MRShare: Sharing Across Multiple Queries in MapReduce

Tomasz Nykiel
University of Toronto
tnykiel@cs.toronto.edu

Michalis Potamias *
Boston University
mp@cs.bu.edu

Chaitanya Mishra †
Facebook
cmishra@facebook.com

George Kollios *
Boston University
gkollios@cs.bu.edu

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

ABSTRACT

Large-scale data analysis lies in the core of modern enterprises and scientific research. With the emergence of cloud computing, the use of an analytical query processing infrastructure (e.g., Amazon EC2) can be directly mapped to monetary value. MapReduce has been a popular framework in the context of cloud computing, designed to serve long running queries (jobs) which can be processed in batch mode. Taking into account that different jobs often perform similar work, there are many opportunities for sharing. In principle, sharing similar work reduces the overall amount of work, which can lead to reducing monetary charges incurred while utilizing the processing infrastructure. In this paper we propose a sharing framework tailored to MapReduce.

Our framework, `MRShare`, transforms a batch of queries into a new batch that will be executed more efficiently, by merging jobs into groups and evaluating each group as a single query. Based on our cost model for MapReduce, we define an optimization problem and we provide a solution that derives the optimal grouping of queries. Experiments in our prototype, built on top of Hadoop, demonstrate the overall effectiveness of our approach and substantial savings.

1. INTRODUCTION

Present-day enterprise success often relies on analyzing expansive volumes of data. Even small companies invest effort and money in collecting and analyzing terabytes of data, in order to gain a competitive edge. Recently, Amazon Webservices deployed the Elastic Compute Cloud (EC2) [1], which is offered as a commodity, in exchange for money. EC2 enables third-parties to perform their analytical queries on massive datasets, abstracting the complexity entailed in building and maintaining computer clusters.

Analytical queries are usually long running tasks, suitable for batch execution. In batch execution mode, we can avoid redundant computations by *sharing work* among queries and save on total execution time. Therefore, as has been argued recently [16], it is imperative to apply ideas from *multiple query optimization* (MQO) [12, 21, 25] to analytical queries in the cloud. For EC2 users, reducing the execution time is directly translated to monetary savings. Furthermore, reducing energy consumption in data centers is a problem that has recently attracted increased interest [4]. Therefore, in this paper, we apply MQO ideas to MapReduce, the prevalent computational paradigm in the cloud.

MapReduce [11] serves as a platform for a considerable amount of massive data analysis. Besides, cloud computing has already turned MapReduce computation into commodity, e.g., with EC2's Elastic MapReduce. The MapReduce paradigm has been widely adopted, mainly because it abstracts away parallelism concerns and is very scalable. The wide scale of MapReduce's adoption for specific analytical tasks, can also be credited to Hadoop [3], a popular open source implementation. Yet, MapReduce does not readily provide the high level primitives that have led SQL and relational database systems to their success.

MapReduce logic is incorporated in several novel data analysis systems [5, 8, 10, 13, 14, 19, 22, 24]. Beyond doubt, high level language abstractions enable the underlying system to perform automatic optimization [16]. Along these lines, recently developed systems on top of Hadoop, such as Hive [22] and Pig [14], speak HiveQL [22], an SQL-like language, and Pig Latin [17], a dataflow language, respectively. They allow programmers to code using high level language abstractions, so that their programs can afterwards be compiled into MapReduce jobs. Already, standard optimizations, such as filter pushdown, are implemented in Pig [14, 16]. Pig also implements a number of work sharing optimizations using MQO ideas. However, these optimizations are usually not automatic; the programmer needs to specify the details of sharing among multiple jobs [14].

We present `MRShare`, a novel sharing framework, which enables *automatic* and principled work-sharing in MapReduce. In particular, we describe a module, which merges MapReduce jobs coming from different queries, in order to avoid performing redundant work and, ultimately, save processing time and money. To that end, we propose a cost model for MapReduce, independent of the MapReduce platform. Using the cost model, we define an optimization problem to find the optimal grouping of a set of queries and solve it, using dynamic programming. We demonstrate that significant savings are possible using our Hadoop based prototype

*G. Kollios and M. Potamias were partially supported by NSF grant CT-ISG-0831281.

†C. Mishra conducted this work at University of Toronto.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

on representative MapReduce jobs.

We summarize our contributions:

1. We discuss sharing opportunities (Section 2), define *formally* the problem of *work-sharing* in MapReduce, and propose `MRShare`, a platform independent sharing framework.
2. We propose a simple MapReduce cost-model (Section 3) and we validate it experimentally.
3. We show that finding the optimal groups of queries (merged into single queries) according to our cost model is **NP**-hard. Thus, we relax the optimization problem and solve it efficiently (Section 4).
4. We implement `MRShare` on top of Hadoop (Section 5) and present an extensive experimental analysis demonstrating the effectiveness of our techniques (Section 6).

2. BACKGROUND

In this section, we review MapReduce and discuss sharing opportunities in the MapReduce pipeline.

2.1 MapReduce Preliminaries

A MapReduce *job* consists of two stages. The *map* stage processes input key/value pairs (tuples), and produces a new list of key/value pairs for each pair. The *reduce* stage performs group-by according to the intermediate key, and produces a final key/value pair per group. In brief, the stages perform the following transformation to their input (see Appendix A for examples):

$$map(K_1, V_1) \rightarrow list(K_2, V_2)$$
$$reduce(K_2, list(V_2)) \rightarrow (K_3, V_3)$$

The computation is distributed across a cluster of hosts, with a designated *coordinator*, and the remaining *slave* machines. Map-Reduce engines utilize a *distributed file system* (DFS), instantiated at the nodes of the cluster, for storing the input and the output. The coordinator partitions the input data I_i of job J_i into m physical *input splits*, where m depends on the user-defined split size. Every input split is processed as a *map task* by a slave machine.

Map tasks: A slave assigned a map task processes its corresponding input split by reading and parsing the data into a set of input key/value pairs (K_1, V_1). Then, it applies the user-defined *map* function to produce a new set of key/value pairs (K_2, V_2). Next, the set of keys produced by map tasks is partitioned into a user-defined number of r partitions.

Reduce tasks: A slave assigned a reduce task copies the parts of the map output that refer to its partition. Once all outputs are copied, sorting is conducted to co-locate occurrences of each key K_2. Then, the user-defined *reduce* function is invoked for each group $(K_2, list(V_2))$, and the resulting output pairs (K_3, V_3) are stored in the DFS.

For ease of presentation, we will encode map and reduce functions of job J_i in the form of map and reduce *pipelines*:

$$mapping_i : I_i \rightarrow (K_{i1}, V_{i1}) \rightarrow map_i \rightarrow (K_{i2}, V_{i2}) \quad (1)$$
$$reducing_i : (K_{i2}, list(V_{i2})) \rightarrow reduce_i \rightarrow (K_{i3}, V_{i3}) \quad (2)$$

Note also that there are interesting MapReduce jobs, such as joins, that read multiple inputs and/or have multiple map pipelines. In this paper, we consider only single-input MapReduce jobs. We do not consider reducing locally with *combiners*. However, our techniques can be modified to handle scenarios with combiners.

2.2 Sharing Opportunities

We can now describe how several jobs can be processed as a single job in the `MRShare` framework, by merging their map and reduce tasks. We identify several non-trivial *sharing opportunities*. These opportunities have also been exploited in Pig [14]. For simplicity, we will use two jobs, J_i and J_j, for our presentation. Further details, together with examples can be found in Appendix B. We consider the following sharing opportunities:

Sharing Scans. To share scans, the input to both mapping pipelines for J_i and J_j must be the same. In addition, we assume that the key/value pairs are of the same type. This is a typical assumption in MapReduce settings [14]. Given that, we can merge the two pipelines into a single pipeline and scan the input data only once. Thus, the map tasks invoke the user-defined map functions for both merged jobs:

$$mapping_{ij} : I \rightarrow (K_{ij1}, V_{ij1}) \rightarrow \frac{map_i \rightarrow tag(i) + (K_{i2}, V_{i2})}{map_j \rightarrow tag(j) + (K_{j2}, V_{j2})}$$

Note that the combined pipeline $mapping_{ij}$, produces two streams of output tuples. In order to distinguish the streams at the reducer stage, each tuple is tagged with a $tag()$ part, indicating its origin. So at the reducer side, the original pipelines are no longer of the form of Equation 2. Instead, they are merged into one pipeline.

$$reducing_{ij} : \quad \frac{tag(i)+(K_{i2}, list(V_{i2})) \rightarrow reduce_i \rightarrow (K_{i3}, V_{i3})}{tag(j)+(K_{j2}, list(V_{j2})) \rightarrow reduce_j \rightarrow (K_{j3}, V_{j3})}$$

To distinguish between the tuples originating from two different jobs, we use a *tagging* technique, which enables evaluating multiple jobs within a single job. The details are deferred to Section 5.

Sharing Map Output. Assume now, that in addition to sharing scans, the map output key and value types are the same for both jobs J_i and J_j (i.e. $(K_{i2}, V_{i2}) \equiv (K_{j2}, V_{j2})$). In that case, the map output streams for J_i and J_j can also be shared. The shared map pipeline is described as follows:

$$mapping_{ij} : I \rightarrow (K_{ij1}, V_{ij1}) \rightarrow \frac{map_i}{map_j} \rightarrow tag(i) + tag(j) + (K_{ij2}, V_{ij2})$$

Here, map_i and map_j are applied to each input tuple. Then, the map output tuples coming only from map_i are tagged with $tag(i)$ only. If a map output tuple was produced from an input tuple by both map_i and map_j, it is tagged by $tag(i) + tag(j)$. Hence, any overlapping parts of the map output will be shared. Producing a smaller map output results to savings on sorting and copying intermediate data over the network.

At the reduce side, the grouping is based on K_{ij2}. Each group contains tuples belonging to both jobs, with each tuple possibly belonging to one or both jobs. The reduce stage needs to dispatch the tuples and push them to the appropriate reduce function, based on tag.

$$reducing_{ij} : \quad tag(i) + tag(j) + (K_{ij2}, list(V_{ij2})) \rightarrow \frac{reduce_i \rightarrow (K_{i3}, V_{i3})}{reduce_j \rightarrow (K_{j3}, V_{j3})}$$

Sharing Map Functions. Sometimes the map functions are identical and thus they can be executed once. At the end of the map stage two streams are produced, each tagged with its job tag. If the map output is shared, then only one stream needs to be generated. Even if only some filters are common

in both jobs, it is possible to share parts of map functions. Details and examples can be found in the appendix.

Discussion. Among the identified sharing opportunities, sharing scans and sharing map-output yield I/O savings. On the other hand, sharing map functions, and parts of map functions additionally yield CPU savings. The I/O costs, which are due to reading, sorting, copying, etc., are usually dominant, and thus, in the remainder of this paper, we concentrate on the I/O sharing opportunities. Nevertheless, we believe that sharing expensive predicates (i.e., parts of map functions) is a promising direction for future work.

3. A COST MODEL FOR MapReduce

In this section, we introduce a simple cost model for MapReduce, based on the assumption that the execution time is dominated by I/O operations. We emphasize that our cost model is based on Hadoop but it can be easily adjusted to the original MapReduce [11].

3.1 Cost without Grouping

Assume that we have a batch of n MapReduce jobs, $\mathbb{J} = \{J_1, \ldots, J_n\}$, that read from the same input file F. Recall that a MapReduce job is processed as m map tasks and r reduce tasks[1]. For a given job J_i, let $|M_i|$ be the average output size of a map task, measured in pages, and $|R_i|$ be the average input size of a reduce task. The size of the intermediate data D_i of job J_i is $|D_i| = |M_i| \cdot m = |R_i| \cdot r$.

We also define some system parameters. Let C_r be the cost of reading/writing data remotely, C_l be the cost of reading/writing data locally, and C_t be the cost of transferring data from one node to another. All costs are measured in seconds per page. The sort buffer size is $B + 1$ pages.

The total cost of executing the set of the n individual jobs is the sum of the cost T_{read} to read the data, the cost T_{sort} to do the sorting and copying at the map and reduce nodes, and the cost T_{tr} of transferring data between nodes[2]. Thus, the cost in Hadoop is:

$$T(\mathbb{J}) = T_{read}(\mathbb{J}) + T_{sort}(\mathbb{J}) + T_{tr}(\mathbb{J}) \quad (3)$$

where:

$$T_{read}(\mathbb{J}) = C_r \cdot n \cdot |F| \quad (4)$$

$$T_{sort}(\mathbb{J}) = C_l \cdot \Sigma_{i=1}^{n}(|D_i|2(\lceil \log_B |D_i| - \log_B m \rceil + \lceil \log_B m \rceil)) \quad (5)$$

$$T_{tr}(\mathbb{J}) = \Sigma_{i=1}^{n} C_t \cdot |D_i| \quad (6)$$

In the case that no sorting is performed at the map tasks [11], we consider a slightly different cost function for sorting:

$$T_{sort}(\mathbb{J}) = C_l \cdot \Sigma_{i=1}^{n}|D_i| \cdot 2 \cdot (\lceil \log_B |D_i| + log_B m - log_B r \rceil) \quad (7)$$

Since we implemented `MRShare` in Hadoop, for the remainder of this paper, we use Equation 6 to calculate the sorting cost. However, all our algorithms can be adjusted to handle the sorting cost (Equation 7) of the original MapReduce [11]. See Appendix C for further details of the cost model.

3.2 Cost with Grouping

Another way to execute $\mathbb{J}$ is to create a single group G that contains all n jobs and execute it as a single job J_G. However, as we show next, this may not be always beneficial.

[1]We assume that all jobs use the same m, r parameters, since m depends on the input size, and r is usually set based on the cluster size.

[2]We omit the cost of writing the final output, since it is the same for grouping and non-grouping scenarios.

Let $|X_m|$ be the average size of the combined output of map tasks, and $|X_r|$ be the average input size of the combined reduce tasks of job J_G. The size of the intermediate data is $|X_G| = |X_m| \cdot m = |X_r| \cdot r$. Reasoning as previously:

$$T_{read}(J_G) = C_r \cdot |F|$$
$$T_{sort}(J_G) = C_l \cdot |X_G| \cdot 2(\lceil \log_B |X_G| - \log_B m \rceil + \lceil \log_B m \rceil)$$
$$T_{tr}(J_G) = C_t \cdot |X_G|$$
$$T(J_G) = T_{read}(J_G) + T_{sort}(J_G) + T_{tr}(J_G) \quad (8)$$

We can determine if sharing is beneficial by comparing Equation 3 with 8. Let $p_i = \lceil \log_B |D_i| - \log_B m \rceil + \lceil \log_B m \rceil$ be the original number of sorting passes for job J_i. Let $p_G = \lceil \log_B |X_G| - \log_B m \rceil + \lceil \log_B m \rceil$ be the number of passes for job J_G. Let J_j be the constituent job with the largest intermediate data size $|D_j|$. Then, p_j is the number of passes it takes to sort D_j. No other original job takes more than p_j passes. We know that $\lceil \frac{|X_G|}{|D_j|} \rceil \leq n$. Then p_G is bounded from above by $\lceil \log_B (n \cdot |D_j|) - \log_B m \rceil + \lceil \log_B m \rceil$. Now, if $n \leq B$, $\lceil \log_B |D_j| - \log_B m + log_B n \rceil + \lceil \log_B m \rceil$ is at most $p_j + 1$. p_G is clearly bounded from below by p_j. Hence, after merging all n jobs, the number of sorting passes is either equal to p_j or increases by 1.

We write $p_G = p_j + \delta_G$, where $\delta_G = \{0, 1\}$. Let $d_i = |D_i|/|F|$, d_i be the *map-output-ratio* of the map stage of job J_i, and $x_G = |X_G|/|F|$ be the map-output-ratio of the merged map stage. The map-output-ratio is the ratio between the average map output size and the input size. Unlike selectivity in relational operators, this ratio can be greater than one since in MapReduce the map stage can perform arbitrary operations on the input and produce even larger output.

Let $g = C_t/C_l$ and $f = C_r/C_l$. Then, sharing is beneficial *only if* $T(J_G) \leq T(\mathbb{J})$. From Equations 3 and 8 we have:

$$f(n-1) + g\Sigma_{i=1}^{n} d_i + 2\Sigma_{i=1}^{n}(d_i \cdot p_i) - x_G(g + 2 \cdot p_G) \geq 0 \quad (9)$$

We remark that greedily grouping all jobs (the `GreedyShare` algorithm) is not always the optimal choice. If for some job J_i it holds that $p_G > p_i$, Inequality 9 may not be satisfied. The reason is that the benefits of saving scans can be canceled or even surpassed by the added costs during sorting [14]. This is also verified by our experiments.

4. GROUPING ALGORITHMS

The core component of `MRShare` is the *grouping* layer. In particular, given a batch of queries, we seek to group them so that the overall execution time is minimized.

4.1 Sharing Scans only

Here, we consider the scenario in which we share only scans. We first formulate the optimization problem and show that it is **NP**-hard. In Section 4.1.2 we relax our original problem. In Section 4.1.3 we describe `SplitJobs`, an exact dynamic programming solution for the relaxed version, and finally in Section 4.1.4 we present the final algorithm `MultiSplitJobs`. For details about the formulas see Appendix G.

4.1.1 Problem Formulation

Consider a group G of n merged jobs. Since no map output data is shared, the overall map-output-ratio of G is the sum of the original map-output-ratios of each job:

$$x_G = d_1 + \cdots + d_n \quad (10)$$

Recall that J_j is the constituent job of the group, hence $p_j = max\{p_1, \ldots, p_n\}$. Furthermore, $p_G = p_j + \delta_G$. Based on our previous analysis, we derive the savings from merging the jobs into a group G and evaluating G as a single job J_G:

$$\mathcal{SS}(G) = \Sigma_{i=1}^{n} (f - 2 \cdot d_i \cdot (p_j - p_i + \delta_G)) - f \quad (11)$$

We define the Scan-Shared Optimal Grouping problem of obtaining the optimal grouping sharing just the scans as:

PROBLEM 1 (SCAN-SHARED OPTIMAL GROUPING). *Given a set of jobs $\mathbb{J} = \{J_1, \ldots, J_n\}$, group the jobs into S non-overlapping groups $G_1, G_2, \ldots, G_S$, such that the overall sum of savings $\sum_s \mathcal{SS}(G_s)$, is maximized.*

4.1.2 A Relaxation of the Problem

In its original form, Problem 1 is **NP**-hard (see Appendix D). Thus we consider a relaxation.

Among all possible groups where J_j is the constituent job, the highest number of sorting passes occurs when J_j is merged with *all* the jobs J_i of $\mathbb{J}$ that have map-output-ratios lower than J_j (i.e., $d_i < d_j$). We define $\delta_j = \{0, 1\}$ based on this worst case scenario. Hence, for any group G with J_j as the constituent job we assign δ_G to be equal to δ_j. Then, δ_G depends only on J_j.

Thus, we define the gain of merging job J_i with a group where J_j is the constituent job:

$$gain(i, j) = f - 2 \cdot d_i \cdot (p_j - p_i + \delta_j) \quad (12)$$

The total savings of executing G versus executing each job separately is $\sum_{i=1}^{n} gain(i, j) - f$. In other words, each merged job J_i saves one scan of the input, and incurs the cost of additional sorting if the number of passes $p_j + \delta_j$ is greater than the original number of passes p_i. Also, the cost of an extra pass for job J_j must be paid if $\delta_j = 1$. Finally, we need to account for one input scan per group.

4.1.3 SplitJobs - DP for Sharing Scans

In this part, we present an exact, dynamic programming algorithm for solving the relaxed version of Problem 1. Without loss of generality, assume that the jobs are sorted according to the map-output-ratios (d_i), that is $d_1 \leq d_2 \leq \cdots \leq d_n$. Obviously, they are also sorted on p_i.

Our main observation is that the optimal grouping for the relaxed version of Problem 1 will consist of consecutive jobs in the list (see Appendix E for details.) Thus, the problem now is to split the sorted list of jobs into sublists, so that the overall savings are maximized. To do that, we can use a dynamic programming algorithm that we call `SplitJobs`.

Define $GAIN(t, u) = \sum_{t \leq i \leq u} gain(i, u)$, to be the total gain of merging a sequence of jobs $J_t, \ldots, J_u$ into a single group. Clearly, $p_u = max\{p_t, \ldots, p_u\}$. The overall group savings from merging jobs $J_t, \ldots, J_u$ become $GS(t, u) = GAIN(t, u) - f$. Clearly, $GS(t, t) = 0$. Recall that our problem is to maximize the sum of $GS()$ of all groups.

Consider an optimal arrangement of jobs $J_1, \ldots, J_l$. Suppose we know that the last group, which ends in job J_l, begins with job J_i. Hence, the preceding groups, contain the jobs $J_1, \ldots, J_{i-1}$ in the optimal arrangement. Let $c(l)$ be the savings of the optimal grouping of jobs $J_1, \ldots, J_l$. Then, we have: $c(l) = c(i-1) + GS(i, l)$. In general, $c(l)$ is:

$$c(l) = max_{1 \leq i \leq l}\{c(i-1) + GS(i, l)\} \quad (13)$$

Now, we need to determine the first job of the last group for the subproblem of jobs $J_1, \ldots J_l$. We try all possible cases for job J_l, and we pick the one that gives the greatest overall savings (i ranges from 1 to l). We can compute a table of $c(i)$ values from left to right, since each value depends only on earlier values. To keep track of the split points, we maintain a table $source()$. The pseudocode of the algorithm is:

`SplitJobs`$(J_1, \ldots, J_n)$

1. Compute $GAIN(i, l)$ for $1 \leq i \leq l \leq n$.
2. Compute $GS(i, l)$ for $1 \leq i \leq l \leq n$.
3. Compute $c(l)$ and $source(l)$ for $1 \leq l \leq n$.
4. Return c and $source$

`SplitJobs` has $O(n^2)$ time and space complexity.

4.1.4 Improving SplitJobs

Consider the following example: Assume $J_1, J_2, \ldots, J_{10}$ are sorted according to their map-output-ratio. Assume we compute the worst case δ_j value for each job. Also, assume the `SplitJobs` algorithm returns the following groups: J_1, J_2, $(J_3 J_4 J_5)$, $(J_6 J_7)$, J_8, J_9, J_{10}. We observe that for the jobs that are left as singletons, we may run the `SplitJobs` program again. Before that, we *recompute* δ_j's, omitting the jobs that have been already merged into groups. Notice that δ_j's will change. If J_j is merged with all jobs with smaller output, we may not need to increase the number of sorting passes, even if we had to in the first iteration. For example, it is possible that $\delta_{10} = 1$ in the first iteration, and $\delta_{10} = 0$ in the second iteration.

In each iteration we have the following two cases:

- The iteration returns some new groups (e.g., $(J_1 J_2 J_8)$). Then, we remove all jobs that were merged in this iteration, and we iterate again (e.g., the input for next iteration is J_9 and J_{10} only).

- The iteration returns all input singleton groups (e.g., J_1, J_2, J_8, J_9, J_{10}). Then we can safely remove the smallest job, since it does not give any savings when merged into any possible group. We iterate again with the remaining jobs (e.g., J_2, J_8, J_9, and J_{10}).

In brief, in each iteration we recompute δ_j for each job and we remove at least one job. Thus, we have at most n iterations. The final `MultiSplitJobs` algorithm is the following:

`MultiSplitJobs`$(J_1, \ldots, J_n)$
  $\mathbb{J} \leftarrow \{J_1, \ldots, J_n\}$ (input jobs)
  $\mathbb{G} \leftarrow \emptyset$ (output groups)
  while $\mathbb{J} \neq \emptyset$ **do**
    compute δ_j for each $J_j \in \mathbb{J}$
    $ALL = SplitJobs(\mathbb{J})$
    $\mathbb{G} \leftarrow \mathbb{G} \cup ALL.getNonSingletonGroups()$
    $SINGLES \leftarrow ALL.getSingletonGroups()$
    if $|SINGLES| < |\mathbb{J}|$ **then**
      $\mathbb{J} \leftarrow SINGLES$
    else
      $J_x = SINGLES.theSmallest()$
      $\mathbb{G} \leftarrow \mathbb{G} \cup \{J_x\}$
      $\mathbb{J} \leftarrow SINGLES \setminus J_x$
    end if
  end while
  return $\mathbb{G}$ as the final set of groups.

`MultiSplitJobs` yields a grouping which is at least as good as `SplitJobs` and runs in $O(n^3)$.

4.2 Sharing Map Output

We move on to consider the problem of optimal grouping when jobs share not only scans, but also their map output. For further details about some of the formulas see Appendix G.

4.2.1 Problem Formulation

The map-output-ratios, d_is of the original jobs and x_G of the merged jobs no longer satisfy Equation 10. Instead, we have:

$$x_G = \frac{|D_1 \cup \cdots \cup D_n|}{|F|}$$

where the union operator takes into account lineage of map output tuples. By Equation 9, the total savings from executing n jobs together are:

$$\mathcal{SM}(G) = (f(n-1) - x_G(g+2p_G)) + \left(g\Sigma_{i=1}^{n} d_i + 2\Sigma_{i=1}^{n}(d_i p_i)\right)$$

Our problem is to maximize the sum of savings over groups:

PROBLEM 2 (SCAN+MAP-SHARED OPTIMAL GROUPING). *Given a set of jobs $\mathbb{J} = \{J_1, \ldots, J_n\}$, group the jobs into S non-overlapping groups $G_1, G_2, \ldots, G_S$, such that the overall sum of savings $\Sigma_s \mathcal{SM}(G_s)$ is maximized.*

First, observe that the second parenthesis of $\mathcal{SM}(G)$ is constant among all possible groupings and thus can be omitted for the optimization problem. Let n_{G_s} denote the number of jobs in G_s, and p_{G_s} the final number of sorting passes for G_s. We search for a grouping into S groups, (where S is not fixed) that maximizes:

$$\sum_{s=1}^{S} \left(f \cdot (n_{G_s} - 1) - x_{G_s}(g + 2(p_{G_s})\right), \tag{14}$$

which depends on x_{G_s}, and p_{G_s}. However, any algorithm maximizing the savings now needs explicit knowledge of the size of the intermediate data of all subsets of $\mathbb{J}$. The cost of collecting this information is exponential to the number of jobs and thus this approach is unrealistic. However, the following holds (see Appendix F for the proof):

THEOREM 1. *Given a set of jobs $\mathbb{J} = \{J_1, \ldots, J_n\}$, for any two jobs J_k and J_l, if the map output of J_k is entirely contained in the map output of J_l, then there is some optimal grouping that contains a group with both J_k and J_l.*

Hence, we can greedily group jobs J_k and J_l, and treat them cost-wise as a single job J_l, which can be further merged with other jobs. We emphasize that the containment can often be determined by syntactical analysis. For example, consider two jobs J_k, and J_l which read the input $T(a, b, c)$. The map stage of J_k filters tuples by $T.a > 3$, and J_l by $T.a > 2$; both jobs perform aggregation on $T.a$. We are able to determine syntactically that the map output of J_k is entirely contained in the map output of J_l. For the remainder of the paper, we assume that all such containments have been identified.

4.2.2 A Single Parameter Problem

We consider a simpler version of Problem 2 by introducing a global parameter γ ($0 \leq \gamma \leq 1$), which quantifies the amount of sharing among all jobs. For a group G of n merged jobs, where J_j is the constituent job, γ satisfies:

$$x_G = d_j + \gamma \sum_{i=1, i\neq j}^{n} d_i \tag{15}$$

We remark that if $\gamma = 1$, then no map output is shared. In other words, this is the case considered previously, where only scans may be shared. If $\gamma = 0$ then the map output of each "smaller" job is fully contained in the map output of the job with the largest map-output. Our relaxed problem is the following:

PROBLEM 3 (γ SCAN+MAP-SHARED OPTIMAL GROUPING). *Given a set of jobs $\mathbb{J} = \{J_1, \ldots, J_n\}$, and parameter γ, group the jobs into S non-overlapping groups $G_1, G_2, \ldots, G_S$, such that the overall sum of savings is maximized.*

By Equation 9 the total savings from executing n jobs together are:

$$savings(G) = fn - f - \left(g(\gamma - 1)\sum_{i=1, i\neq j}^{n} d_i + 2d_j\delta_j + 2\sum_{i=1, i\neq j}^{n}(d_i(\gamma(p_j + \delta_j) - p_i)\right)$$

As before, we define the gain:

$$gain(i, j) = f - (g(\gamma - 1)d_i + 2 \cdot d_i \cdot (\gamma(p_j + \delta_j) - p_i))$$
$$gain(j, j) = f - 2 \cdot d_j \cdot \delta_j \tag{16}$$

The total savings of executing group G together versus executing each job separately are:

$$savings(G) = \sum_{i=1}^{n} gain(i, j) - f \tag{17}$$

Due to the monotonicity of the $gain()$ function, we can show that Problem 3 reduces to the problem of optimal partitioning of the sequence of jobs sorted according to d_i's. We devise algorithm `MultiSplitJobs`$^\gamma$ based on algorithm `MultiSplitJobs` from Section 4.1.3. $GAIN$, GS, and c are defined as in Section 4.1.3, but this time, using Equation 16 for the $gain$. We just need to modify the computation of the increase in the number of sorting passes (δ_j) for each job (i.e., we consider the number of sorting passes when job J_j is merged with all jobs with map-output-ratios smaller than d_j, but we add only the γ part of the map output of the smaller jobs).

5. IMPLEMENTING MRSHARE

We implemented our framework, `MRShare`, on top of Hadoop. However, it can be easily plugged in any MapReduce system. First, we get a batch of MapReduce jobs from queries collected in a short time interval T. The choice of T depends on the query characteristics and arrival times [6]. Then, `MultiSplitJobs` is called to compute the optimal grouping of the jobs. Afterwards, the groups are *rewritten*, using a *meta-map* and a *meta-reduce* function. These are `MRShare` specific containers, for merged map and reduce functions of multiple jobs, which are implemented as regular map and reduce functions, and their functionality relies on *tagging* (explained below). The new jobs are then submitted for execution. We remark that a simple change in the infrastructure, which in our case is Hadoop, is necessary. It involves adding the capability of writing into multiple output files on the reduce side, since now a single MapReduce job processes multiple jobs and each reduce task produces multiple outputs.

Next, we focus on the tagging technique, which enables evaluating multiple jobs within a single job.

5.1 Tagging for Sharing Only Scans

Recall from Section 2.2 that each map output tuple contains a $tag()$, which refers to exactly one original job. We include $tag()$ in the key of each map output tuple. It consists of b bits, $B_{b-1}, \ldots, B_0$. The MSB is reserved, and the remaining $b-1$ bits are mapped to jobs[3]. If the tuple belongs to job J_j, the $j-1^{th}$ bit is set. Then, the sorting is performed on the $(key + tag())$. However, the grouping is based on the key only. This implies that each group at the reduce side, will contain tuples belonging to different jobs. Given n merged MapReduce jobs $J_1, \ldots, J_n$, where $n \leq b-1$, each reduce-group contains tuples to be processed by up to n different original reduce functions (see also Appendix H).

5.2 Tagging for Sharing Map Output

We now consider sharing map output, in addition to scans. If a map output tuple originates from two jobs J_i and J_j, the $tag()$ field, after merging, has both the $i-1^{th}$ and the $j-1^{th}$ bits set. The most significant bit is set to 0, meaning that the tuple belongs to more than one job. At the reduce stage, if a tuple belongs to multiple jobs, it needs to be pushed to all respective reduce functions.

6. EMPIRICAL EVALUATION

In this section, we present an experimental evaluation of the `MRShare` framework. We ran all experiments on a cluster of 40 virtual machines, using Amazon EC2 [1]. Our real-world 30GB text dataset, consists of blog posts [2]. We present our experimental setting and the measurements of the system parameters in Appendix I. In 6.1 we validate experimentally our cost model for MapReduce. Then, in 6.2, we establish that the `GreedyShare` policy is not always beneficial. Subsequently, in 6.3 we evaluate our `MultiSplitJobs` algorithm for sharing scans. In 6.4 we demonstrate that sharing intermediate data can introduce tremendous savings. Finally, in 6.5 we evaluate our heuristic algorithm `MultiSplitJobs`$^{\gamma}$ for sharing scans and intermediate data. We show scale independence of our approach in Appendix I.

6.1 Validation of the Cost Model

First, we validate the cost model that we presented in Section 3. In order to estimate the system parameters, we used a text dataset of 10GBs and a set of MapReduce jobs based on random *grep-wordcount* queries (see Appendix I) with map-output-ratio of 0.35. Recall that the map-output-ratio of a job is the ratio of the map stage output size to the map stage input size. We run the queries using Hadoop on a small cluster of 10 machines. We measured the running times and we derived the values for the system parameters.

Then, we used new datasets of various sizes between 10 and 50 GBs and a new set of queries with map-output-ratio of 0.7. We run queries on the same cluster and we measured the running times in seconds. In addition, we used the cost model from Section 3 to derive the estimated running times. The results are shown in Figure 1(a). As we can see, the prediction of the model is very close to the actual values.

6.2 The GreedyShare Approach

Here, we demonstrate that the `GreedyShare` approach for sharing scans is not always beneficial. We created three random series of 16 *grep-wordcount* jobs $(H1, H2, H3)$, which we describe in Table 1, with constraints on the average map-output-ratios (d_i). The maximum d_i was equal to 1.35.

[3]We decided to reserve a bit for future use of sharing among jobs processing multiple inputs

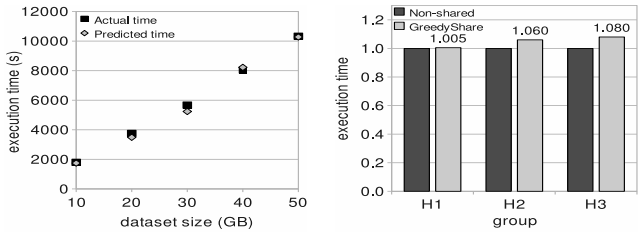

Figure 1: (a) Cost Model (b) Non-shared vs `GreedyShare`. Sharing is not always beneficial.

Table 1: `GreedyShare` setting

Series	Size	map-output-ratio d_i
H1	16	$0.3 \leq d_i \leq 0.7$
H2	16	$0.7 \leq d_i$
H3	16	$0.9 \leq d_i$

For each of the series we compare running all the jobs separately versus merging them in a single group. Figure 1(b) illustrates our results. For each group, the execution times when no grouping was performed, is normalized to 1.00. The measurements were taken for multiple randomly generated groups. Even for $H1$, where the map output sizes were, on average, half size of the input, sharing was not beneficial. For $H2$ and $H3$, the performance decreased despite savings introduced by scan-sharing, e.g., for $H2$, the 6% increase incurs an overhead of more than 30 minutes.

Obviously, the `GreedyShare` yields poorer performance as we increase the size of the intermediate data. This confirms our claim, that sharing in MapReduce has associated costs, which depend on the size of the intermediate data.

6.3 MultiSplitJobs Evaluation

Here, we study the effectiveness of the `MultiSplitJobs` algorithm for sharing scans. As discussed in Section 3, the savings depend strongly on the intermediate data sizes. We created five series of jobs varying the map-output-ratios (d_i), as described in Table 2. Series $G1, G2, G3$ contained jobs with increasing sizes of the intermediate data, while $G4$ and $G5$ consisted of random jobs. We performed the experiment for multiple random series $G1, \ldots, G5$ of *grep-wordcount* jobs.

Figure 2(a) illustrates the comparison between the performance of individual-query execution, and the execution of merged groups obtained by the `MultiSplitJobs` algorithm. The left bar for each series is 1.00, and it represents the time to execute the jobs individually. The left bar is split into two parts: the upper part represents the percentage of the execution time spent for scanning, and the lower part represents the rest of the execution time. Obviously, the savings cannot exceed the time spent for scanning for all the queries. The right bar represents the time to execute the jobs according to the grouping obtained from `MultiSplitJobs`.

We observe that even for jobs with large map output sizes $(G1)$, `MultiSplitJobs` yields savings (contrary to the `GreedyShare` approach). For $G2$ and $G3$, which exhibit smaller map-output sizes, our approach yielded higher savings. For jobs with map-output sizes lower than 0.2 $(G3)$ we achieved up to 25% improvement.

In every case, `MultiSplitJobs` yields substantial savings

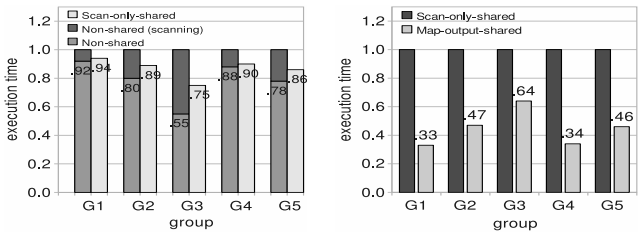

Figure 2: `MultiSplitJobs` **(a) scans (b) map-output.**

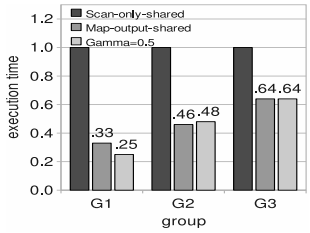

Figure 3: `MultiSplitJobs`$^{\gamma}$ **behaviour**

Table 2: `MultiSplitJobs` **setup and grouping example**

Series	Size	map-output-ratio d_i	#groups	Group-size
G1	16	$0.7 \leq d_i$	4	5,4,4,3
G2	16	$0.2 \leq d_i \leq 0.7$	3	8,5,3
G3	16	$0 \leq d_i \leq 0.2$	2	8,8
G4	16	$0 \leq d_i \leq max$	3	7,5,4
G5	64	$0 \leq d_i \leq max$	5	16,15,14,9,5,4

with respect to the original time spent for scanning. For example, for $G4$ the original time spent for scanning was 12%. `MultiSplitJobs` reduced the overall time by 10%, which is very close to the ideal case. We do not report results for `SplitJobs` since `MultiSplitJobs` is provably better.

In addition, Table 2 presents example groupings obtained for each series $G1, \ldots, G5$. Observe that in every case, the optimal grouping consisted of groups of variable size. With increasing intermediate data sizes, the number of groups increases, and the groups become smaller. Indeed, merging many jobs with high map-output-ratios is likely to increase the number of sorting passes, and degrade the performance.

6.4 Map Output Sharing Utility

Here, we demonstrate that sharing intermediate data (i.e., map output) introduces additional savings. For the purpose of this experiment we used the same series of jobs as in the previous Section (Table 2). We used the same groupings obtained by `MultiSplitJobs`. We enable the sharing-map-outputs feature and we compare it to sharing scans only.

Our results are illustrated in Figure 2(b). For $G1$, where map output sizes were large, the jobs shared large portions of intermediate data. The execution time dropped by 67%. Recall, that sharing intermediate data, introduces savings on copying data over the network and sorting. $G2$ and $G3$ had smaller intermediate map output sizes and the savings were lower, but still significant. For $G4$ (fully random jobs) we achieved savings up to 65%. Clearly, the greater the redundancy among queries, the more the savings. We emphasize that the degree of sharing of the intermediate data is query dependent. Note also that `MultiSplitJobs` does not provide the optimal solution in this case (i.e., some other grouping could yield even higher savings), since it has no information on the degree of sharing among map outputs. Even so, our experiments show significant savings.

6.5 MultiSplitJobs$^{\gamma}$ Evaluation

Finally, we study the performance of the `MultiSplitJobs`$^{\gamma}$ algorithm, where γ is the global parameter indicating the desired aggressiveness of sharing. We ran the series of jobs $G1, \ldots, G3$ described before, and set $\gamma = 0.5$. Our objective, is to study the possible outcomes when using `MultiSplitJobs`$^{\gamma}$.

With respect to the value for γ we remark the following. In some cases, we are able to determine γ syntactically, e.g., when the aggregation key is different for each job, then no map output sharing is possible and $\gamma = 1.00$. On the other hand, if filters in the map stage for different jobs have hierarchical structure (each job's map stage produces a superset of map output of all jobs with smaller map output), we can set $\gamma = 0.00$. By tuning γ, we can be more pessimistic or more optimistic. In principle, a *good* γ can be learned from the data (e.g., sampling of the current workload, statistics from previous runs), but this issue is out of the scope of this paper. However, we note that when gamma is set to 1 the scheme performs at least as well as `MultiSplitJobs`.

Figure 3 illustrates the results. We compare `MultiSplitJobs` with the map-output-sharing feature disabled (left bar) and enabled (center), with `MultiSplitJobs`$^{\gamma}$ for $\gamma = 0.5$ (right), which allows for more aggressive merging.

For $G1$ we achieved further savings. For $G2$ the average execution time increased with respect to the original solution with map-output-sharing enabled; $\gamma = 0.5$ was too optimistic, since the jobs shared less on average. It caused more aggressive merging, and, in effect, degraded the performance. For $G3$ setting $\gamma = 0.5$ did not cause any changes.

We conclude that `MultiSplitJobs`$^{\gamma}$ can introduce additional savings, if the choice of γ is appropriate. If the choice is too pessimistic, the additional savings may be moderate. If it is too optimistic, the savings might decrease with respect to `MultiSplitJobs`.

6.6 Discussion

Overall, our evaluation demonstrated that substantial savings are possible in MapReduce. Our experimental evaluation on EC2 utilized 8000 machine hours, with a cost of $800. Introducing even 20% savings in the execution time, translates into $160. Our experiments confirm our initial claim, that work-sharing in the MapReduce setting, may yield significant monetary savings.

7. RELATED WORK

MapReduce. Since its original publication [11], MapReduce style computation has become the norm for certain analytical tasks. Furthermore, it is now offered as a cloud service from Amazon EC2 [1]. Moreover, MapReduce logic has been integrated as a core component in various projects towards novel, alternative data analysis systems [5, 8, 10, 14, 13, 19, 22, 24]. Hadoop [3] is the most popular open source implementation of MapReduce and serves as the platform for many projects [5, 10, 14, 22], including ours.

MapReduce systems. There has been an increased interest in combining MapReduce and traditional database systems in an effort to maintain the benefits of both. Projects such as Pig [17], Hive [22], and Scope [8] focus on providing high-level SQL-like abstractions on top of MapReduce engines, to enable programmers to specify more complex queries in an easier way. SQL/MapReduce [13] in-

tegrates MapReduce functionality for UDF processing in Asterdata's nCluster, a shared nothing parallel database. Greenplum's [10] approach is similar. HadoopDB [5] is an architectural hybrid of MapReduce and relational databases, that is based on the findings of an experimental comparison between Hadoop and parallel database systems [18] and tries to combine the advantages of both approaches. MapReduce-Merge extends MapReduce by adding a *merger* step which combines multiple reducers' outputs [23]. Our framework, `MRShare`, enables work-sharing within MapReduce, and relies only on the core MapReduce functionality. Thus, it is complementary to all aforementioned systems.

Work sharing. Cooperative scans have been studies in traditional database systems [20, 26]. Among the MapReduce systems, Hive [22] supports user-defined scan-sharing. Given two jobs reading from the same file, Hive adds a new, preprocessing MapReduce job. This job reads and parses the data in order to create two temporary files, which the original jobs will eventually read. No automatic optimization is supported, and the execution time can be worse than without sharing, due to the newly added job. Contrary to Hive, `MRShare` shares scans by creating a single job for multiple jobs, with no use of temporary files. Besides both our cost model and our experimental analysis confirm that greedily sharing scans is not always beneficial. Pig [14] supports a large number of sharing mechanisms among multiple queries, including shared scans, partial sharing of map pipelines, and even partial sharing of the reduce pipeline, by executing multiple queries in a single group. However no cost-based optimization takes place. In this paper, we provide novel algorithms that provably perform automatic *beneficial* sharing. We perform cost-based optimization without the user's interference, for a set of ad-hoc queries. On another perspective, scheduling scans for MapReduce has been considered by Agrawal et al. [6] for a dynamic environment. Their objective is to schedule jobs, so that more scans will get to be shared eventually, while making sure that jobs will not suffer from starvation. Finally, in dynamic settings, work-sharing can be performed at runtime [15, 7].

8. CONCLUSION AND FUTURE WORK

This paper described `MRShare` - the first principled analysis for automatic work-sharing across multiple MapReduce jobs. Based on specific sharing opportunities that we identified and our cost model for MapReduce we defined and solved several optimization problems. Moreover, we described a system that implements the `MRShare` functionality on top of Hadoop. Our experiments on Amazon EC2 demonstrated that our approach yields vast savings.

There are plenty of directions for future work – this work is a first step towards automatic optimization for MapReduce. We have not yet considered scenarios where the jobs operate on multiple inputs (e.g., joins). Also, sharing parts of map functions was identified as a sharing opportunity, but not addressed. Finally, we aim to handle sequences of jobs, which are common in systems like Hive [22] or Pig [17].

9. REFERENCES

[1] Amazon EC2. `http://aws.amazon.com/ec2/`.

[2] Blogscope. http://www.blogscope.net/.

[3] Hadoop project. `http://hadoop.apache.org/`.

[4] Saving energy in datacenters. `http://www1.eere.energy.gov/industry/datacenters/`.

[5] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. In *VLDB*, 2009.

[6] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *Proc. VLDB Endow.*, 1(1):958–969, 2008.

[7] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. In *VLDB*, 2009.

[8] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[9] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.*, 24(2):177–228, 1999.

[10] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. Mad skills: New analysis practices for big data. *PVLDB*, 2(2):1481–1492, 2009.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150.

[12] S. Finkelstein. Common expression analysis in database applications. In *SIGMOD '82*, pages 235–245, 1982.

[13] E. Friedman, P. Pawlowski, and J. Cieslewicz. Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. In *VLDB*, 2009.

[14] 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*, 2(2):1414–1425, 2009.

[15] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *SIGMOD '05*, pages 383–394, 2005.

[16] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Tech. Conf.*, pages 267–273, 2008.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[18] 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 '09*, pages 165–178, 2009.

[19] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[20] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *Proc. VLDB Endow.*, 1(1):610–621, 2008.

[21] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[22] 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. In *VLDB*, 2009.

[23] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, 2007.

[24] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[25] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD '07*, pages 533–544, 2007.

[26] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a dbms. In *VLDB '07*, pages 723–734, 2007.

APPENDIX

A. EXAMPLES OF MapReduce JOBS

As an example, in a *wordcount* job, the map stage reads the input text, line by line, and emits a tuple *(word, 1)* for each *word*. The reduce stage counts all tuples corresponding to a particular *word*, and emits a final output tuple (*word*, *group_cardinality*).

An aggregation on a relational table can be evaluated in a similar way [22]. Each input tuple represents a relational tuple with a specified schema. After parsing the input tuple, the map stage emits a key/value pair of the form (*aggr_col*, *aggr_value*). The reduce stage groups all tuples corresponding to a particular value in the aggregation column, and computes the final value of the aggregate function.

B. EXAMPLES OF SHARING OPPORTUNITIES

In this section we provide several useful examples of sharing opportunities. For illustration purposes we use SQL notation. An SQL group-by query, over a table $T(a, b, c)$:

```
SELECT   T.a, aggr(T.b)
FROM     T
WHERE    T.c > 10
GROUP BY T.a
```

can be translated into the following MapReduce job:

Map tasks: A slave assigned a map task reads the corresponding input split and parses the data according to $T(a, b, c)$ schema. The input key/value pairs are of the form $(\emptyset, (T.a, T.b, T.c))$. Each tuple is checked against $T.c > 10$, and if it passes the filter, a map output tuple of the form $(T.a, T.b)$ is produced. The output is partitioned into a user-defined number of r partitions.

Reduce tasks: A slave assigned a reduce task copies the parts of the map output corresponding to its partition. Once the copying is done, the outputs are sorted to co-locate occurences of each key $T.a$. Then, the aggregate function $aggr()$ is applied for each group, and the final output tuples are produced.

Example 1 (Sharing Scans - Aggregation). *Consider an input table $T(a, b, c)$, and the following queries:*

```
SELECT   T.a, sum(T.b)          SELECT   T.c, avg(T.b)
FROM     T                      FROM     T
WHERE    T.c > 10               WHERE    T.a = 100
GROUP BY T.a                    GROUP BY T.c
```

The original map pipelines are:

$$mapping_i : T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow filter(T.c > 10) \rightarrow (T.a, T.b)$$
$$mapping_j : T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow filter(T.a = 100) \rightarrow (T.c, T.b)$$

The shared scan conditions are met. Thus, the merged pipeline is:

$$mapping_{ij} : T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow \rightarrow \frac{filter(T.c>10) \rightarrow tag(i)+(T.a,T.b)}{filter(T.a=100) \rightarrow tag(j)+(T.c,T.b)}$$

The reduce stage groups tuples based on their key. If a tuple contains $tag(i)$, the reduce stage pushes the tuple to $reduce_i$, otherwise it pushes the tuple to $reduce_j$.

$$reducing_{ij} : \frac{tag(i) + (T.a, T.b) \rightarrow sum(T.b) \rightarrow (T.a, sum)}{tag(j) + (T.c, T.b) \rightarrow avg(T.b) \rightarrow (T.c, avg)}$$

In this scenario, the savings result from scanning and parsing the input only once. Clearly, this sharing scheme can be easily extended to multiple jobs. Note that there is no sharing at the reduce stage. After grouping at the reduce side, each tuple has either $tag(i)$ or $tag(j)$ attached, hence we can easily push each tuple to the appropriate reduce function. The size of the intermediate data processed is the same as in the case of two different jobs, with a minor overhead that comes from the tags.

Example 2 (Sharing Map Output - Aggregation). *Consider an input table $T(a, b, c)$, and the following queries:*

```
SELECT   T.a, sum(T.b)          SELECT   T.a, avg(T.b)
FROM     T                      FROM     T
WHERE    T.a>10 AND T.a<20      WHERE    T.b>10 AND T.c<100
GROUP BY T.a                    GROUP BY T.a
```

The map pipelines are described as follows:

$$mapping_i : T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow \rightarrow filter(T.a > 10), filter(T.a < 20) \rightarrow (T.a, T.b)$$
$$mapping_j : T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow \rightarrow filter(T.b > 10), filter(T.c < 100) \rightarrow (T.a, T.b)$$

The map functions are not the same. However, the filtering can produce overlapping sets of tuples. The map output key (T.a) and value (T.b) types are the same. Hence, we can share the overlapping parts of map output.

$$mapping_{ij} : T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow \frac{filter(T.a>10, T.a<20)}{filter(T.b>10, T.c<100)} \rightarrow tag(i) + tag(j) + (T.a, T.b)$$

The reduce stage applies the appropriate reduce function by dispatching the tuples based on tag():

$$reducing_{ij} : tag(i) + tag(j) + (T.a, T.b) \rightarrow \frac{sum(T.b) \rightarrow (T.a, sum)}{avg(T.b) \rightarrow (T.c, avg)}$$

Producing a smaller map output results to savings on sorting and copying intermediate data over the network. This mechanism can be easily generalized to more than two jobs.

Example 3 (Sharing Map - Aggregation). *Consider an input table $T(a, b, c)$, and the following queries:*

```
SELECT   T.c, sum(T.b)          SELECT   T.a, avg(T.b)
FROM     T                      FROM     T
WHERE    T.c > 10               WHERE    T.c > 10
GROUP BY T.c                    GROUP BY T.a
```

The map pipelines are described as follows:

$$mapping_j : I_j \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow filter(T.c > 10) \rightarrow (T.c, T.b)$$
$$mapping_i : I_i \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow filter(T.c > 10) \rightarrow (T.a, T.b)$$

The map pipelines are identical. Note that in this case, by identical map pipelines we mean the parsing and the set of filers/transformations in the map function - the map output key and value types are not necessarily the same. After merging we have:

$$mapping_{ij} : T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow filter(T.c > 10) \rightarrow \rightarrow \frac{tag(i) + (T.c, T.b)}{tag(j) + (T.a, T.b)}$$

If, additionally, the map output key and value types are the same, we can apply map output sharing as well. In our

example, assuming that the second query groups by T.c instead of T.a, we would have:

$$mapping_{ij} : T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow filter(T.c > 10) \rightarrow \\ \rightarrow tag(i) + tag(j) + (T.c, T.b)$$

The reducing pipeline is similar to the previous examples.

EXAMPLE 4 (SHARING PARTS OF MAP - AGGREGATION). *Consider an input table $T(a, b, c)$, and the following queries:*

```
SELECT   T.a, sum(T.b)          SELECT   T.a, avg(T.b)
FROM     T                      FROM     T
WHERE    T.c>10 AND T.a<20      WHERE    T.c>10 AND T.c<100
GROUP BY T.a                    GROUP BY T.a
```

The map pipelines are described as follows:

$$mapping_i : \quad T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow \\ \rightarrow filter(T.c > 10), filter(T.a < 20) \rightarrow (T.a, T.b)$$

$$mapping_j : \quad T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow \\ \rightarrow filter(T.c > 10), filter(T.c < 100) \rightarrow (T.a, T.b)$$

In this case, the map pipelines are not the same. However, some of their filters overlap:

$$mapping_{ij} : \quad T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow filter(T.c > 10) \rightarrow \\ \rightarrow \frac{filter(T.a<20)}{filter(T.c<100)} \rightarrow \frac{tag(i)+(T.a,T.b)}{tag(j)+(T.a,T.b)}$$

Also in this case, the key and value types of map output tuples are the same, and we can apply map output sharing:

$$mapping_{ij} : \quad T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow filter(T.c > 10) \rightarrow \\ \rightarrow \frac{filter(T.a<20)}{filter(T.c<100)} \rightarrow tag(i) + tag(j) + (T.a, T.b)$$

We remark that sharing parts of map functions has many implications. It involves identifying common subexpressions [12, 21] and filter reordering [9], which are hard problems.

C. COST MODEL FOR MapReduce DETAILS

Recall that for a given job J_i, $|M_i|$ is the average output size of a map task, measured in pages, and $|R_i|$ is the average input size of a reduce task. The size of the intermediate data D_i of job J_i is $|D_i| = |M_i| \cdot m = |R_i| \cdot r$. In the no-grouping scenario, the cost of reading the data is:

$$T_{read}(\mathbb{J}) = C_r \cdot n \cdot |F|$$

Hadoop buffers and sorts map outputs locally at the map task side. We recall that The cost of sorting and writing the data at the output of the map tasks for the n jobs is approximately:

$$\begin{aligned} T_{sort-map}(\mathbb{J}) &= C_l \cdot \Sigma_{i=1}^{n}(m \times |M_i|(2 + 2(\lceil \log_B \tfrac{|M_i|}{(B+1)} \rceil))) \\ &= C_l \cdot \Sigma_{i=1}^{n}(|D_i|(2 + 2(\lceil \log_B \tfrac{|D_i|}{(B+1)\times m} \rceil))) \\ &\approx C_l \cdot \Sigma_{i=1}^{n}(|D_i| \lceil 2(\log_B |D_i| - \log_B m) \rceil) \end{aligned} \quad (18)$$

At the reduce task side we start with m sorted runs. A merge step of the m runs involves $\lceil \log_B m \rceil$ passes. Therefore the sorting cost at the reduce tasks is:

$$\begin{aligned} T_{sort-red}(\mathbb{J}) &= C_l \cdot \Sigma_{i=1}^{n}(r \times |R_i| \lceil 2 \log_B m \rceil) \\ &= C_l \cdot \Sigma_{i=1}^{n}(|D_i| \lceil 2 \log_B m \rceil) \end{aligned}$$

Thus the total cost of sorting is:

$$T_{sort}(\mathbb{J}) = C_l \cdot \Sigma_{i=1}^{n}(|D_i| 2(\lceil \log_B |D_i| - \log_B m \rceil + \lceil \log_B m \rceil)) \quad (19)$$

Finally, the cost of transferring intermediate data is:

$$T_{tr}(\mathbb{J}) = \Sigma_{i=1}^{n} C_t \cdot |D_i| \quad (20)$$

The overall cost is the sum of the above:

$$T(\mathbb{J}) = T_{read}(\mathbb{J}) + T_{sort}(\mathbb{J}) + T_{tr}(\mathbb{J}) \quad (21)$$

D. HARDNESS OF PROBLEM 1

THEOREM 2. *Scan-Shared Optimal Grouping (Problem 1) is* **NP***-hard.*

Proof. We reduce the Set-Partitioning (SP) problem to the Scan-Shared Optimal Grouping (SSOG) problem. The SP problem is to decide whether a given multiset of integers $\{a_1, \ldots, a_n\}$ can be partitioned into two "halves" that have the same sum $t = \frac{\sum_{i=1}^{n} a_i}{2}$. Without loss of generality we can assume that $\forall i \; a_i < t$, otherwise the answer is immediate.

Every instance of the SP problem, can be transformed into a valid instance of the SSOG as follows. Let the size of the sort buffer B be equal to t, and the size of the input data be equal to $2 \cdot t$. We construct a job J_i for each of a_i in the set. Let the map output size for each job J_i be $|D_i| = a_i$, then the map-output-ratio of J_i is $d_i = \frac{a_i}{2 \cdot t}$. The number of sorting passes for each job J_i is $p_i = \lceil \log_B |D_i| - \log_B m \rceil + \lceil \log_B m \rceil$. Let $m = 1$, then $p_i = \lceil \log_B |D_i| \rceil$. Since $\forall i \; a_i < t$, hence $\forall i \; a_i < B$, then $\forall i \; p_i = 0$ – the map output of each job J_i is sorted in memory. We also set $f = 1.00$. This is a valid instance of the SSOG problem.

An optimal solution for SSOG with two group exists, if and only if, there exists a partitioning of the original SP problem. For a group G_s of n_{G_s} jobs, $p_{G_s} = \lceil \log_B |X_{G_s}| \rceil$. Hence, $p_{G_s} = 1$ for all G_s such that $|X_{G_s}| > B$ (aka $x_{G_s} > 0.5$) – if the size of the intermediate data exceeds the buffer size we need one additional sorting pass. By our assumption on f, for any such group G_s, the savings are $\mathcal{SS}(G_s) = (n_{G_s} \cdot f - 2 \cdot x_{G_s} \cdot p_{G_s}) - f < 0$. Hence, it is better to execute the jobs in G_s separately. We conclude that the final solution will have only groups G_s with $p_{G_s} = 0$, and for any group G_s in the solution $\mathcal{SS}(G_s) = n_{G_s} \times f - f$. We maximize $\mathcal{SS}(G_s)$ over all S groups, but $\Sigma_{s=1}^{S} n_{G_s} \cdot f$ is constant among all groupings, hence our problem is equivalent to minimizing the number of groups.

If the optimal solution consists of only two groups G_1 and G_2, then there exists a partitioning of jobs J_i into two sets, such that $\Sigma_{J_i \in G_1} D_i = \Sigma_{J_i \in G_2} D_i = B = t$, which is a solution to the original SP problem. Since we are minimizing the number of groups (subject to constraints), then if there is a partitioning of the original SP problem, our algorithm will return two groups (as three groups would yield lower savings). We conclude that the exact SSOG problem is **NP**-hard.

E. RELAXED PROBLEM 1 PROPERTY

THEOREM 3. *Given a list of jobs $\mathbb{J} = \{J_1, \ldots, J_n\}$ and assuming that the jobs are sorted according to the map-output-ratios (d_i), each group of the optimal grouping of the relaxed version of Problem 1 will consist of consecutive jobs as they appear in the list.*

Proof. Assume the optimal solution contains the following group: $G_s = (t, \ldots, u-1, u+1, \ldots, v)$, which is sorted

by the indices (and thus d_is, and $p_i + \delta_i$s). Observe that group G_s does not contain u.

i) $\{u\}$ is a singleton group. If $gain(u, v) > 0$ then putting u into this group would yield higher savings (as we will have one less group), hence the solution can not be optimal. If $gain(u, v) < 0$, then also $gain(u-1, v) < 0$. Hence executing $\{u-1\}$ as a singleton would give higher savings. Same for all $\{t, \ldots, u-1\}$. The given solution when $\{u\}$ would be a singleton cannot be the optimal solution.

ii) $\{u\}$ is not a singleton group.

- $\{u\}$ is in $G_{s+1} = (u, w, \ldots, z)$ where $z > v$. If $p_v + \delta_v = p_z + \delta_z$ (the final number of sorting passes are equal) then G_{s+1} and G_s can be merged with no cost yielding higher savings (having one scan versus two). Hence this would not be the optimal solution. By our sorting criterion: $p_z + \delta_z > p_v + \delta_v \geq p_u + \delta_u$, hence $gain(u, z) < gain(u, v)$. Putting $\{u\}$ to G_s yields higher savings, hence the partitioning is not optimal.
- $\{u\}$ is in $G_{s-1} = (w, \ldots, x, u, z)$ where $z < v$. Again we know that $p_v + \delta_v > p_z + \delta_z \geq p_u + \delta_u$. But then $gain(u-1, z) > gain(u-1, v)$. Putting $\{u-1\}$ (all $\{t, \ldots u-1\}$) to G_{s-1} yields higher savings. Similarly, if $G_{s-1} = (w, \ldots, x, u)$ (u is the constituent job), putting $\{u-1\}$ to G_{s-1} yields higher savings.

F. PROOF OF THEOREM 1

Proof. Assume that the optimal solution $\mathcal{S}$ contains two groups G_1 and G_2, such that $J_k \in G_1$ and $J_l \in G_2$. Then $\mathcal{SM}(G_1) = f \cdot (n_{G_1} - 1) - x_{G_1}(g + 2(p_{G_1}))$, and same for G_2. By switching J_k to G_2, we obtain $\mathcal{SM}(G_2 \cup \{J_k\}) = \mathcal{SM}(G_2) + f$, since $x_{G_2 \cup \{J_k\}} = x_{G_2}$. However, $\mathcal{SM}(G_1 \setminus \{J_k\}) \geq \mathcal{SM}(G_1) - f$, since $x_{G_1 \setminus \{J_k\}} \leq x_{G_1}$. By switching J_k to G_2 we obtain a solution at least as good as $\mathcal{S}$ and thus still optimal. □

G. DESCRIPTION OF THE FORMULAS

In this section, we describe in detail several formulas used in our algorithms.

- $\mathcal{SS}(G) = \Sigma_{i=1}^{n} (f - 2 \cdot d_i \cdot (p_j - p_i + \delta_G)) - f$ (Equation 11): recall that J_j is the constituent job of group G, i.e., it has the largest intermediate data size (d_j), incurring the highest number of passes (p_j). δ_G indicates if the final number of sorting passes increases with respect to p_j. Each job J_i in G introduces savings of $(f - 2 \cdot d_i \cdot (p_j - p_i + \delta_G))$, where f accounts for the input scan, and $-2 \cdot d_i \cdot (p_j - p_i + \delta_G)$ is subtracted to account for additional cost of sorting – if the original number of passes p_i of job J_i is lower than the final number of passes $p_j + \delta_G$. Overall savings $\mathcal{SS}(G)$ for group G is a sum over all jobs minus f for performing a single input scan when executing G.
- $gain(i, j) = f - 2 \cdot d_i \cdot (p_j - p_i + \delta_j)$ (Equation 12): represents savings introduced by job J_i merged with a group where J_j is a constituent job in a sharing scans only scenario. Originally, job J_i sorts its intermediate data in p_i passes, and job J_j in p_j passes. Here, δ_j is only dependent on J_j, and is either 0 or 1. $gain(i, j)$ quantifies savings incurred by one input scan (f) minus the cost of additional sorting of J_i's map output – if the original number of passes p_i is lower than $p_j + \delta_j$.
- $GAIN(t, u) = \sum_{t \leq i \leq u} gain(i, u)$ (Section 4.1.3): quantifies savings of merging a sequence of jobs $J_t, \ldots, J_u$ into a single group, without accounting the single input scan that needs to be performed for this group. $GS(t, u) = GAIN(t, u) - f$ accounts for one additional input scan per group.
- $c(l) = max_{1 \leq i \leq l}\{c(i-1) + GS(i, l)\}$ (Equation 13): represents the maximal savings over all possible groupings of a sequence of jobs $J_1, \ldots, J_l$. To obtain it in our dynamic program, we need to try all possible is ranging from 1 to l, and pick the one that yields the highest $c(l)$.
- $gain(i, j) = f - (g(\gamma - 1)d_i + 2 \cdot d_i \cdot (\gamma(p_j + \delta_j) - p_i))$ (Equation 16): quantifies savings introduced by job J_i merged with a group where J_j is a constituent job in a sharing map output scenario, where we assume that $(1 - \gamma)$ part of J_i's map output is shared with J_j. Hence, merging J_j with the group introduces savings on scanning the input f, plus the savings on copying $(1 - \gamma)$ part of the J_i's map output, minus the cost of additional sorting (which can be negative in this case). $-g(\gamma - 1)d_i$ quantifies the savings on copying the intermediate data. $2 \cdot d_i \cdot (\gamma(p_j + \delta_j) - p_i)$ is the difference of sorting cost in the shared scenario, where we need to sort only the γ part of J_i's map output in $p_j + \delta_j$ passes, and the non-shared scenario, where we sort the entire map output of J_i in p_i passes.

H. EXAMPLES OF TAGGING SCHEMES

There are two cases for which `MRShare` automatically combines the processing logic of two jobs: (i) If we know that the parsing of the input is identical then scan sharing is applied - the map output tuples are tagged as shown in Example 5. (ii) If, in addition, the two map functions produce identical tuples from the same input tuple, then we also share the common map output - Example 6 depicts how the map output tuples are tagged in this case. In both cases the original map functions are treated as black boxes and they are combined by the meta-map wrapper function.

In both cases, we do not consider separating the map output after the map stage. This would require substantial changes in the MapReduce architecture and counter our goal of developing a framework that minimally modifies the existing MapReduce architecture. Other side-effects of such an approach would be: more files would need to be managed which means more overhead and errors; running multiple reduce stages would cause network and resource contention.

EXAMPLE 5 (TAGGING FOR SHARING SCANS). *Table 3 is an example of tagging for sharing scans. The appropriate reduce function is indicated for each tuple, based on the tag() field. Each tuple within a group will be processed by exactly one of the original reduce functions. Since tuples are sorted according to the (key + tag()), the reduce functions will be executed sequentially. First, all tuples belonging to J_3 will be processed, then J_2, etc. At any point in time, only the state of one reduce function must be maintained. In this example, tag() is one byte long. Also, B_7 is always set to 1, meaning that a tuple belongs to exactly one job. $B_6 \ldots B_0$ determines the job, in which the tuple belongs.*

EXAMPLE 6 (TAGGING FOR SHARING MAP OUTPUT). *Table 4 describes tuples produced by n mapping pipelines*

Table 3: Sharing-scans tagging

key	B_7	$B_6 \dots B_0$	value	reduce function
key	1	0000100	v_1	$reduce_3$
key	1	0000100	v_2	$reduce_3$
key	1	0000100	v_3	$reduce_3$
key	1	0000010	v_4	$reduce_2$
key	1	0000010	v_5	$reduce_2$
key	1	0000001	v_6	$reduce_1$

from one input tuple, sorted according to key. The tag() field is byte long ($B7, \dots, B0$), as before.

Table 4: Map output of n pipelines

job	key	B_7	$B_6 \dots B_0$	value
1	1	1	0000001	v_1
3	1	1	0000100	v_1
2	1	1	0000010	v_1
4	2	1	0001000	v_2
6	2	1	0100000	v_2
5	3	1	0010000	v_3

Going back to our example, the output produced by the mapping pipelines can be shared as shown in Table 5. Observe that the MSB of tag() is set according to the number of originating pipelines.

Table 5: Merged map output

job	key	B_7	$B_6 \dots B_0$	value
1,2,3	1	0	0000111	v_1
4,6	2	0	0101000	v_2
5	3	1	0010000	v_3

Table 6 depicts some groups' examples, processed at the reduce side. The last column indicates the reduce functions to which each tuple needs to be pushed. We remark that we need to maintain the states of multiple reduce functions. However, since the group is sorted also on tag(), we are able to finalize some reduce functions as the tuples are processed.

I. EXPERIMENTAL EVALUATION

I.1 Experimental Setting

We ran all experiments on a cluster of 40 virtual machines, using Amazon EC2 [1], unless stated otherwise. We used the small size instances (1 virtual core, 1.7 GB RAM, 160GB of disk space). All settings for Hadoop were set to defaults. Our real-world 30GB text dataset, consists of blog posts [2]. We utilized approximately 8000 machine hours to evaluate our framework. We ran each experiment three times and averaged the results.

We ran *wordcount* jobs, which were modified to count only words containing given regular expressions (aka *grep-wordcount*). Hence, we were able to run jobs with various intermediate data sizes, depending on the selectivity of the regular expression. We remark that *wordcount* is a commonly used benchmark for MapReduce systems. However, we cannot use it in our experiments because we cannot control the map-output sizes. Another reason for choosing *grep-wordcount* is that it is a generic MapReduce job. The map stage filters the input (by the given regular expression), while the reduce stage performs aggregation. In other words, any group-by-aggregate job in MapReduce is similar in its structure to *grep-wordcount*. We clarify that we produced random jobs with various intermediate data sizes. In real-world settings, however, the information about approximate intermediate data sizes would have been obtained either from historical data, or by sampling the input, or by syntactical analysis of the jobs.

Table 6: Processing a group at the reduce side

key	B_7	$B_6 \dots B_0$	value	reduce function
k	1	0100000	v_1	$reduce_6$
k	1	0010000	v_2	$reduce_5$
k	0	0001001	v_3	$reduce_1$; $reduce_4$
k	0	0000111	v_4	$reduce_1$; $reduce_2$; $reduce_3$

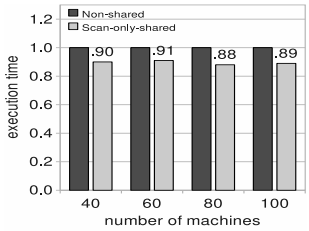

Figure 4: MRShare scale independence.

I.2 System Dependent Parameters

The first step of our evaluation was to measure the system dependent parameters, f and g, by running multiple *grep-wordcount* jobs. Our experiments revealed that the costs of reading from the DFS and the cost of reading/writing locally during sorting, were comparable, hence we set $f = 1.00$. This is not surprising, since Hadoop favors reading blocks of data from the DFS, which are placed locally. Hence, the vast majority of data scans are local.

The cost of copying the intermediate data of jobs over the network was approximately $g = 2.3$ times the cost of reading/writing locally. This is expected, since copying intermediate data between map and reduce stages involves reading the data on one machine, copying over the network, and writing locally at the destination machine.

We remark that our experiments revealed that the cost of scanning the input is rarely dominant, unless the map output sizes are very small. For example, when we ran a series of random *grep-wordcount* jobs, the average cost of scanning was approximately 12% of the total execution time of a job. Thus, even if there is no scanning at all, the overall savings from sharing scans cannot exceed this threshold.

I.3 MRShare Scale Independence

We demonstrate the scalability of `MRShare` with respect to the size of the MapReduce cluster. In particular, we show that the relative savings, using `MultiSplitJobs` for sharing scans, do not depend on the size of the cluster. We ran queries G4 with random map-output-ratio (see Table 2), and measured the resulting savings for 4 cluster sizes, 40,60,80, and 100. Figure 4 illustrates our results. In each case, the left bar represents the normalized execution time, when no sharing occurs. The relative saving from sharing scans were approximately the same in each case, independent of the size of the cluster. Indeed, our cost model does not depend on the number of machines and this is confirmed by our results. The overall cost is only distributed among the machines in the cluster. However, the relative savings from sharing e.g., scans do not depend on the cluster's size.