Multi-Query Optimization in MapReduce Framework

Guoping Wang and Chee-Yong Chan
Department of Computer Science, School of Computing
National University of Singapore
{wangguoping, chancy}@comp.nus.edu.sg

ABSTRACT

MapReduce has recently emerged as a new paradigm for large-scale data analysis due to its high scalability, fine-grained fault tolerance and easy programming model. Since different jobs often share similar work (e.g., several jobs scan the same input file or produce the same map output), there are many opportunities to optimize the performance for a batch of jobs. In this paper, we propose two new techniques for multi-job optimization in the MapReduce framework. The first is a generalized grouping technique (which generalizes the recently proposed MRShare technique) that merges multiple jobs into a single job thereby enabling the merged jobs to share both the scan of the input file as well as the communication of the common map output. The second is a materialization technique that enables multiple jobs to share both the scan of the input file as well as the communication of the common map output via partial materialization of the map output of some jobs (in the map and/or reduce phase). Our second contribution is the proposal of a new optimization algorithm that given an input batch of jobs, produces an optimal plan by a judicious partitioning of the jobs into groups and an optimal assignment of the processing technique to each group. Our experimental results on Hadoop demonstrate that our new approach significantly outperforms the state-of-the-art technique, MRShare, by up to 107%.

1. INTRODUCTION

MapReduce has recently emerged as a new paradigm for large-scale data analysis and has been widely embraced by Amazon, Google, Facebook, Yahoo!, and many other companies. There are two key reasons for its popular adoption. First, the framework can scale to thousands of commodity machines in a fault-tolerant manner and thus is able to use more machines to support parallel computing. Second, the framework has a simple and yet expressive programming model through which users can parallelize their programs without concerning about issues like fault-tolerance and execution strategy.

To simplify the expression of MapReduce programs, some high level languages, such as Hive [16, 17], Pig [14, 5] and MRQL [4], have recently been proposed for the MapReduce framework. The declarative property of these languages also open up new opportunities for automatic optimization in the framework [12, 3, 11]. Since different jobs (specified in or translated from queries) often perform similar work (e.g., jobs scanning the same input file or producing some shared map output), there are many opportunities to exploit the shared processing among the jobs to optimize performance. As noted by several researchers [13, 12], it is useful to apply the ideas from multi-query optimization to optimize the processing of multiple jobs by avoiding redundant computation in the MapReduce framework.

The state-of-the-art work in this direction is MRShare [12] which has proposed two sharing techniques for a batch of jobs. The *share map input scan* technique aims to share the scan of the input file for jobs, while the *share map output* technique aims to reduce the communication cost for map output tuples by generating only one copy of each shared map output tuple. The key idea behind MRShare is a *grouping technique* to merge multiple jobs that can benefit from the sharing opportunities into a single job. Thus, compared to MRShare, the naive technique of processing each job independently would need to scan the same input file multiple times and generate multiple copies of the same map output tuple. However, MRShare incurs a higher sorting cost compared to the naive technique as sorting a larger map output produced by the merged job is more costly than multiple independent sortings of smaller map outputs produced by the unmerged jobs.

In this paper, we present a more comprehensive study of multi-job optimization techniques. Our first contribution is the proposal of two new job sharing techniques that expands the opportunities for multi-job optimizations. The first technique is a *generalized grouping technique (GGT)* that relaxes MRShare's requirement for sharing map output. The second technique is a *materialization technique (MT)* that partially materializes the map output of jobs (in the map and/or reduce phase) which provides another alternative means for jobs to share both map input scan and map output. Comparing with the naive technique, GGT incurs a higher sorting cost (similar to MRShare's grouping technique) while MT incurs an additional materialization cost. Thus, neither *GGT* nor *MT* is strictly more superior, as demonstrated also by our experimental results.

Given the expanded repertoire of three sharing techniques (i.e., the naive independent evaluation technique, GGT which subsumes MRShare's grouping technique, and MT), finding an optimal evaluation plan for an input batch of jobs becomes an even more challenging problem. Indeed, the optimization problem is already NP-hard when only the naive and grouping techniques are considered in MRShare [12]. Our second contribution is the proposal of a novel two-phase approach to solve this non-trivial optimization problem.

Our third contribution is a comprehensive performance evaluation of the multi-job optimization techniques using Hadoop. Our experimental results show that our proposed techniques are scalable for a large number of queries and significantly outperform MRShare's techniques by up to 107%.

The rest of the paper is organized as follows. Section 2 presents background information on MapReduce and introduces the assumptions and notations used in this paper. Section 3 presents several multi-job optimization techniques to share map input scan and map output; their cost models are presented in Section 4. Section 5 presents a novel two-phase algorithm to optimize the evaluation of a batch of jobs given the expanded repertoire of optimization techniques. Section 6 presents a performance evaluation of the presented techniques. Section 7 presents related work, and we conclude in Section 8.

2. BACKGROUND

In this section, we review the MapReduce framework and introduce the assumptions and notations used in the paper.

2.1 MapReduce Preliminaries

MapReduce is proposed by Google as a programming model for large-scale data processing [2]. It adopts a master/slave architecture where a master node manages and monitors map/reduce tasks and slave nodes process map/reduce tasks assigned by the master node, and uses a distributed file system (DFS) to manage the input and output files. The input files are partitioned into fix-sized splits when they are first loaded into the DFS. Each split is processed by a map task and thus the number of map tasks for a job is equal to the number of its input splits. Therefore, the number of map tasks for a job is determined by the input file size and split size. However, the number of reduce tasks for a job is a configurable parameter.

A job is specified by a pair of map and reduce functions, and its execution consists of a map phase and a reduce phase. In the map phase, each map task first parses its corresponding input split into a set of input key-value pairs. Then it applies the map function on each input key-value pair and produces a set of intermediate key-value pairs which are sorted and partitioned into r partitions, where r is the number of configured reduce tasks. Note that both the sorting and partitioning functions are customizable. An optional combine function can be applied on the intermediate map output to reduce its size and hence the communication cost to transfer the map output to the reducers[1]. In the reduce phase, each reduce task first gets its corresponding map output partitions from the map tasks and merges them. Then for each key, the reducer applies the reduce function on the values associated with that key and outputs a set of final key-value pairs.

[1]For presentation simplicity, we do not consider combine functions in this paper; however, our proposed techniques can be easily extended to operate in the presence of combine functions.

Id	Job	<Key,Value>
J_1	***select*** a, $sum(d)$ ***from*** T ***where*** $a \geq 10$ ***group by*** a	$<a, d>$
J_2	***select*** a, b, $sum(d)$ ***from*** T ***where*** $b \leq 20$ ***group by*** a, b	$<(a, b), d>$
J_3	***select*** a, b, c, $sum(d)$ ***from*** T ***where*** $c \leq 20$ ***group by*** a, b, c	$<(a, b, c), d>$
J_4	***select*** a, $sum(d)$ ***from*** T ***where*** $b \leq 20$ ***group by*** a	$<a, d>$
J_5	***select*** b, $sum(d)$ ***from*** T ***where*** $a \geq 20$ ***group by*** b	$<b, d>$
J_6	***select*** $*$ ***from*** T, R ***where*** $T.a = R.e$	T:$<a, T.*>$ R:$<e, R.*>$
J_7	***select*** $*$ ***from*** T, R ***where*** $T.a = R.e$ *and* $T.b = R.f$	T:$<(a, b), T.*>$ R:$<(e, f), R.*>$

Table 1: Running examples of MapReduce jobs.

2.2 Assumptions & Notations

We assume that the input queries are specified in some high-level language (e.g., [16, 17, 14, 5, 4]) which are then translated to MapReduce jobs. By specifying the input jobs via a high-level query language, it facilitates the identification of sharing opportunities among jobs (via their query schemas); and standard statistics-based techniques [9, 15, 20] could be used to estimate the sizes of their shared map outputs. This assumption is also adopted in several related work [12, 3, 11, 20].

In the rest of this paper, we will use the terms queries and jobs interchangeably. Table 1 shows seven jobs (J_1 to J_7) that we will be using as running examples throughout this paper.

For a job J_i, we use K_i to represent its map output key, A_i to represent the set of attributes in K_i, $|A_i|$ to represent the number of attributes in A_i, M_i to represent its map output, and R_i to represent its reduce output. For example, for J_2 in Table 1, $K_2 = (a, b)$, $A_2 = \{a, b\}$ and $|A_2| = 2$.

We use $K_i \preceq K_j$ to denote that K_i is a prefix of K_j, and $K_i \prec K_j$ to denote that K_i is a proper prefix of K_j (i.e., $K_i \neq K_j$). For example, $K_4 \prec K_2$ and $K_5 \not\prec K_2$.

Consider a map output M_i with schema (A_i, V_i) where A_i and V_i refers to the map output key and value attributes, respectively. Given a set of attributes $A \subseteq A_i$, we use M_i^A to denote the map output derived from M_i where its map output key attributes are projected onto A; i.e., $M_i^A = \pi_{A,V_i}(M_i)$. For example, $M_2^{\{a\}} = M_4$.

Consider two jobs J_i and J_j where $A_j \subseteq A_i$. We use $M_{i,j} \subseteq M_i$ to denote the subset of M_i such that $M_{i,j}^{A_j} = M_i^{A_j} \bigcap M_j$ represents the subset of M_j that can be derived from M_i. Furthermore, we use $M_i \biguplus M_j$ to represent the (key, value-list) representation of the map output $M_i \bigcap M_j$. For example, if $M_i \bigcap M_j = \{(k_1, v_1), (k_1, v_2), (k_2, v_3)\}$, then $M_i \biguplus M_j = \{(k_1, <v_1, v_2>), (k_2, <v_3>)\}$.

3. MULTI-JOB OPTIMIZATION

In this section, we discuss several multi-job optimization techniques. We first review the grouping technique in MRShare [12], which is the most relevant work to ours, and then present our proposed generalized grouping technique (GGT) and materialization technique (MT). For simplicity, we focus our presentation on two single-input jobs J_i and J_j on an input file F; the handling of multi-input jobs is discussed in Section 3.4. Figure 1 gives a pictorial comparison of the techniques to process two jobs J_i and J_j, where $K_j \preceq K_i$.

3.1 Grouping Technique

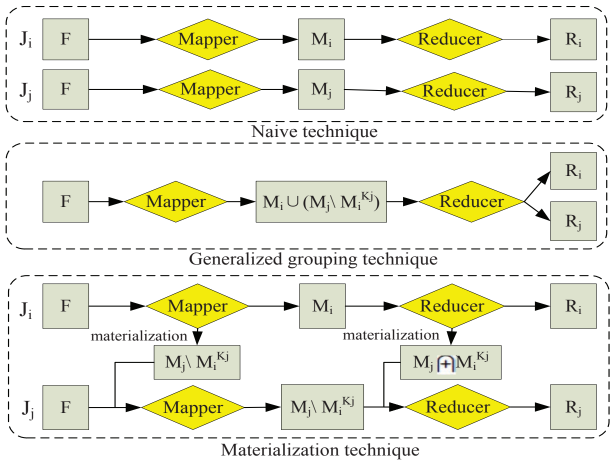


Figure 1: Multi-job optimization techniques

In this section, we review MRShare's grouping technique.

Sharing map input scan. For two jobs J_i and J_j to share their map input scan, the input files of J_i and J_j, the input key and value types of J_i and J_j, and the map output key and value types of J_i and J_j must be all the same. We can then combine J_i and J_j into a new job to share the scan of the map input for the two jobs. We now describe the map and reduce phases of the new job.

In the map phase, the common input file is scanned to generate the map outputs M_i for J_i and M_j for J_j. To distinguish the map outputs of the two jobs in the reduce phase, we use $tag(i)$ to tag the map output M_i and $tag(j)$ to tag the map output M_j. The tags are stored as part of the map output values; thus, each map output tuple is of the form (key,(tag,value)).

In the reduce phase, for each key and for each value associated with the key, if the tag of the value is $tag(i)$, we distribute the value to the reduce function of J_i; otherwise, we distribute the value to the reduce function of J_j. When all the values associated with a key have been examined, we generate the results for that key for the two jobs.

Sharing map output. For J_i and J_j to also share map output besides sharing map input scan, the two jobs must additionally satisfy the requirement that $K_i = K_j$. We can then combine J_i and J_j into a new job to share both their map input scan as well as any common map output (i.e., $M_i \bigcap M_j$). Sharing map output reduces the map output size and hence the sorting and communication cost. We now describe the map and reduce phases of the new job.

In the map phase, the values of the map output are tagged $tag(i)$, $tag(j)$, and $tag(ij)$, respectively, for tuples that belong to $M_i \setminus M_j$, $M_j \setminus M_i$, and $M_i \bigcap M_j$. In this way, tuples that belong to $M_i \bigcap M_j$ are produced only once with the tag $tag(ij)$.

In the reduce phase, for each key and for each value associated with the key, if the tag of the value is $tag(i)$, we distribute the value to the reduce function of J_i; if the tag of the value is $tag(j)$, we distribute the value to the reduce function of J_j; otherwise, we distribute the value to the reduce functions of both J_i and J_j. When all the values associated with a key have been examined, the reducer generates the results for that key for both jobs.

EXAMPLE 1. *Consider the two jobs J_1 and J_4. We can combine them into a new job to both share the scan of T as well as share the common map output for $a \geq 10 \wedge b \leq 20$. In the map phase, for each tuple t from T, if $t.a \geq 10 \wedge t.b > 20$, we produce the key-value pair $(t.a, (tag(1), t.d))$ indicating that it is produced by only J_1; if $t.a < 10 \wedge t.b \leq 20$, we produce the key-value pair $(t.a, (tag(4), t.d))$ indicating that it is produced by only J_4; if $t.a \geq 10 \wedge t.b \leq 20$, we produce the key-value pair $(t.a, (tag(14), t.d))$ indicating that it is produced by both J_1 and J_4; otherwise, we do not produce any map output for the tuple. In the reduce phase, for each key and for each value associated with the key, if the tag of the value is $tag(1)$, we aggregate the value for J_1; if the tag of the value is $tag(4)$, we aggregate the value for J_4; otherwise, we aggregate the value for both J_1 and J_4. When all the values associated with a key have been aggregated, we output the results for that key for J_1 and J_4.* □

3.2 Generalized Grouping Technique

In this section, we present a generalized grouping technique (GGT) that relaxes the requirement of MRShare's grouping technique (i.e., $K_i = K_j$) to enable the sharing of map output. To motivate our technique, consider the two jobs J_1 and J_2 in Table 1. Although $K_1 \neq K_2$, it is clear that the map output of J_2 for $a \geq 10$ could be used to derive the partial map output of J_1. We first present the basic ideas for processing two jobs and then discuss the generalization to handle more than two jobs.

Basic Ideas. To share the map output of two jobs J_i and J_j, GGT requires that $K_j \preceq K_i$ which is a weaker condition than MRShare's grouping technique (i.e., $K_i = K_j$). The jobs J_i and J_j are combined into a new job to enable the map output of J_i to be reused for J_j.

In the map phase of the new job, we generate the map output M_i for J_i and the partial map output $M_j \setminus M_i^{A_j}$ for J_j. The remaining map output of J_j (i.e., $M_{i,j}^{A_j}$) is not generated explicitly since they can be derived from M_i (i.e., $M_{i,j}$). By sharing the map output of J_i and J_j via $M_{i,j}$, we reduce the overall size of the map output. The values of the map output are tagged $tag(i)$, $tag(j)$, and $tag(ij)$, respectively, for tuples that belong to $M_i \setminus M_{i,j}$, $M_j \setminus M_i^{A_j}$, and $M_{i,j}$.

Note that in the MapReduce framework, the map output tuples for a job must all share the same output schema (i.e., same key and value types). While this requirement is satisfied by MRShare's grouping technique (i.e., $K_i = K_j$), the relaxed requirement (i.e., $K_j \preceq K_i$) of GGT may require us to additionally convert the map output of J_i and J_j (produced by our new job) to be of the same type. To achieve this, we use the simple approach of converting both the key and value components of the map output to string values if their types are different. Let us take the conversion of the key component for example. For the key component of a map output tuple, we represent it as a string value that is formed by concatenating the string representation of each of its key attributes separated by some special delimiter (e.g., ":"). For example, the string representations of the key components of J_2 and J_3 are of the form "a:b" and "a:b:c", respectively. This representation enables each key attribute value to be easily extracted from the string representation of the key component.

Since $K_j \preceq K_i$, the map output of the new job is partitioned on K_j and sorted on K_i. By partitioning on K_j, the map output tuples that have the same K_j values are distributed to and processed by the same reducer thereby enabling the reuse of the map output of J_i for J_j. The sorting on K_i is to facilitate the processing at the reducers (to be explained later); note that this sorting is well defined: for the map output tuples of J_j (whose key values do not contain all the values of K_i), the missing attribute values are treated as being converted to empty string values.

In the reduce phase of the new job, to compute the results of J_i, for each key of J_i, we apply the reduce function on the values associated with that key from tuples tagged $tag(i)$ or $tag(ij)$. To compute the results of J_j, for each key of J_j, besides the values associated with that key (from tuples tagged $tag(j)$), we also need to find the values of J_i that can be reused for J_j; i.e., tuples tagged $tag(ij)$ where the projection of its key on A_j is equal to the key of J_j. The reduce function of J_j is applied on all these values to produce the result for that key. Note that all the relevant tuples needed for the reduce function can be found very efficiently with a partial sequential scan of the map output (which is sorted on K_i).

Generalization. We now discuss how GGT can be generalized to handle more than two jobs.

Consider a batch of jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ that are sorted in non-ascending order of $|A_i|$. For each job $J_i \in \mathcal{J}$, let P_{J_i} denote all the jobs preceding J_i in $\mathcal{J}$ whose map output can be reused for J_i; i.e., $P_{J_i} = \{J_j \in \mathcal{J} \mid j < i, K_i \preceq K_j\}$. Furthermore, let $NM_i = M_i \setminus (\bigcup_{J_j \in P_{J_i}} M_j^{A_i})$ denote the map output of J_i that cannot be derived from the map output of any job in P_{J_i}. We refer to NM_i as the *non-derivable map output* of J_i in $\mathcal{J}$. We use $NM = \bigcup_{i=1}^{n} NM_i$ to denote the *non-derivable map output* for all the jobs in $\mathcal{J}$.

GGT combines the batch of jobs $\mathcal{J}$ into a single new job to share map input scan and map output. In the map phase of the new job, for each $J_i \in \mathcal{J}$, we produce and tag the map output NM_i; each tag here is of the form $tag(S)$, where $S \subseteq \{i, i+1, \cdots, n\}$. In the reduce phase of the new job, we apply the reduce functions on the appropriate values based on their tags to produce the results for the batch of jobs.

Unlike the grouping technique where each reduce function is applied on the values associated with one key, GGT may need to apply each reduce function on the values associated with multiple consecutive keys due to the different number of map output key attributes for the jobs. Therefore, in GGT, we have to determine when to apply the reduce functions and output the results for the jobs; the details of this are discussed elsewhere [18].

Example 2. *Consider the two jobs J_1 and J_2. As $K_1 \prec K_2$, GGT is applicable to enable both jobs to share map input scan and map output. In the map phase, for each tuple t from T, if $t.a \geq 10 \wedge t.b > 20$, we produce the key-value pair (t.a, (tag(1), t.d)) indicating that it is produced and consumed by only J_1; if $t.a < 10 \wedge t.b \leq 20$, we produce the key-value pair (t.a:t.b, (tag(2), t.d)) indicating that it is produced and consumed by only J_2; if $t.a \geq 10 \wedge t.b \leq 20$, we produce the key-value pair (t.a:t.b, (tag(12), t.d)) indicating that it is produced by J_2 and consumed by both J_1 and J_2; otherwise, we do not produce any map output for that tuple. We then partition the map output on a and sort the map output on a:b. In the reduce phase, we apply the reduce functions of J_1 and J_2 on the appropriate values to produce the results for J_1 and J_2.* □

3.3 Materialization Techniques

In this section, we present an alternative approach, termed *materialization techniques (MT)*, for enabling multiple jobs to share map input scan and map output. Given a batch of jobs, the main idea of MT is to process the jobs in a specific sequence such that the map outputs of some of the preceding jobs can be materialized and used by the succeeding jobs in the sequence. There are two basic materialization techniques, namely, *map output materialization* and *reduce input materialization*, to enable sharing of map input and map output, respectively. We first present the techniques for processing two jobs and then discuss the generalization to handle more than two jobs.

Map Output Materialization (MOM). Our first materialization technique, which enables jobs J_i and J_j to share the scan of the map input file, requires that the input files and input key and value types of J_i and J_j to be the same. Assume that J_i is to be processed before J_j.

In the map phase of J_i, we read the map input file F to compute both the map output M_i for J_i as well as the map output M_j for J_j. M_j is materialized to the distributed file system (DFS) to be used later for processing J_j. The reduce phase of J_i is processed as usual.

In the map phase of J_j, instead of reading the map input file F a second time, we read the materialized map output M_j from the DFS. The reduce phase of J_j is processed as usual.

This simple materialization technique is beneficial if the total cost of materializing and reading M_j is lower than the cost of reading the input file F.

Reduce Input Materialization (RIM). Our second materialization technique aims to enable jobs J_i and J_j to share map output. This technique requires that $K_j \preceq K_i$, J_i to be processed before J_j, and the map output of J_i and J_j to be partitioned on K_j. The key idea of this technique is to materialize the map output $M_i^{A_j} \Cap M_j$ in the reduce phase of J_i, to be used later by the reduce phase of J_j. In this way, the sorting and communication cost of the map output $M_i^{A_j} \bigcap M_j$ is eliminated when processing J_j.

The map phase of J_i is processed as usual: we scan the input file F to produce the map output M_i for J_i. To enable the reduce phase of J_i to materialize $M_i^{A_j} \Cap M_j$ later, the map output M_i is tagged as follows: tuples in $M_{i,j}$ are tagged using $tag(ij)$ while the remaining tuples (i.e., tuples in $M_i \setminus M_{i,j}$) are tagged using $tag(i)$.

In the reduce phase of J_i, for each key, we apply the reduce function of J_i on the values associated with the key to produce the results of J_i. At the same time, for values that are tagged $tag(ij)$, we derive and materialize the sorted map output $M_i^{A_j} \Cap M_j$ into the DFS so that the materialized output will be later used by the reduce phase of J_j. Note that an optional combine function can be applied to reduce the size of the materialized map output $M_i^{A_j} \Cap M_j$ and hence the materializing and reading costs.

In the map phase of J_j, we scan the input file F to generate the partial map output $M_j \setminus M_i^{A_j}$ for J_j. The remaining map output of J_j (i.e., $M_{i,j}^{A_j}$) is not generated explicitly since they have already been sorted and materialized by J_i's reduce phase.

In the reduce phase of J_j, we first read the materialized map output $M_j \mathbin{\cap\!\!\!\!\!+} M_i^{A_j}$ from DFS and merge them with the map output that are shuffled from the map phase. Then for each key, we apply the reduce function of J_j on the values associated with that key to produce the results of J_j.

Thus, RIM reduces the sorting and communication costs for J_j by reducing the size of J_j's map output, but incurs an additional cost to materialize and read $M_i^{A_j} \mathbin{\cap\!\!\!\!\!+} M_j$.

Combining MOM & RIM. Both MOM and RIM can be applied together as follows. In the map phase of J_i, besides producing the map output M_i for J_i, we also generate the map output $M_j \setminus M_i^{A_j}$ for J_j. $M_j \setminus M_i^{A_j}$ is materialized into the DFS to be reused later for J_j. Then we process J_i as before.

In the map phase of J_j, instead of reading from the input file F, we read the materialized map output $M_j \setminus M_i^{A_j}$ from DFS and simply redirect the read tuples as the map output. Then we process J_j as before. The question of whether MOM and RIM should used together is decided in a cost-based manner depending on whether the total cost of materializing and reading $M_j \setminus M_i^{A_j}$ is lower than the cost of reading the input file F.

Generalization. Given a batch of jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ sorted in non-ascending order of $|A_i|$, MT processes the jobs sequentially based on this ordering since the map output of a preceding job can possibly be reused for a succeeding job.

When processing J_1, in the map phase, we first produce NM_1 for J_1 (which is simply M_1), and tag each tuple t accordingly depending on the subset of remaining jobs in $\mathcal{J}$ that t can be used to derive their map outputs. Then for each J_i $(1 < i \leq n)$, if the cost of materializing and reading NM_i is lower than the cost of reading the input file F, we produce, tag, and materialize NM_i for J_i. In the reduce phase, when applying the reduce function for J_1, for each J_i $(1 < i \leq n)$, based on the tags in the values, we materialize the map output $NM_1^{A_i} \mathbin{\cap\!\!\!\!\!+} M_i$ to be reused later for J_i.

When processing J_i $(1 < i \leq n)$, in the map phase, if NM_i has been materialized, we read NM_i and simply redirect the read tuples as the map output; otherwise, we read the input file F to produce and tag the map output NM_i for J_i. In the reduce phase, we first merge NM_i with the map output that are materialized by the previous jobs (i.e., $NM_j^{A_i} \mathbin{\cap\!\!\!\!\!+} M_i$ for each $j \in [1, i-1]$) and then process the reduce function of J_i. When processing the reduce function of J_i, for each J_j $(i < j \leq n)$, based on the tags in the values, we materialize the map output $NM_i^{A_j} \mathbin{\cap\!\!\!\!\!+} M_j$ to be reused later for J_j.

Example 3. *Consider the two jobs J_1 and J_2 again. As $K_1 \prec K_2$, MT is applicable to enable both jobs to share map input scan and map output. As the map output of J_2 can be reused for J_1, we process J_2 before J_1. In the map phase of J_2, for each tuple t from T, if $t.b \leq 20 \wedge t.a < 10$, we produce the key-value pair (t.a:t.b,(tag(2),t.d)); if $t.b \leq 20 \wedge t.a \geq 10$, we produce the key-value pair (t.a:t.b,(tag(12),t.d)); if $t.b > 20 \wedge t.a \geq 10$, we produce the key-value pair $(t.a, t.d)$ and materialize it into DFS to be reused later for J_1 to share map input scan; otherwise, we do not produce any map output for that tuple. In the reduce phase of J_2, for each key, we sum the values associated with the key to produce the results of J_2. At the same time, for each specific key $t.a_i$:$t.b_i$, for all the values $<v_1, \cdots, v_n>$ associated with the key and tagged by $tag(12)$, we materialize $(t.a_i, \sum_{i=1}^{n} v_i)$ into DFS to be reused later for J_1 to share map output. When processing J_1, in the map phase, we read the materialized map output and sort and partition them. In the reduce phase, we first read the materialized map output and merge them with the map output shuffled from the map phase. Then for each key, we sum the values associated with the key to produce the results of J_1.* □

3.4 Discussions

In this section, we compare the proposed techniques, discuss the choices for map output keys and show how our proposed techniques apply to multi-input jobs.

Comparison of techniques. Our GGT generalizes and subsumes MRShare's grouping technique. However, there is no clear-cut winner between GGT and MT. Since GGT merges a group of jobs into a single new job, it requires the map output key and value types of the group of jobs to be the same, which may require a type conversion overhead. Moreover, GGT also incurs a higher sorting cost due to the larger map output of the merged job. On the other hand, MT has the limitation that the jobs within a group must be executed sequentially, and MT also incurs the overhead of result materialization and subsequent reading of the materialized results.

Choices for map output keys. For both GGT and MT, the choice of the map output key (i.e., ordering of A_i that specifies the map output key K_i for a job J_i) is important as it affects the sharing opportunities among jobs. For example, consider the jobs J_1, J_2 and J_5 in Table 1. Observe that there are two alternative map output keys for J_2: if we choose K_2 to be (a,b), we can share map output for J_1 and J_2; otherwise, with $K_2 = (b, a)$, we can share map output for J_5 and J_2. Thus, to optimize the sharing benefits for a given batch of jobs, we need to determine the map output key for each job; we defer a discussion of this optimization to Section 5.

Handing multi-input jobs. Our proposed techniques can be easily extended to handle multi-input jobs as well. Consider the two jobs J_6 and J_7 in Table 1 which have the common input files T and R. For both T and R, the map output key of J_6 is a proper prefix of the map output key of J_7. Therefore, we can apply MT to share both the map input scan as well as map output for the two jobs. Furthermore, by converting the map output keys of the two jobs into the same type, MRShare's grouping technique can share the map input scan for the two jobs while our GGT can share both the map input scan and map output for the two jobs.

4. COST MODEL

In this section, we present a cost model to estimate the evaluation cost of a batch of jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ in the MapReduce framework using the proposed techniques. Similar to MRShare, we model only the disk and network I/O costs as these are the dominant cost components. However, our cost model can be extended to include the CPU cost as well. Table 2 shows the system parameters used in our model, where the disk and network I/O costs are in units of seconds to process a page.

We assume the jobs in $\mathcal{J}$ are sorted in non-ascending order of $|A_i|$ and each $J_i \in \mathcal{J}$ is processed as m map tasks and

Table 2: System parameters

Parameter	Meaning
C_{lr}	cost of reading a page from local disk
C_{lw}	cost of writing a page to local disk
C_l	sum of C_{lr} and C_{lw}
C_{dr}	cost of reading a page from DFS
C_{dw}	cost of writing a page to DFS
C_d	sum of C_{dr} and C_{dw}
C_t	network I/O cost of a page transfer
D	merge order for external sorting
B_m	buffer size for external sorting at mapper nodes
B_r	buffer size for external sorting at reducer nodes

r reduce tasks on the input file F. We use $|R|$ to denote the size of R in terms of number of pages, where R can be an input file or map/reduce output of some job. For a map output M_i, we use $p^m_{M_i} = \lceil \log_D \lceil \frac{|M_i|}{mB_m} \rceil \rceil$ to denote the number of sorting passes of its map tasks where $\frac{|M_i|}{m}$ denote the average size of a map task, $p^r_{M_i} = \lceil \log_D \lceil \frac{|M_i|}{rB_r} \rceil \rceil - 1$ to denote the number of sorting passes of its reduce tasks where $\frac{|M_i|}{r}$ denote the average size of a reduce task [2], and p_{M_i} to denote the sum of $p^m_{M_i}$ and $p^r_{M_i}$.

4.1 A Cost Model for MapReduce

Given a job J_i, its total cost (denoted as C_{j_i}) consists of its map and reduce costs (denoted as C_{M_i} and C_{R_i} respectively). The map cost is given by:

$$C_{M_i} = C_{dr}|F| + C_{lw}|M_i| + C_l p^m_{M_i}|M_i| \tag{1}$$

where $C_{dr}|F|$ denote the cost to read the input file, $C_{lw}|M_i|$ denote the cost to write the initial runs of the map output, and $C_l|M_i|p^m_{M_i}$ denote the cost to sort the initial runs.

The reduce cost is given by:

$$C_{R_i} = C_t|M_i| + C_l p^r_{M_i}|M_i| + C_{lr}|M_i| \tag{2}$$

where $C_t|M_i|$ denote the transfer cost of the map output, $C_l|M_i|p^r_{M_i}$ denote the sorting cost of the map output, and $C_{lr}|M_i|$ denote the reading cost for the final merge pass. We do not include the cost to write the job results since this cost is common for all the proposed techniques.

Therefore, the total cost can be expressed as follows:

$$C_{R_i} = C_{dr}|F| + (C_t + C_l + C_l p_{M_i})|M_i| \tag{3}$$

Our cost model for Hadoop has one major difference from MRShare's cost model. In MRShare's model, the number of initial runs for sorting in the reduce phase is assumed to be equal to the number of map tasks (i.e., m). Based on this assumption, using the grouping technique does not increase the sorting cost in the reduce phase. However, in practice, Hadoop's reduce phase actually merges the transferred map output in main memory based on B_r to build initial runs which implies that using the grouping technique could increase the sorting cost in the reduce phase. Our cost model does not have this simplifying assumption and it is therefore more accurate than MRShare's model. In our performance evaluation, we apply our more accurate cost model to MRShare's *GT* technique as well so that all the techniques are compared based on the same cost model.

[2]The final merge pass optimization is enabled for sorting in Hadoop's reduce phase.

4.2 Costs for the Proposed Techniques

In this section, we use the above cost model to estimate the costs for the naive technique and our proposed GGT (which subsumes MRShare's GT technique) as well as MT techniques.

Naive technique: The naive technique processes each job independently. Thus, the cost of the naive technique is simply the sum of the cost of each job which is given by:

$$C_A = nC_{dr}|F| + (C_t + C_l)\sum_{i=1}^{n}|M_i| + C_l\sum_{i=1}^{n}p_{M_i}|M_i| \tag{4}$$

Generalized grouping technique: GGT combines the batch of jobs $\mathcal{J}$ into a single new job whose map output is denoted as $NM = \bigcup_{i=1}^{n} NM_i$. Thus, the cost of GGT is given by:

$$C_G = C_{dr}|F| + (C_t + C_l + C_l p_{NM})|NM| \tag{5}$$

Materialization technique: MT processes the jobs in $\mathcal{J}$ sequentially in non-ascending order of $|A_i|$ and materialize and reuse the map output as we have described in Section 3.3. Thus the cost of MT is given by:

$$\begin{aligned} C_M = C_{dr}|F| + \sum_{i=2}^{n}\min\{C_{dr}|F|, C_d|NM_j|\} + (C_t + C_l) \\ |NM| + C_l\sum_{i=1}^{n}p_{NM_i}|NM_i| + C_d\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}|NM_i^{A_j} \mathbin{\cap\!\!\!\!+} M_j| \end{aligned} \tag{6}$$

Note that $\sum_{i=2}^{n}\min\{C_{dr}|F|, C_d|NM_j|\}$ denote the materialization and reading cost in the map phase, and $C_d\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}|NM_i^{A_j} \mathbin{\cap\!\!\!\!+} M_j|$ denote the materialization and reading cost in the reduce phase.

5. OPTIMIZATION ALGORITHMS

In this section, we discuss how to find an optimal evaluation plan for a batch of jobs $\mathcal{J} = (J_1, J_2, \cdots, J_n)$.

An evaluation plan for $\mathcal{J}$ specifies the following: (1) the map output key K_i for each job $J_i \in \mathcal{J}$; (2) a partitioning of the jobs in $\mathcal{J}$ into some number of disjoint groups, $G_1, \cdots, G_k$, where $k \geq 1$ and $\mathcal{J} = G_1 \cup \cdots \cup G_k$; and (3) a processing technique T_i for evaluating the jobs in each group G_i. Since MRShare's grouping technique is subsumed by GGT, and the naive evaluation technique is equivalent to partitioning $\mathcal{J}$ into n groups each of which consists of a single job that is processed by GGT, we can simply consider only GGT or MT for each T_i.

Let Cost(G_i, T_i) denote the the cost of evaluating the group of jobs $G_i \subseteq \mathcal{J}$ with technique $T_i \in \{GGT, MT\}$. The estimation of Cost(G_i, T_i) has already been discussed in Section 4.

The optimization problem is to find an evaluation plan for $\mathcal{J}$ such that the total evaluation cost $\sum_{i=1}^{k}$ Cost(G_i, T_i) is minimized. A simpler version of this optimization problem was studied in MRShare and shown to be NP-hard. The problem is simpler in MRShare for two reasons: first, MRShare considers only the naive and grouping techniques; and second, MRShare does not have to consider the selection of the map output keys as this does not affect the sharing opportunities for the grouping technique. As a result, the heuristic approach in MRShare can not be extended for our more complex optimization problem.

To cope with the complexity of the problem, we present a two-phase approach to optimize the evaluation plan. In the first phase, we choose the map output key for each job to maximize the sharing opportunities among the batch of jobs. In the second phase, we partition the batch of jobs into groups and choose the processing technique for each group to minimize the total evaluation cost.

5.1 Map Output Key Ordering Algorithm

In this section, we discuss how to choose the map output key for each job (i.e., determine the ordering of the key attributes) to maximize the sharing opportunities for a batch of jobs. To quantify the sharing opportunities for a batch of jobs $\mathcal{J}$, we use the notion of the *non-derivable map output* for $\mathcal{J}$, denoted by NM, that was defined in Section 3.2. Since a smaller size of NM represents a larger amount of sharing among the jobs in $\mathcal{J}$, to maximize the sharing among the jobs in $\mathcal{J}$, the map output key for each job is chosen to minimize the size of NM.

A naive solution to optimize this problem is to enumerate all the combinations of map output keys for the jobs and choose the combination that minimizes the size of NM. However, the time complexity of this brute-force solution is $O(|A_1|!|A_2|! \cdots |A_n|!)$ which is infeasible for large number of jobs[3]. In this paper, we propose a greedy heuristic to optimize the map output key for each job.

Our greedy algorithm determines the ordering of the map output key attributes for each job J_i progressively by maintaining a list of sets of attributes, referred to as the ordering list (denoted by OL_i), to represent the ordering relationship for the map output key attributes of J_i. The attributes within a set are unordered, and the attributes in a set S are ordered before the attributes in another set S' if S appears before S' in the list. We use $|OL_i|$ to denote the number of sets in OL_i. For example, in the ordering list $<\{a,b,c\},\{d\}>$, the attributes in $\{a,b,c\}$ are unordered and they precede the attribute d. Furthermore, given two jobs J_i and J_j, we use $OL_i \preceq OL_j$ to represent that OL_i is a prefix of OL_j, i.e., for each $i \in [1, |OL_i|]$, the i^{th} sets in OL_i and OL_j are the same. For example, $<\{a,b\},\{c\}> \preceq <\{a,b\},\{c\},\{d\}>$.

Besides maintaining OL_i for each job J_i, our approach also maintains a *reuse set*, denoted by RS_i, for each job J_i. The purpose of RS_i is to keep track of the all jobs that can be reused for computing the map output of J_i.

Initially, as we have not chosen any jobs to share map output, the size of NM is simply the sum of each job's map output size. Furthermore, for each $J_i \in \mathcal{J}$, we initialize OL_i to be a list with a single set containing all the attributes in A_i and initialize RS_i to be empty. We then construct a weighted, undirected graph $G = (V, E)$ to represent all the potential sharing opportunities in $\mathcal{J}$ as follows. Each $J_i \in \mathcal{J}$ is represented by a vertex in V. An edge $e = (J_i, J_j)$ is in E if there exists two map output keys K_i and K_j, respectively, for J_i and J_j such that the map output of one job can be reused for the other job (i.e., $K_i \preceq K_j$ or $K_j \preceq K_i$). The weight of (J_i, J_j) is initialized to be the reused map output size for the two jobs (i.e., $|M_i^{A_j} \bigcap M_j|$ if $K_j \preceq K_i$ or $|M_j^{A_i} \bigcap M_i|$ if $K_i \preceq K_j$). All the edges in E are initialized to be unmarked.

[3]For example, we experimented with a batch of 25 randomly generated jobs each with a maximum of four attributes in its map output key, and the brute-force approach did not complete running in 12 hours.

Figure 2 shows an example of the initial graph constructed for a batch of five jobs $\{J_8, \cdots, J_{12}\}$. For ease of presentation, we use an interval of integers to represent the map output of a job where the size of an integer is 1. For example, the map output size of J_8 is 20 since it contains 20 integers in its map output $[1, 20]$. The initial graph contains the edge $e_1 = (J_8, J_{10})$ since there exists $K_{10} = (a, b, c)$ and $K_8 = (a, b, c, d)$ such that $K_{10} \preceq K_8$; moreover, the weight of e_1 is 16 since there are 16 values (i.e., [5,20]) in the map output of J_8 that can be reused for J_{10}.

For convenience, we use E_{J_i} to denote the set of all the unmarked edges incident on a node $J_i \in V$, and use N_{J_i} to denote the set of all the vertices that have a marked edge with a node $J_i \in V$.

Overall algorithm. Given an initial graph $G = (V, E)$, to reduce the size of NM, our greedy approach iteratively selects and marks one edge from the graph G until all the edges in G have been marked. At each iteration, it first chooses an unmarked edge with the maximum weight (i.e., the chosen edge represents the largest sharing opportunity and maximizes the reduction of the size of NM) to share and marks the edge. Then based on the chosen edge, it updates the ordering lists and reusing sets for some jobs. We refer to V_1 and V_2 as the set of jobs whose ordering lists and reuse sets, respectively, have been changed in the updating. Finally, for each $J_i \in V_1$, we check the edge validity for all the edges in E_{J_i} and remove the invalid edges (to be explained). For each $J_i \in V_2$, we update the weights for all the edges in E_{J_i} (to be explained). After the iterative process terminates, we derive the map output key for each job based on its ordering list.

In the following, we explain how the graph is updated in each iteration and how the map output key is derived at the end of the iterative process.

Updating ordering lists. Suppose that the edge $e = (J_i, J_j)$ is selected in an iteration. We first update the ordering lists for J_i and J_j. Then for each job $J_k \in \{J_i, J_j\}$, if the ordering list of J_k has changed, we also update the ordering lists for the jobs in N_{J_k} and recursively propagate the updating for the jobs in N_{J_k} whose ordering lists have changed until all the jobs have been examined or there is no more job whose ordering list has changed.

Given an edge $e = (J_i, J_j)$, the main idea to update OL_i and OL_j is to ensure that after the updating, one ordering list is a prefix of the other ordering list (i.e., $OL_i \preceq OL_j$ or $OL_j \preceq OL_i$). For example, the first iteration chooses $e_1 = (J_8, J_{10})$ to share since the weight of e_1 is the highest, and since $OL_8 = <\{a,b,c,d\}>$ and $OL_{10} = <\{a,b,c\}>$, OL_8 is updated to $<\{a,b,c\},\{d\}>$ to ensure that $OL_{10} \preceq OL_8$. Therefore, to update OL_i and OL_j, we iterate through the sets in OL_i and OL_j and accordingly decompose the corresponding sets to maintain the prefix relationship between the two lists. The time complexity for this updating is $O(m)$, where m is the maximum number of map output key attributes in a job. Since m is usually very small, we assume this checking can be done in $O(1)$ time.

For example, in Figure 2, the first iteration chooses the edge $e_1 = (J_8, J_{10})$. Then OL_{10} and OL_8 are updated as follows: OL_{10} does not change and OL_8 becomes $<\{a,b,c\},\{d\}>$. The second iteration chooses the edge $e_6 = (J_{10}, J_{12})$, and OL_{12} and OL_{10} are updated as follows: OL_{12} does not change and OL_{10} becomes $<\{a,b\},\{c\}>$ which triggers the

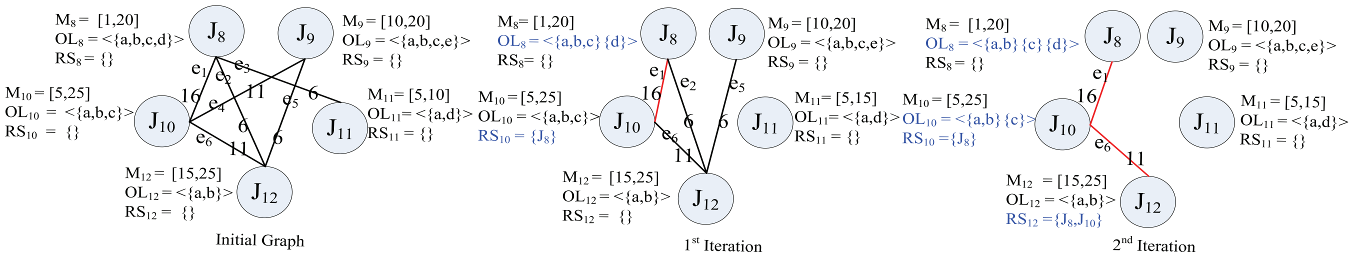


Figure 2: Example to illustrate key ordering algorithm.

updating for OL_8 since J_8 has a marked edge with J_{10}. Then we update OL_8 to be $<\{a,b\},\{c\},\{d\}>$.

Updating reuse sets. The updating of reuse sets is also done recursively similar to the updating of ordering lists. Therefore, we focus on explaining the updating of reuse sets for two jobs.

Given an edge $e = (J_i, J_j)$, the main idea to update RS_i and RS_j is as follows: if $A_i \subset A_j$, we update RS_i by adding the jobs in $RS_j \cup \{J_j\}$ into the set RS_i since all the jobs in $RS_j \cup \{J_j\}$ can be reused for J_i. Similarly, if $A_j \subset A_i$, we update RS_j by adding the jobs in $RS_i \cup \{J_i\}$ into the set RS_j since all the jobs in $RS_i \cup \{J_i\}$ can be reused for J_j. Otherwise, we have $A_i = A_j$, and we update both RS_i and RS_j by assuming that the map output of J_j will be reused for J_i as follows. Let S denote a copy of RS_i. We update RS_i by adding the jobs in $RS_j \cup \{J_j\}$ into RS_i, and update RS_j by adding the jobs in S into RS_j. The time complexity of the updating is $O(1)$.

After updating the ordering lists and reuse sets as described above, we then use the updated information to update the graph G; this includes identifying invalid edges (to be defined) in G, and updating some edge weights.

Identifying invalid edges. For a job $J_i \in V_1$, since OL_i has changed, for each $e \in E_{J_i}$, we need to check whether e is still a valid edge. An unmarked edge $e = (J_i, J_j)$ in G is defined to be a *valid edge* if we can derive two map output keys K_i and K_j, respectively, for J_i and J_j from OL_i and OL_j such that $K_i \preceq K_j$ or $K_j \preceq K_i$ (i.e., we can share map output for the two jobs); otherwise, e is considered an *invalid edge* and is removed from G.

We can check whether an unmarked edge $e = (J_i, J_j)$ is a valid edge or not as follows. If we can derive two ordering lists OL'_i and OL'_j respectively from OL_i and OL_j such that they satisfy the prefix relationship (i.e., $OL'_i \preceq OL'_j$ or $OL'_j \preceq OL'_i$), then the edge is a valid edge; otherwise, the edge is an invalid edge and can be removed from G. This process is similar to the process of updating the ordering lists for two jobs, and the time complexity is also $O(1)$. For example, in Figure 2, after choosing e_1 to share in the first iteration, OL_8 becomes $<\{a,b,c\}\{d\}>$ which makes e_3 an invalid edge since OL_{11} is $<\{a,d\}>$.

Updating edge weights. For a job $J_i \in V_2$, since RS_i has changed, for each $e \in E_{J_i}$, we need to update the weight for e. If the updated weight is 0, we can simply remove the edge since sharing the edge will not reduce the size of NM.

Given an edge $e = (J_i, J_j)$, its weight is updated as follows. If $A_i \subset A_j$ (i.e., the map output of the jobs in RS_j can be reused for J_i), then the weight of e is updated to $|S_{i1}| - |S_{i2}|$, where $|S_{i1}|$ and $|S_{i2}|$ denote, respectively, the size of the map output that J_i needs to produce (i.e., the size of the map output of J_i that can not be reused from RS_i) before and after we share e. Note that both $|S_{i1}|$ and $|S_{i2}|$ are computed based on RS_i which has to be updated if we share e. Similarly, if $A_j \subset A_i$ (i.e., the map output of the jobs in RS_i can be reused for J_j), then the weight of the edge is updated to $|S_{j1}| - |S_{j2}|$, where $|S_{j1}|$ and $|S_{j2}|$ denote, respectively, the size of the map output that J_j needs to produce before and after we share e. Otherwise, we have $A_i = A_j$ (i.e., the map output of the jobs in RS_i and RS_j can be respectively reused for J_i and J_j), and the weight of the edge is updated to be $|S_{i1}| - |S_{i2}| + |S_{j1}| - |S_{j2}|$. The time complexity of this updating is $O(1)$.

For example, in Figure 2, after choosing e_1 to share in the first iteration, RS_{10} becomes $\{J_8\}$ which triggers the weight updating for the edges in $E_{J_{10}} = \{e_4, e_6\}$. Let us first consider e_4. After choosing e_1 to share, J_{10} only needs to produce the map output [21,25] (i.e., the remaining map output [5,20] can be reused from J_8) and the map output of J_9 can not be reused to reduce the map output [21,25] further. Therefore, the weight of e_4 decreases to 0 and e_4 is removed from the graph. Next, consider e_6. After choosing e_1 to share, both the map output of J_8 and J_{10} can be reused for J_{12}. However, the weight of e_6 remains the same since J_8 does not enable additional reusing for J_{12}.

Deriving map output key. Note that at the end of the iterative process, it is possible for some set in an ordering list OL_i to contain more than one attribute (i.e., the ordering of the key attributes for J_i is not yet a total ordering). To derive the map output key for J_i, we have to determine an ordering for the remaining partially ordered attributes. To correctly derive the ordering of key attributes for such scenarios, we make use of a default ordering for all the attributes. For example, in Figure 2, at the end of the iterative process (i.e., after we have chosen the edge e_6 to share), the ordering lists for the five jobs $J_8, \cdots, J_{12}$ all contain at least one set that have more than one attribute. Assuming that the default ordering for all the attributes is (a,b,c,d), then the map output keys for J_{12}, J_{10} and J_8 are, respectively, (a,b), (a,b,c), and (a,b,c,d), which captures all the sharing that our algorithm has chosen. Note that without using a default ordering, we could wrongly choose the map output key (b,a) for J_{12} and the map output key (a,b,c) for J_{10} which does not allow these two jobs to share their map output.

Time Complexity. The time complexity of the algorithm depends on the number of iterations. The time complexity for the i^{th} iteration is $O(|E_i|)$, where $|E_i|$ is the number of edges in the graph in this iteration. Therefore, the time complexity of the algorithm is $O(In^2)$ where I is the number of iterations and $O(n^2)$ is the maximum number of edges in the graph.

5.2 Partitioning Algorithm

In this section, we discuss the second phase of our approach; i.e., how to partition a batch of jobs into multiple groups and choose the processing technique for each group to minimize the overall evaluation cost. We use the notation (G_i, T_i) to denote that a group of jobs G_i is being processed by a technique T_i. Recall that since GGT subsumes MRShare's grouping technique, and the naive evaluation technique is equivalent to partitioning the batch of jobs into single-job groups each of which is processed by GGT, it is sufficient to consider only the GGT and MT processing techniques.

Our partitioning algorithm is based on the concept of *merging benefit* which is defined as follows. Consider two groups of jobs, (G_1, T_1) and (G_2, T_2), where $G_1 \cap G_2 = \emptyset$. We define the *merging benefit* from (G_1, T_1) and (G_2, T_2) to $(G_1 \cup G_2, T_3)$, where $T_3 \in \{GGT, MT\}$, as Cost(G_1, T_1) + Cost(G_2, T_2) - Cost$(G_1 \cup G_2, T_3)$.

Our partitioning algorithm is a greedy approach that iteratively selects a pair of groups of jobs to be merged based on their merging benefit. Initially, each job is treated as a single-job group processed by GGT (which is equivalent to the naive technique since the group has only one job). At each iteration, it merges the two groups that have the maximum positive merging benefit into a new group. The iterative process terminates when the maximum merging benefit is non-positive.

Note that the time complexity of the grouping algorithm is $O(n^2)$, where n is the number of jobs in the batch. In the first iteration, we compute the merging benefit for each pair of groups, and in each subsequent iteration, since there is only one new group produced in the previous iteration, we only need to compute the merging benefit for each group with the new group.

6. EXPERIMENTAL RESULTS

In this section, we present an experimental study to evaluate our proposed approach. Section 6.1 examines the performance of our approach, and Section 6.2 evaluates the effectiveness of our map output key ordering algorithm.

Algorithms. We compared six algorithms (denoted by NA, $MRGT$, GGT, MT, $GGTMT$, and GT) in our experiments. The two competing algorithms were NA, which denote the naive approach of evaluating each job independently, and $MRGT$, which denote MRShare's grouping technique. For $MRGT$, we experimented with two different implementation variants: the original variant [12], which uses only a single global tuning parameter $\gamma \in [0, 1]$ to quantify the sharing among all the jobs in a batch, and an enhanced variant which provides a more fine-grained and accurate approach to estimate job sharing using a tuning parameter $\gamma_{i,j}$ for each pair of jobs J_i and J_j. As our experimental results show that the enhanced variant strictly outperforms the original variant[4], we do not report results for the original variant and use $MRGT$ to denote the enhanced variant.

Our three main proposed algorithms include: GGT, which denote the generalized grouping technique; MT, which denote the materialization technique; and $GGTMT$, which denote the approach combining both GGT and MT. In addition, to demonstrate the effectiveness of our partitioning heuristic (Section 5.2), we also introduce a variant of $MRGT$, denoted by GT, which combines MRShare's grouping technique with our partitioning heuristic.

Datasets and Queries. We used synthetic datasets and queries for our experiments. The schema of the datasets was *Data (key char(8), dim1 char(20), dim2 char(20), dim3 char(20), dim4 char(20), range int, value int)* which consisted of one unique key attribute, four dimensional attributes used as group-by attributes, one range attribute used as the selection attribute, and one value attribute used as the aggregation attribute. Each of the four dimensional attributes had 500 distinct values and all the attribute values were uniformly distributed. The datasets were stored as text format and the size of each tuple was about 100 bytes. The default dataset had 1.7 billion tuples with a size of 160GB.

The synthetic queries were generated from the following query template: ***select*** *T, sum(value)* ***from*** *Data* ***where*** $a \le$ *range* $\le b$ ***group by*** T, where T is a randomly selected list of dimensional attributes, and a and b are randomly selected values such that $a \le b$. The default number of queries in a query batch was 20. Each batch of queries was run three times and we report their average running times.

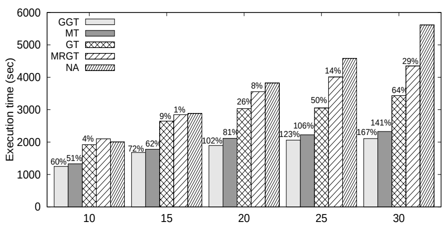
Experimental environment. Our experiments were performed using Hadoop 1.0.1 on a cluster of nodes that were interconnected with a 1Gbps switch. Each node was equipped with an Intel X3430 2.4GHz processor, 8GB memory, 2x500G SATA disks and running CentOS Linux 5.5. The default cluster size was 41 (with 1 master node and 40 slave nodes).

Hadoop configuration. The following Hadoop configuration was used for our experiments: (1) the heap size of JVM running was 1024MB; (2) the default split size of HDFS was 512MB; (3) the data replication factor of HDFS was 3; (4) the I/O buffer size was 128KB; (5) the memory for the map-side sort was 200MB; (6) the space ratio for the intermediate metadata was 0.4; (7) the maximum number of concurrent mappers and the maximum number of concurrent reducers for each node was both 2; (8) the number of reduce tasks was 240; (9) speculative execution was disabled[5]; (10) JVM reuse was enabled; and (11) the default FIFO scheduler was used which supports concurrent execution of jobs; note that for MT, while the jobs within a group were executed sequentially, jobs from different groups were executed concurrently.

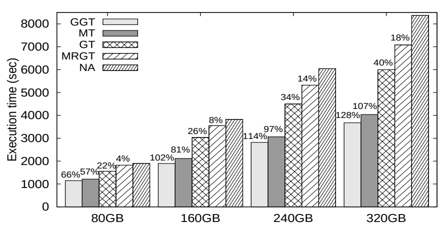
Cost model parameters. We ran some I/O benchmarks in the cluster to calibrate our cost model parameters [18] as follows: the cost ratio of local read/write is 1, the cost ratio for DFS read and write are, respectively, 1 and 2 (due

[4]For example, in the default setting, the running time for the enhanced variant was 3555s while that for the original variant was, 3820s, 3942s, 3931s, 3802s, 3885s, 3860s, 4385s, 4872s, and 4881s, respectively, for a γ value of 1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3 and $\{0.2, 0.1, 0\}$.

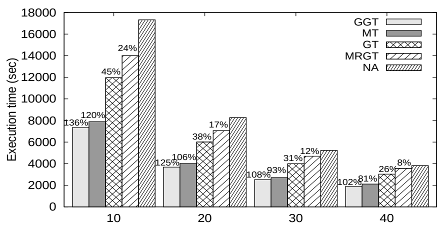
[5]Speculative execution is typically disabled in a busy cluster due to its negative impact on performance [19]. Indeed, in our preliminary experiments with speculative execution enabled, we observed that the performance of all the algorithms degraded. For example, in the default setting, the running times for both NA and $MRGT$ increased by 10% while that for GGT and MT increased by 6%. Thus, the winning margin of our algorithms increased slightly over NA and $MRGT$ with speculative execution enabled.



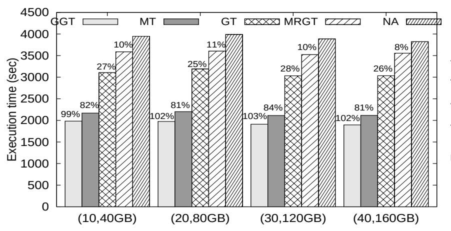
(a) Effect of number of queries



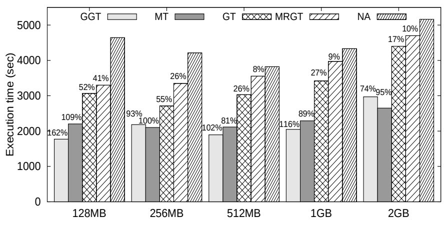
(b) Effect of data size



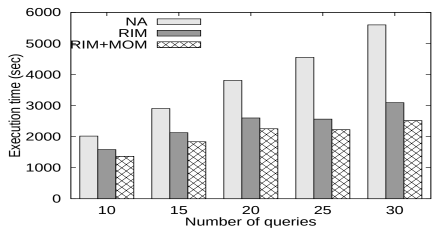
(c) Effect of cluster size



(d) Effect of data size and cluster size



(e) Effect of split size



(f) Analysis of MT

Figure 3: Effectiveness of optimization algorithms

to replication factor), and the cost ratio of network I/O is 1.4. Note that the setting of the same cost ratio for both local and DFS reads is reasonable due to the data locality property of the MapReduce framework.

Summary of results. First, our algorithms (GT, GGT, MT, $GGTMT$) significantly outperform NA by up to 167% and $MRGT$ by up to 107%. In particular, GT outperforms $MRGT$ by up to 31% demonstrating the effectiveness of our partitioning algorithm against MRShare's partitioning algorithm. Second, among our algorithms, GT performs the worst, and there is no clear winner between GGT and MT (as explained in Section 3): GGT outperforms MT by up to 24% for some cases and MT outperforms GGT by up to 12% for other cases. The overall winning approach is $GGTMT$ which outperforms the best of GGT and MT very slightly. Given this, to avoid cluttering the graphs, we do not explicitly show $GGMT$ in the graphs as its performance is approximated by the best of GGT and MT. Finally, our results show that the optimization overhead incurred by our approach is only a negligible fraction of the total processing time. For example, in the default setting (with 20 queries), the evaluation time for our best algorithm (i.e., GGT) took 1895 seconds compared to only 50ms for optimization; the optimization time increases to 1 second for 100 queries. Thus, the optimization overhead of our approach is negligible even if the queries do not have any sharing opportunities; more details are given elsewhere [18].

6.1 Performance Comparison

In this section, we evaluate the effectiveness of our optimization algorithms by varying four parameters, i.e., data size, split size, number of queries and cluster size. Figure 3 shows the experimental results with the the improvement factors (in %) of GGT, MT, GT and $MRGT$ over NA indicated. Due to space constraint, the number and size of partitioned groups for our algorithms are given in [18].

Effect of number of queries. Figure 3(a) compares the performance as the size of a query batch is increased. Observe that our algorithms significantly outperform NA and $MRGT$. For example, GGT outperforms NA by 105% on average and up to 167% when the number of queries is 30, and GGT outperforms $MRGT$ by 85% on average and up to 107% when the number of queries is 30. Furthermore, as the number of queries increases, the winning margin of our algorithms over NA also increases. This is expected as the sharing opportunities among queries also increase with the query batch size.

Effect of data size. Figure 3(b) examines the performance as a function of data size. Note that as we increase the data size, we also increase the number of reduce tasks. This is reasonable as the number of reduce tasks is usually proportional to the data size, as noted also in [1]. Therefore, we set the number of reduce tasks to be 120, 240, 360, and 480, respectively, for data size of 80GB, 160GB, 240GB, and 320GB.

Here again, our algorithms significantly outperform NA and $MRGT$. For example, GGT outperforms NA by 103% on average and up to 128% when the data size is 320GB, and GGT outperforms $MRGT$ by 82% on average and up to 93% when the data size is 320GB. Furthermore, as the data size increases, the running time for the algorithms also increases. In particular, the running time for NA increases much faster than for the other algorithms which therefore increases the winning margin of the other algorithms over NA. The reason behind this is that by partitioning the queries into groups, the non-NA algorithms are more scalable. For example, in the default setting (with a batch of 20 queries), NA needs to scan the input table 20 times while GGT, which has partitioned the batch of queries into two groups, only needs to scan the input table twice.

Effect of cluster size. Figure 3(c) compares the effect of number of slave nodes in the cluster. Here again, our algorithms significantly outperform NA and $MRGT$. For example, GGT outperforms NA by 118% on average and up to 136% when the number of nodes is 10, and GGT outperforms $MRGT$ by 89% on average and up to 92% when the number of nodes is 10 (the improvement factor of GGT over $MRGT$ does not show significant differences for all the node sizes). Furthermore, as the cluster size increases, the running time for all the algorithms decreases. In particular,

Table 3: Comparison of key ordering algorithms

Number of Queries	$\frac{\lvert NM_{Rka}\rvert - \lvert NM_{Pka}\rvert}{\lvert NM_{Pka}\rvert} \times 100\%$			$\frac{\lvert NM_{Pka}\rvert - \lvert NM_{Oka}\rvert}{\lvert NM_{Oka}\rvert} \times 100\%$		
	Min	Max	Avg	Min	Max	Avg
10	10%	26%	16%	0	8%	3%
15	11%	20%	18%	0	7%	2%
20	16%	25%	19%	1%	2%	1%
25	16%	20%	19%	–	–	–
30	14%	22%	19%	–	–	–

the running time for NA decreases much faster than for the other algorithms which therefore reduces the winning margin of the other algorithms over NA as cluster size increases. Thus, the performance improvement from the increased parallelism using a larger cluster benefits the non-optimized NA more than the already optimized non-NA algorithms.

Effect of both data size and cluster size. Besides studying the effect of the data size and cluster size parameters separately, we also conducted an additional experiment to examine the joint effect of both these parameters. In Figure 3(d), a cluster size of 10, 20, 30, and 40 slave nodes was used, respectively, for a data size of 40GB, 80GB, 120GB, and 160GB. As the results show, the performance of each algorithm does not vary very much as both the cluster size and data size jointly increase; this demonstrates the scalability of our algorithms wrt these two parameters.

Effect of split size. Figure 3(e) compares the effect of the split size. Here again, our algorithms significantly outperform NA and $MRGT$. For example, our best algorithm (i.e., GGT or MT) outperforms NA by 115% on average and up to 162% when the split size is 128MB, and our best algorithm (i.e., GGT or MT) outperforms $MRGT$ by 81% on average and up to 94% when the split size is 1GB. Observe that there is no clear winner between GGT and MT as explained in Section 3. For NA, we observe that its running time decreases with increasing split size until a certain threshold (e.g., 512MB for NA) after which its running times increases. This is because when the split size is too small, more map tasks will be launched for processing the job which incurs a higher startup cost; on the other hand, when the split size is too large, each map task will process more data which increases its sorting cost.

Analysis of MT. In this experiment, we analysis the relative effectiveness of the two techniques, MOM and RIM, that form MT. Figure 3(e) compares NA against two variants of MT: MT itself (denoted explicitly as RIM+MOM) and MT with only RIM technique (denoted as RIM). As the results show, RIM is more effective than MOM in reducing the running time. However, by further combining with MOM, we can improve the performance of RIM by 17% on average and up to 23% when the number of queries is 30.

6.2 Effectiveness of key ordering algorithm

In this section, we evaluate the effectiveness of our key ordering algorithm (denoted by *Pka*) by comparing against two extreme solutions: a brute-force algorithm that generates the optimal key ordering (denoted by *Oka*) and a naive heuristic that uses a random key ordering (denoted by *Rka*).

Recall from Section 5.1 that our map output key ordering algorithm is designed to maximize job sharing by minimizing the size of the non-derivable map output (denoted by NM) for the input batch of jobs. To assess its effectiveness, we compare two ratios, $\frac{|NM_{Rka}| - |NM_{Pka}|}{|NM_{Pka}|}$ and $\frac{|NM_{Pka}| - |NM_{Oka}|}{|NM_{Oka}|}$, where $|NM_x|$ denote the size of the non-derivable map output for an input batch of queries using algorithm x, $x \in \{Pka, Oka, Rka\}$. The first ratio measures the improvement factor of Pka over Rka, while the second ratio measures the improvement factor of Oka over Pka.

Table 3 compares these two ratios for various query sizes. For each query size, we randomly generate five batches of queries and report the average, minimum, and maximum values of the ratios. From Table 3, the $\frac{|NM_{Rka}| - |NM_{Pka}|}{|NM_{Pka}|}$ values show that our key ordering heuristic is indeed effective in minimizing $|NM|$ compared to the naive random ordering heuristic, while the $\frac{|NM_{Pka}| - |NM_{Oka}|}{|NM_{Oka}|}$ values show that our heuristic is almost as effective as the brute-force approach. Note that for query sizes 25 and 30, we were not able to compute values for $\frac{|NM_{Pka}| - |NM_{Oka}|}{|NM_{Oka}|}$ as Oka did not finish running in 12 hours. Indeed, as expected, Oka is not a scalable solution: for a query size of 20, Oka took about 3 hours to run compared to only 50ms taken by our heuristic Pka.

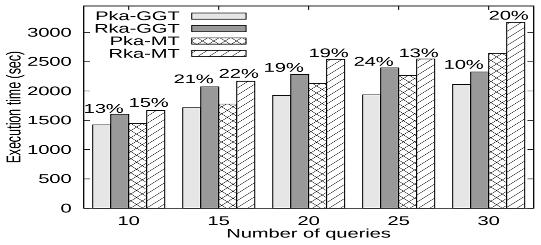


Figure 4: Comparison of Key Ordering Algorithms

To evaluate the effectiveness of the key ordering heuristics in terms of their impact on query evaluation time (excluding optimization time), we also compared their running times to evaluate query batches of difference size. In the following, we use the notation X-Y to denote the evaluation algorithm Y when used in combination with the key ordering heuristic X, where $Y \in \{GGT, MT\}$ and $X \in \{Pka, Rka, Oka\}$. Note that the evaluation algorithms NA, $MRGT$, and GT were excluded from the comparison as these algorithms do not require the key ordering step.

Figure 4(a) shows the running times for a representative query batch where its $\frac{|NM_{Rka}| - |NM_{Pka}|}{|NM_{Pka}|}$ ratio is ranked in the middle among the five batches. As the performance of *Oka*-Y is very close to that of *Pka*-Y (e.g., the former outperforms the latter by only 0.7% in the best case), we omit the results for *Oka*-Y in the graph. For each query size, Figure 4(a) also indicates two improvement factors (in %) which represent the performance improvement of *Pka*-Y over *Rka*-Y, $Y \in \{GGT, MT\}$. The results show that for both GGT and MT, Pka outperforms Rka by 17% on average.

7. RELATED WORK

We can broadly classify the optimization-related work in the MapReduce framework into three categories. A recent survey on data management in MapReduce can be found in [10].

Job optimization. There are several work [8, 6, 7] on optimizing general MapReduce jobs. The work in [8] proposes a system to automatically optimize MapReduce programs. The work in [6, 7] discusses the optimization opportunities presented by the large space of MapReduce configuration parameters and proposes a cost-based optimizer to choose the best configuration parameters for a job. Different from these work, our work focuses on optimizing multiple jobs specified in or translated from some high-level query language.

Query optimization. The proposal of high-level declarative query languages for MapReduce such as Hive [16, 17], Pig [14, 5] and MRQL [4], opens up new opportunities for query optimization in the framework. These work include optimization strategies for Pig [13], multi-way join optimization [1], optimization techniques for Hive [20], algebraic optimization for MRQL [4], and query optimization using materialized results [3]. All these work focus on query optimization techniques for a single query; in contrast, our work focuses on optimizing multiple jobs specified in or translated from some high-level query language.

The work in [3] presents a system ReStore to optimize query evaluation using materialized results. Given a space budget for storing materialized results, ReStore uses heuristics to both decide whether to materialize the complete map and/or reduce output of each job being processed as well as choose which previously materialized results to be evicted if the space budget is exceeded. Our work differs from ReStore in both the problem focus and the developed techniques. The results materialized by our *MT* technique for a given job could be the partial map output of another job; in contrast, ReStore materializes the complete output of the job being processed. Moreover, whereas the materialized output produced by ReStore might not be reused at all, this is not the case for our context as the query workload is known and our techniques only materialize output that will be reused.

Multi-query optimization. There are several work on multi-query optimization [12, 11]. The work that is the most closely related to ours is MRShare [12]. Compared with MRShare, our work is more comprehensive with additional optimization techniques (i.e., GGT and MT) which leads to a more complex optimization problem (e.g., the ordering of the map output key of each job becomes important) and a novel cost-based, two-phase approach to find optimal evaluation plans. In MRShare, an input batch of jobs is partitioned based on the following heuristic: the jobs are first sorted in non-descending order of their map output size, and a dynamic-programming based algorithm is used to find an optimal partitioning of the ordered jobs into disjoint consecutive groups. Thus, an optimal job partitioning where the jobs in a group are not consecutively ordered would not be produced by MRShare's heuristic. Note that our partitioning heuristic (with a time-complexity of $O(n^2)$) does not have this drawback and is more efficient than MRShare's partitioning heuristic ($O(n^3)$ time-complexity).

The work in [11] proposes a transformation-based optimizer for MapReduce workflows (translated from queries). The work considers two key optimization techniques: vertical (horizontal, resp.) packing techniques aim to optimize jobs with (without resp.) producer-consumer relationships; the horizontal packing techniques are based on MRShare's grouping technique. In contrast, our work does not specifically consider MapReduce workflow jobs that have explicit producer-consumer relationships; therefore, their proposed vertical packing techniques are not applicable for our work.

8. CONCLUSIONS

In this paper, we have presented a comprehensive study of multi-job optimization techniques for the MapReduce framework. We have proposed two new job sharing techniques and a novel two-phase optimization algorithm to optimize the evaluation of a batch of jobs given the expanded repertoire of optimization techniques. Our experimental results show that our proposed techniques outperform the state-of-the-art approach significantly by up to 107%.

Acknowledgements We would like to thank the reviewers for their constructive comments. This research is supported in part by NUS Grant R-252-000-512-112.

9. REFERENCES

[1] F. N. Afrati and J. D. Ullman. Optimizing joins in a mapreduce environment. In *EDBT*, 2010.

[2] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.

[3] I. Elghandour and A. Aboulnaga. Restore: Reusing results of mapreduce jobs. In *VLDB*, 2012.

[4] L. Fegaras, C. Li, and U. Gupta. An optimization framework for map-reduce queries. In *EDBT*, 2012.

[5] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. In *VLDB*, 2009.

[6] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. In *VLDB*, 2011.

[7] H. Herodotou, F. Dong, and S. Babu. Mapreduce programming and cost-based optimization? crossing this chasm with starfish. In *VLDB*, 2011.

[8] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. In *VLDB*, 2011.

[9] J. Jestes, K. Yi, and F. Li. Building wavelet histograms on large data in mapreduce. In *VLDB*, 2011.

[10] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using mapreduce. *ACM Computing Surveys*. To appear in 2014.

[11] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. In *VLDB*, 2012.

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

[13] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *ATC*, 2008.

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

[15] Y. Shi, X. Meng, F. Wang, and Y. Gan. Hedc: a histogram estimator for data in the cloud. In *CloudDb*, 2012.

[16] 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.

[17] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, and H. Liu. Hive-a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.

[18] G. Wang and C.-Y. Chan. Multi-query optimization in mapreduce framework. Technical Report http://www.comp.nus.edu.sg/~g0800170/techreport-MJQ.pdf, National University of Singapore, February 2013.

[19] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[20] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *SOCC*, 2011.