared scans of large data files," *PVLDB*, vol. 1, no. 1, pp. 958–969, 2008.

[30] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "Mrshare: Sharing across multiple queries in mapreduce," in *VLDB*, 2010.

[31] F. N. Afrati and J. D. Ullman, "Optimizing joins in a Map-Reduce environment," in *EDBT*, 2010.

[32] J. Zhou, P.-Å. Larson, and R. Chaiken, "Incorporating partitioning and parallel plans into the scope optimizer," in *ICDE*, 2010, pp. 1060–1071.

XI. Appendix

The following code is corresponding to the sub-tree "Left Outer Join 1" in the plan in Fig. 8(b). The relationships between the code and the tree are as follows: 1) Lines 3-7 are for JOIN1, 2) Lines 8-12 are for AGG1, 3) Lines 2-16 are for JOIN2 that is the parent node of JOIN1 and AGG1, 4) Lines 18-23 are for AGG2, and 5) at the top level, Line 17 and line 24 show a left outer join between JOIN2 and AGG2.

```
1: SELECT sq12.l_suppkey FROM
2:   (SELECT sq1.l_orderkey,sq1.l_suppkey FROM
3:     (SELECT l_suppkey,l_orderkey
4:      FROM lineitem, orders
5:      WHERE o_orderkey = l_orderkey
6:       AND l_receiptdate>l_commitdate
7:       AND o_orderstatus = 'F') AS sq1,
8:     (SELECT l_orderkey,
9:             count(distinct l_suppkey) AS cs
10:            max(l_suppkey) AS ms
11:     FROM lineitem
12:     GROUP BY l_orderkey ) AS sq2
13:     WHERE sq1.l_orderkey = sq2.l_orderkey
14:      AND ((sq2.cs>1) OR
15:           ((sq2.cs=1) AND (sq1.l_suppkey<>sq2.ms)))
16:  ) AS sq12
17:  left outer join
18:  (SELECT l_orderkey,
19:          count(distinct l_suppkey) AS cs
20:          max(l_suppkey) AS ms
21:   FROM lineitem
22:   WHERE l_receiptdate>l_commitdate
23:   GROUP BY l_orderkey ) AS sq3
24:   ON sq12.l_orderkey = sq3.l_orderkey
25: WHERE (sq3.cs IS NULL) OR
26:       ((sq3.cs=1) AND (sq12.l_suppkey=sq3.ms))
```