MRPacker: An SQL to MapReduce Optimizer

Xuelian Lin, Yue Ye, Shuai Ma*  
SKLSDE Lab, Beihang University, China  
{linxl, yeyue, mashuai}@act.buaa.edu.cn

ABSTRACT

There have been recently quite a few works on optimizing the MapReduce execution plans, which either optimize the join operators or apply a set of translation rules to reduce the number of MapReduce jobs in an execution plan. However, none of these works has put into consideration and utilized how MapReduce jobs are generated and combined. To further improve the efficiency of MapReduce execution plans, we incorporate into our optimization approach the way how MapReduce jobs are generated and combined. In this paper, we propose MRPacker, a novel SQL-to-MapReduce optimizer by (a) using a set of transformation rules to reduce the number of MapReduce jobs, and (b) merging MapReduce jobs in a more reasonable way. We have finally experimentally demonstrated the effectiveness and efficiency of MRPacker, using the TPC-H benchmark.

Categories and Subject Descriptors

H.2.4 [DATABASE MANAGEMENT]: Systems—Query processing

General Terms

Algorithms, Performance, Design.

Keywords

SQL, MapReduce, Translator, Optimizer

1. INTRODUCTION

MapReduce [1], together with its open-source implementation Hadoop [2], has been widely adopted in many practical data processing applications. In the early days when MapReduce was initially developed, programmers implemented their data processing tasks in MapReduce with an explicitly hand-coding of the map and reduce functions. Although this provides a high flexibility for programmers, it increases the difficulty for program debugging [3] and restricts its usages to sophisticated developers only. Further, many data analysts prefer SQL-like declarative languages. This need drives the development of several SQL-like declarative languages, e.g., Pig Latin [6], HiveQL/Hive [7] and Tenzing [4]. These High Level Query Languages (HLQLs) have already played a more important role than hand-coded programs in MapReduce [5] since they can greatly simplify the efforts of Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

CIKM’13, October 27 - November 01 2013, San Francisco, CA, USA  
Copyright 2013 ACM 978-1-4503-2263-8/13/10...$15.00.  
http://dx.doi.org/10.1145/2505515.2507813

application developments by reducing hand-coded programs in MapReduce.

Normally, programs written with HLQL are compiled as query plans [12], and then an SQL-to-MapReduce translator parses each query plan into an execution plan, i.e., a sequence of MapReduce jobs. However, it has been observed that the translated MapReduce jobs are often extremely inefficient in practice, compared with the MapReduce jobs programmed by experienced programmers [9]. Furthermore, programs generated from the inefficient SQL-to-MapReduce translators would create many inefficient jobs or even unnecessary jobs [5], which results in a serious waste of computer cluster resources.

There have been recently quite a few works [5, 9, 10] that focus on optimizing the MapReduce execution plans. They either optimize the join operators [9, 10] or apply a set of translation rules to reduce the number of MapReduce jobs in an execution plan [5]. However, we found in practice and experimentally that not only the number of MapReduce jobs of execution plans, but also the way how MapReduce jobs are generated (a query operation in a query plan may be bind into the map or reduce function when it is converted into a MapReduce job by some SQL-to-MapReduce translator) and the way how MapReduce jobs are combined (it is typically common that different translators may allow or restrict different job combinations. For instance, a translator may merge a MapReduce job with its preceding or subsequent MapReduce job) could have a big impact on the efficiency of MapReduce execution plans.

Contributions. To further improve the efficiency of MapReduce execution plans, we incorporate the way how MapReduce jobs are generated and combined into our optimization approach. More specifically, we design a new SQL-to-MapReduce optimizer, i.e., MRPacker, which significantly improves the efficiency of MapReduce tasks, by (a) using a set of transformation rules to reduce the number of MapReduce jobs, and (b) merging MapReduce jobs in a more reasonable way. Experimental results over the TPC-H benchmark [11] have justified the effectiveness and efficiency of MRPacker. For instance, it reduces 30-45% and 7-18% cost than the Hive Translator and YSmart optimizer, respectively, for the TPC-H Q17 query.

2. PRELIMINARY

We first present the basic notions to be used.

Query operations. HLQLs, such as Hive [7], Pig [6], Tenzing [4], etc., implement a subset of query operations supported in SQL DML. These operations can be divided into three groups in the context of big data processing: (1) relation operations, including Project(π), Selection(σ), Join(⋈), Intersect(∩),Difference(−) and Union(∪); (2) aggregation operations, including Sum, Count, Max, Min, Average and Distinct; and (3) Order and Group operations.

Query plans. A Query plan is defined as a directed acyclic graph
(DAG), \( Q(V, E) \), where (1) \( V \) is a finite set of vertexes, in which \( u \) belongs to \( \Sigma_0 \), and denotes a query operation; (2) \( E \subseteq V \times V \) is a finite set of edges, in which \((u, v)\) denotes a data stream from vertex \( u \) to vertex \( v \). Here \( \Sigma_0 \) is the set of all query operations.

A query plan is converted to an execution plan, which will be transferred to the MapReduce platform to be executed.

### Query operations in MapReduce (MRO)

A MapReduce job that implements a set of query operations is called MRO.

#### Execution plan (MRO)

An execution plan implemented as MapReduce jobs (execution plan for short) is defined as a DAG, \( MRQ(J, D) \), where (1) \( J \subseteq \Sigma_0 \) is a finite set of vertexes, in which \( u \) belongs to \( J \), and denotes a MapReduce Job; (2) \( D \subseteq J \times J \) is a finite set of edges, in which \((u, v)\) denotes a data stream from vertex \( u \) to vertex \( v \). Here \( \Sigma_{MRO} \) is a set of MROs.

Let \( \Sigma_{MRO} \) be a set of MapReduce jobs and each job implements only one query operation, and function \( f \) is the one-to-one-mapping from \( \Sigma_0 \) to \( \Sigma_{MRO} \).

### Initial execution plan (MRO_0)

The initial execution plan, denoted as \( MRQ_0(J_0, D_0) \), is the one to one transformation from \( Q(V, E) \) to an MRO such that (1) \( \forall v \in V, \exists u \in J_0 \wedge u = f(v) \); (2) \( \forall (u, v) \in E, \exists u', v' \in J_0 \wedge (u', v') \in D_0 \).

### Equivalent execution plans.

For any input, if MRQ, and MRQ always have the equivalent output, then MRQ, and MRQ_0 are called equivalent execution plans, denoted by \( MRQ \leftrightarrow MRQ_0 \).

### Cost of MRQ

Given an execution plan, MRQ (J, D), and \( C \) is the cost function, the cost of MRQ is defined as \( C(MRQ) = \Sigma_{u \in J} C(u) \).

### Lowest cost execution plan (MRO_0)

A MRQ is an execution plan with the lowest cost, named \( MRQ_0 \), if \( \forall MRQ \in MRQ^* \), \( C(MRQ_0) \leq C(MRQ) \).

### 3. MRPACKER

We aim at finding out the lowest cost execution plan for a MapReduce query plan. As a query plan can be directly mapped to an initial execution plan, the challenge is how to find out an equivalent execution plan with the lowest cost. For this target, we focus on two sub-problems: (1) to incorporate the ways of generating and combining MapReduce jobs as well as to reduce the number of jobs in an execution plan, so as to produce a larger space of possible equivalent execution plans; and (2) to estimate the cost of execution plans with an cost model, so as to pick out the lowest cost execution plan MRQ_0.

To find out the MRQ_0, we have implemented MR PACKER, a novel SQL-to-MapReduce optimizer. MR PACKER first directly maps a MRQ into the MRQ_0. Then it transforms the MRQ_0 to equivalent MRQs by using a set of transformation rules. Finally MR PACKER identifies the MRQ_0 with an enumeration algorithm.

### 3.1 Map MRQ to the MRQ_0

#### Process.

The map function or reduce function of MapReduce is defined as a process, denoting a set of operations over the input key-value pairs.

#### Shuffle.

The partition, sort and shuffle in MapReduce is defined as \( shuffle = (pk, sort) \), where \((1) \ p k = k_1, k_2, \ldots \ k_n \) is the partition key; and \( (2) \ sort = T \ | \ F \) denotes whether the output of map is sorted (T) or not (F), and the default value is false (F).

An MRO is defined as a triple \( (map, shuffle, reduce) \), where (1) \( map \in \Sigma_0 \) denotes the Map process of MapReduce; (2) \( shuffle \in \Sigma_3 \) denotes the Shuffle process of MapReduce; and (3) \( reduce \in \Sigma_3 \) denotes the Reduce process of MapReduce. Here \( \Sigma_3 \) is a set of processes, and \( \Sigma_3 \) is a set of shuffles.

An MRO can implement multiple query operations. However, given a query plan, MR PACKER will first directly map each query operation of the query plan into an MRO_0. In Table 1, there is a list of MRO_0s, and each MRO_0 implements only one query operation. MRO_0 has the form of \( (map, \cdot, (, \cdot, shuffle, \cdot)) \) or \( (, \cdot, shuffle, reduce) \).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operations</th>
<th>MRO_0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma )</td>
<td>Selection</td>
<td>((\sigma, \cdot))</td>
<td>Map only, empty shuffle and reduce</td>
</tr>
<tr>
<td>( \pi )</td>
<td>Project</td>
<td>((\pi, \cdot))</td>
<td></td>
</tr>
<tr>
<td>( \bowtie_{k_a} )</td>
<td>Nested-Loop-Join</td>
<td>((\bowtie_{k_a}, \cdot))</td>
<td>Empty map</td>
</tr>
<tr>
<td>( \bowtie_{k_a} )</td>
<td>Hash-Join</td>
<td>((, (pk, F), \bowtie_{k_a}))</td>
<td></td>
</tr>
<tr>
<td>( \bowtie_{k_a} )</td>
<td>Sort-Merge-Join</td>
<td>((, (pk, T), \bowtie_{k_a}))</td>
<td></td>
</tr>
<tr>
<td>( \cap )</td>
<td>Intersection</td>
<td>((, (pk, T), \cap))</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Difference</td>
<td>((, (pk, T), -))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Order and Group</td>
</tr>
<tr>
<td>grp</td>
<td>Group</td>
<td>((, (pk, F), ))</td>
<td>Shuffle only, empty map and reduce</td>
</tr>
<tr>
<td>ord</td>
<td>Order</td>
<td>((, (pk, T), ))</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>Sum</td>
<td>((, (pk, F), sum))</td>
<td>Empty map</td>
</tr>
<tr>
<td>count</td>
<td>Count</td>
<td>((, (pk, F), count))</td>
<td></td>
</tr>
<tr>
<td>avg</td>
<td>Average</td>
<td>((, (pk, F), avg))</td>
<td></td>
</tr>
</tbody>
</table>

MROs are further divided in to two classes according to their operation types implemented:

1. **(1) MR_SPJ.** An MRO that implements query operations except the aggregation operations is called an MR_SPJ.

2. **(2) MR_SPJE.** An MRO that includes aggregation operations is called an MR_SPJE.

If an MR_SPJ has the form of \( (map, \cdot, ) \), then this MR_SPJ is also called an **MR_SP**. The typical MR_SP is an MRO that implements Selection, Project, Nested-Loop-Join or a composition of these operations. MR_SPs will greatly increase the ways of generating and combining MROs.

### 3.2 Deduce Equivalent MRQs

An MRQ can be transformed into equivalent MRQs by MR PACKER using a set of transformation rules when some conditions are satisfied. That is, two jobs can be merged into one MapReduce job if and only if one of them is the unique successor...
of the other, and they are partition compatible, i.e., both jobs have the same partition key or the partition key of one job is null.

3.2.1 Using Standard Translation Rules

For MR_SPJ and MR_SPJE, their combinations have four types: (MR_SP, MR_SP), (MR_SP, MR_SPJE), (MR_SPJE, MR_SP) and (MR_SPJE, MR_SPJE). MRPacker defines two standard rules, namely Rule 1 and Rule 2, for these combinations. Rule 1 and Rule 2 are called standard rules because they simply combine two MapReduce jobs in a conventional way, i.e., map and reduce to reduce.

Rule 1: If u is an MR_SP, v is an MR_SP or MR_SPJE, then \( u + v = (map_u + map_v, shuffle_u + shuffle_v, reduce_u + reduce_v) \).

Rule 2: If u is an MR_SP, v is an MR_SP or MR_SPJE, then \( u + v = (map_u + map_v, shuffle_u + shuffle_v, reduce_u + reduce_v, map_{2:1} + reduce_{2:1}), \) where (1) \( map_{1:1} + map_{2:1} = map_1; \) (2) data fields (columns) processed by \( map_{1:1} \) are disjoint with the fields (columns) processed by any aggregation operations of u.

3.2.2 Using Extended Translation Rules

As an MR_SP is a special MR_SP, there are five kinds of special combinations of MROs: (MR_SP, MR_SP), (MR_SP, MR_SP), (MR_SP, MR_SP), (MR_SP, MR_SP), and (MR_SP, MR_SP). MRPacker also defines three extended rules, namely Rules 3-5, for these five combinations. Rules 3-5 merge map of one MRO with reduce of another MRO, which increases the ways of combination and may be helpful to reduce the cost of execution plans.

Rule 3: If u is an MR_SP, and v is an MR_SP or MR_SPJE, then \( u + v = (map_u + map_v, shuffle_u + shuffle_v, reduce_u + reduce_v) \).

Rule 4: If u is an MR_SP or MR_SPJE, and v is an MR_SP, then \( u + v = (map_u, shuffle_u, reduce_u + map_v) \).

Rule 5: If u and v are both MR_SPs, then \( u + v = (\cdot, map_u + map_v) \).

If u and v are both MR_SPJ and MR_SPJE, MRPacker doesn’t define any extended rules to combine them, because any special combination of them can be achieved by using Rules 1-2 together with Rules 3-5 in the context that query operations are mapped to MROs in the ways listed in Table 1.

3.2.3 Completeness and Correctness

MRPacker can find out all equivalent execution plans of MRQ, that mapped from a query plan as the way shown in Table 1. As MRO has three classes, i.e. MR_SP, MR_SPJ and MR_SPJE, there are totally nine kinds of combinations of any two MROs: (MR_SP, MR_SP), (MR_SP, MR_SPJE), (MR_SPJE, MR_SP), (MR_SPJ, MR_SP), (MR_SPJE, MR_SPJE), (MR_SPJ, MR_SP), (MR_SP, MR_SPJE), (MR_SPJ, MR_SP) and (MR_SPJE, MR_SP). MRPacker uses the five transformation rules to cover all combinations of MROs for getting all equivalent execution plans.

3.3 Picking out the MRQ

MRPacker refers to a Hadoop MapReduce performance model [8] to calculate the costs of MROs and MRQs, and it relies on an enumeration algorithm to find out the MRQ given the equivalent execution plan set.

The enumeration algorithm (see Figure 1) is an exhaustive algorithm. It first takes MRQ as the initial state (lines 1-3). Then it applies an arbitrary rule on any edge (u, v) of MRQ to get MRQ’ (lines 5-8), uses the cost model [8] to calculate the cost of each MRQ (line 9), and compares costs and finds out the MRQ, (line 10). The above process is repeated until it has exhausted all equivalent MRQs of the MROs (lines 4, 11).

![Figure 1. Enumeration Algorithm of MRPacker](image)

4. EXPERIMENTAL STUDY

We have designed a set of experiments to show the efficiency and effectiveness of MRPacker. These experiments are designed to compare MRPacker with Hive [7] translator and YSmart [5], another two important SQL-to-MapReduce translators, to show the benefits and advantages of MRPacker.

All of the experiments are executed over data generated by the database performance benchmark tool TPC-H [11], and we use two data tables created by the tool: `lineitem` and `part`. Also, all experiments are run in a cluster of 10 computing nodes with detailed configuration shown in Table 2.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel® Core(TM) i7, <a href="mailto:860@2.80GHz">860@2.80GHz</a></th>
<th>OS</th>
<th>Linux 2.6.26-2-amd64 #1 SMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>16G</td>
<td>Hadoop</td>
<td>0.23.3</td>
</tr>
<tr>
<td>Disk</td>
<td>1T</td>
<td>Hive</td>
<td>0.9.0</td>
</tr>
<tr>
<td>Ethernet</td>
<td>1Gbps</td>
<td>Nodes</td>
<td>10</td>
</tr>
</tbody>
</table>

4.1 Compare MRPacker with other Optimizers

In these experiments, we compare MRPacker with Hive translator and YSmart. We restate the original TPC-H Q17 query written in SQL to an equivalent query written in HiveQL [7], denoted by TPC-H Q17'. We then use Hive translator, YSmart and MRPacker to generate execution plans, respectively.

For the Hive Translator and YSmart, the Nested-Loop-Join will be fixed in the map phase (see Figure 2), while MRPacker may place it into either map phase (see Figure 2.b) or reduce phase (see Figure 2.c) according to the costs of execution plans, so as to generate the lowest cost execution plan.
We execute execution plans generated by these translators, and compare their execution time. Figure 3 shows the execution time of TPC-H Q17 under these translators, where \( \alpha \) and \( \beta \) are row selectivity and column selectivity of the nest-loop-join operation. (1) When \( \alpha=2 \) and \( \beta=1.94 \), the execution time of the execution plan generated by MRPacker is about 43% and 18% shorter than that of Hive translator and that of YSmart, respectively. (2) When \( \alpha=1 \) and \( \beta=1.88 \), the execution time of the execution plan generated by MRPacker is about 35% and 7% shorter than that of Hive translator and that of YSmart, respectively. (3) When \( \alpha=0.91 \) and \( \beta=1.89 \), the execution time of the execution plan generated by MRPacker is about 40% and 15% shorter than that of Hive translator and that of YSmart, respectively. (4) When \( \alpha=2 \) and \( \beta=0.82 \), the execution time of the execution plan generated by MRPacker is about 40% and 13% shorter than that of Hive translator and that of YSmart, respectively.

\[ \text{Figure 2. Execution Plans of TPC-H Q17} \]

\[ \text{Figure 3. Execution Time of TPC-H Q17} \]

Our experiments indicate that MRPacker typically is no worse than the Hive translator and YSmart, and in cases when MR_SPs have certain data selectivity (e.g., when \( \alpha=2, \beta=0.82 \), \( \alpha=0.91, \beta=1.89 \) and \( \alpha>1, \beta>1 \)), MRPacker indeed has a much better efficiency than the Hive translator and YSmart.

5. RELATED WORK

Optimizing query executions in a MapReduce environment is very challenging, comparing with query optimization in relational systems [12], since the unique programming model of MapReduce typically causes new transformation rules and new cost models for optimizing a query execution. Many efforts have been done on improving the query performance in MapReduce.

AQUA [9] parses users’ queries into a join graph and groups the join operators so as to be evaluated by a single MapReduce job. [10] studies multi-way (natural) joins that join multiple relations in a single map-reduce. YSmart [5] provides a correlation aware SQL-to-MapReduce translator to reduce redundant computations, I/O operations and network transfers. It applies a set of rules to use the minimal number of MapReduce jobs to execute multiple correlated operations in a complex query. These projects are SQL-to-MapReduce translations and optimizations. However, they mainly focus on reducing the number of MapReduce jobs in an execution plan, while they ignore the way how MapReduce jobs are generated or combined.

6. CONCLUSION AND FUTURE WORK

MapReduce applications developed with High Level Query Languages are compiled as query plans, and then each query plan is translated to an execution plan of MapReduce jobs by certain SQL-to-MapReduce translator. This paper introduces MRPacker, a new SQL-to-MapReduce optimizer that considers both the number of MapReduce jobs and the ways how MapReduce jobs are combined, to generate an execution plan with a lower cost. We are revising our enumeration algorithm for MRPacker to further improve the performance.

ACKNOWLEDGMENTS. Shuai is a contact author; Xuelian and Shuai are supported in part by NGFR 973 grant 2014CB340304 and 863 grant 2011AA01A202, NSTM grant 2012ZX01039001, and SKLSDE grant SKLSDE-2012ZX-08.

7. REFERENCE