Sharing across Multiple MapReduce Jobs

TOMASZ NYKIEL, University of Toronto
MICHALIS POTAMIAS, Groupon
CHAITANYA MISHRA, Facebook, Inc.
GEORGE KOLLIOS, Boston University
NICK KOUDAS, University of Toronto

12

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 can be directly associated with monetary cost. 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 for utilizing the processing infrastructure. In this article we present a sharing framework tailored to MapReduce, namely, `MRShare`.

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. Given the query grouping, we merge jobs appropriately and submit them to MapReduce for processing. A key property of `MRShare` is that it is independent of the MapReduce implementation. Experiments with our prototype, built on top of Hadoop, demonstrate the overall effectiveness of our approach.

`MRShare` is primarily designed for handling I/O-intensive queries. However, with the development of high-level languages operating on top of MapReduce, user queries executed in this model become more complex and CPU intensive. Commonly, executed queries can be modeled as evaluating pipelines of CPU-expensive filters over the input stream. Examples of such filters include, but are not limited to, index probes, or certain types of joins. In this article we adapt some of the standard techniques for filter ordering used in relational and stream databases, propose their extensions, and implement them through `MRAdaptiveFilter`, an extension of `MRShare` for expensive filter ordering tailored to MapReduce, which allows one to handle both single- and batch-query execution modes. We present an experimental evaluation that demonstrates additional benefits of `MRAdaptiveFilter`, when executing CPU-intensive queries in `MRShare`.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Systems—*Parallel databases*

General Terms: Algorithms

Additional Key Words and Phrases: Sharing MapReduce jobs, systems, MapReduce, query processing

ACM Reference Format:
Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2014. Sharing across multiple MapReduce jobs. ACM Trans. Datab. Syst. 39, 2, Article 12 (May 2014), 46 pages.
DOI: http://dx.doi.org/10.1145/2560796

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

Authors' addresses: T. Nykiel (corresponding author), University of Toronto, Canada; email: tnykiel@cs.toronto.edu; M. Potamias, Groupon; C. Mishra, Facebook, Inc.; G. Kollios, Boston University, MA; N. Koudas, University of Toronto, Canada.

DOI: http://dx.doi.org/10.1145/2560796

the Elastic Compute Cloud (EC2) [Amazon 2006], 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. Similar services are provided by other companies as well.

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 [Olston et al. 2008a], it is imperative to apply ideas from *Multiple-Query Optimization* (MQO) [Finkelstein 1982; Sellis 1988; Zhou et al. 2007] to analytical queries in the cloud that have been studied in the context of relational databases, to reduce execution costs. For EC2 users, reducing the execution time is directly translated to monetary savings. It has been argued that, with an increasing need for large-scale data analysis, sharing work among multiple queries becomes ever-more imperative [Olston et al. 2008a; Gates et al. 2009]. Therefore, in this article we apply MQO ideas to MapReduce, the prevalent computational paradigm in the cloud.

MapReduce [Dean and Ghemawat 2004] serves as a platform for a considerable amount of massive data analysis. Besides, cloud computing has already turned MapReduce computation into commodity, for example, with EC2's Elastic MapReduce. The MapReduce paradigm has been widely adopted, mainly because it abstracts away parallelism concerns and is very scalable. In particular, programmers are allowed to specify intensive group-by-aggregate MapReduce *jobs*, which are executed by a large number of machines. The wide scale of MapReduce's adoption for specific analytical tasks can also be credited to Hadoop [2007], 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 [Abouzeid et al. 2009; Chaiken et al. 2008; Cohen et al. 2009; Friedman et al. 2009; Gates et al. 2009; Pike et al. 2005; Thusoo et al. 2009; Yu et al. 2008]. Beyond doubt, high-level language abstractions not only allow the programmers to write reusable code, but also enable the underlying system to perform automatic optimization [Olston et al. 2008a]. Along these lines, recently developed systems on top of Hadoop, such as Hive [Thusoo et al. 2009] and Pig [Gates et al. 2009], speak HiveQL [Thusoo et al. 2009], an SQL-like language, and Pig Latin [Olston et al. 2008b], 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 [Gates et al. 2009; Olston et al. 2008a]. Pig also implements a number of work sharing optimizations using MQO ideas. Hive has a similar feature. However, these optimizations are usually not automatic; the programmer needs to specify the details of sharing among multiple jobs [Gates et al. 2009].

MapReduce has been designed for long queries that fit well with batch-query processing, and in turn can benefit from sharing work. Indeed, our experiments in Amazon EC2 [Amazon 2006] indicate that substantial monetary savings are possible.

We present `MRShare` [Nykiel et al. 2010], a sharing framework that enables *automatic* and principled work sharing in MapReduce. We first concentrate on jobs for which the I/O cost is dominating. In particular, we describe a module that 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. Given a set of queries, `MRShare` derives nonoverlapping groups of queries, possibly containing a single query in a group, which are then executed

by *one* MapReduce job per group. We demonstrate that significant savings are possible using our Hadoop-based prototype.

Furthermore, we address optimizing single- and batch-mode CPU-intensive MapReduce queries, which becomes as important as optimizing the I/O-intensive ones. We evaluate how MapReduce can benefit from adaptive optimization for CPU-intensive queries; we extend standard adaptive filter reordering techniques for MapReduce systems and implement them through `MRAdaptiveFilter`, that is, an extension of `MRShare`, to further optimize the execution of the groups obtained from `MRShare`. `MRAdaptiveFilter` is a collection of lightweight techniques designed to serve both single and batched queries, which draws ideas from techniques proposed for relational and streaming databases. It is adaptive and exploits the feedback from the actual execution in real time, making local optimization decisions at MapReduce tasks. `MRShare`, with its extension `MRAdaptiveFilter`, can be easily integrated into existing systems that support high-level querying on top of MapReduce such as Hive [Thusoo et al. 2009] and Pig [Gates et al. 2009]. We summarize our contributions.

(1) We discuss sharing opportunities (Section 4), formally define the problem of work sharing in MapReduce, and propose `MRShare`, a platform-independent sharing framework.
(2) We propose a simple MapReduce cost model (Section 6) 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 7).
(4) We implement `MRShare` on top of Hadoop (Section 8) and present an extensive experimental analysis demonstrating the effectiveness of our techniques (Section 9) for I/O-intensive queries.
(5) We present an extension of `MRShare`, namely `MRAdaptiveFilter`, for executing groups of queries with expensive filters, for a single query (Section 10), and finally a generalization for a group of queries (Section 11).
(6) We demonstrate through extensive empirical validation that `MRAdaptiveFilter` can introduce additional substantial savings atop `MRShare`.

2. RELATED WORK

MapReduce. Since its original publication [Dean and Ghemawat 2004], MapReduce-style computation has become the norm for certain analytical tasks. The same authors report a vast increase of the number of MapReduce processes that ran in Google during the period 2004–2007 [Dean and Ghemawat 2008]. Besides Google, MapReduce has been used in specific applications [Panda et al. 2009; Chu et al. 2006] and furthermore, it is now offered as a cloud service from Amazon EC2 [Amazon 2006]. Moreover, MapReduce logic has been integrated as a core component in various projects towards novel, alternative data analysis systems [Abouzeid et al. 2009; Chaiken et al. 2008, Cohen et al. 2009; Gates et al. 2009; Friedman et al. 2009; Pike et al. 2005; Thusoo et al. 2009; Yu et al. 2008]. Hadoop [2007] is the most popular open-source implementation of MapReduce. The list of Hadoop users includes Amazon, Yahoo!, and a number of startups and various research labs, among many others. It serves as the platform for many projects [Abouzeid et al. 2009; Cohen et al. 2009; Gates et al. 2009; Thusoo et al. 2009], 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 [Olston et al. 2008b; Gates et al. 2009], Hive [Thusoo et al. 2009],

and Scope [Chaiken et al. 2008] 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. Thus, programmers can overcome the rigidness of the plain MapReduce paradigm to write more elaborate, legible, and reusable code. SQL/MapReduce [Friedman et al. 2009] integrates MapReduce functionality for UDF processing in Asterdata's nCluster, a shared-nothing parallel database. Greenplum's [Cohen et al. 2009] approach is similar. Such high-level languages allow one to incorporate more relational-style optimization approaches [Wu et al. 2011; Wang et al. 2011]. HadoopDB [Abouzeid et al. 2009] 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 [Pavlo et al. 2009] and tries to combine the advantages of both approaches, namely scalability and performance. It extends Hive [Thusoo et al. 2009] with the ability to use database logic in various parts of query execution, thus substantially decreasing the final processing cost. Hadoop++ [Dittrich et al. 2010] also applies techniques derived from the relational domain, including an indexing scheme and a join algorithm that helps to improve Hadoop performance without the need to modify it. Manimal [Jahani et al. 2011] applies static analysis techniques to optimize MapReduce queries. Actually, this system may be used in order to provide the input to our framework, if nothing else is possible. MapReduce-Merge extends MapReduce by adding a *merger* step which combines multiple reducers' outputs [Yang et al. 2007]. Jiang et al. [2010] present an nice in-depth empirical study of the performance of the MapReduce model. The aforementioned systems mostly concentrate on improving MapReduce queries in isolation. Our framework, `MRShare`, enables *work sharing* across multiple queries within MapReduce, and relies only on the core MapReduce functionality. Thus, it is complementary to all aforementioned systems. Starfish [Herodotou et al. 2011] is another recent system that is based on MapReduce and considers automatic optimization of MapReduce workloads at different granularities. Therefore, it can be considered as a more general and higher-level framework than ours and can use our framework as one of its components. SCALLA is another MapReduce-based system that tries to minimize I/O overhead and allow for single-pass algorithms [Li et al. 2012]. Finally, Circumflex [Wolf et al. 2012] is another recent system that concentrates on the scheduling part of the job execution.

Work sharing. Cooperative scans have been studies in traditional database systems [Qiao et al. 2008; Zukowski et al. 2007]. Among the MapReduce systems, Hive [Thusoo et al. 2009] 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 that the original jobs will eventually read. The original jobs are modified accordingly to read the newly created input. 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 [Gates et al. 2009] 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. Sharing scans of Pig jobs is studied by Wang et al. [2011]. However, no cost-based optimization takes place; only the greedy sharing in the batch mode. In this article, 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. [2008] 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. Their technique is based on the assumption that sharing scans is always beneficial. In our work, we show that sharing scans is not always beneficial. Work sharing has also been considered in multicore systems [Johnson et al. 2007] and finally in dynamic settings, where it can be performed at runtime [Harizopoulos et al. 2005; Candea et al. 2009].

Multiple-query optimization. Optimizing multiple queries and selecting materialized views can benefit from identifying common subexpressions in queries. In Multiple Query Optimization (MQO) [Chen and Dunham 1998; Finkelstein 1982; Sellis 1988; Shim et al. 1994; Park and Segev 1988; Zhou et al. 2007], the key idea is that, in batch-query execution, optimizing each query individually may lead to suboptimal query plans. In particular, queries may share subexpressions that could, in principle, be evaluated only once. There are two specific problems which are orthogonal to one another. Common subexpressions must be identified and the optimal execution plan must be devised based on a cost-benefit analysis. This problem is hard [Rosenkrantz and Hunt III 1980], however, experiments with heuristic solutions have shown that important savings are possible [Roy et al. 2000; Sellis 1988; Zhou et al. 2007]. In this article, we perform work sharing for MapReduce, instead of traditional databases, with techniques tailored to our environment. We do not perform subexpression identification, but we point out that similar ideas are applicable for our framework, in the case of sharing parts of map output. Finally, in dynamic settings, work sharing can be performed at runtime [Harizopoulos et al. 2005; Candea et al. 2009; Chen et al. 2000]. In traditional database systems [Harizopoulos et al. 2005], data warehouses [Candea et al. 2009], and data stream engines [Madden et al. 2002; Chen et al. 2000] processing a correct mix of queries concurrently can be beneficial.

Filter reordering for single-query pipeline. Research effort has concentrated on expensive filter reordering, both in the context of relational databases [Chaudhuri and Shim 1993; Hellerstein and Stonebraker 1993; Hellerstein 1994; Avnur and Hellerstein 2000] and streaming databases [Madden et al. 2002; Babu et al. 2004; Liu et al. 2008a]. Traditional query optimizers assume filter independence [Hellerstein and Stonebraker 1993; Hellerstein 1994], and that the filter selectivities are static and known a priori. In that case, the optimal ordering is computed by sorting the filters in the order of cost to selectivity ratio. We refer to this ordering as RANK. Chaudhuri and Shim [1993] propose an approach based on dynamic programming. These techniques assume rich knowledge about the statistical properties of the predicates, and are not adaptive. `MRAdaptiveFilter` extension of `MRShare` concentrates on processing ad hoc data, with unknown statistical properties.

Avnur and Hellerstein [2000] proposed the EDDY operator enabling runtime control over the query plans executed by the query engine. The basic idea is to treat the execution as a process of *routing* tuples through the operators that need to be applied. Since the routing is dynamic, the solution is highly adaptive. It has been used in many contexts [Madden et al. 2002; Chandrasekaran et al. 2003] and to a wide range of operators. The EDDY operator has been designed for multithreaded environments, where each operator (e.g., associated with a filter) is running in a separate thread and tuples arrive independently of the consumption rate of the filters [Chandrasekaran et al. 2003]. The EDDY approach introduces substantial overheads for storing the state information, which is attached to each input tuple. In `MRAdaptiveFilter`, the filters are executed in a sequential manner with only one tuple being processed at a time, which allows for significant simplification of processing and much lower overheads. We also concentrate on the *policies* of routing the tuples among the operators.

Babu et al. show that the problem of finding the optimal ordering of filters for a given dataset in the presence of correlations is **NP**-hard [Babu et al. 2004]. They present the AGreedy algorithm, which monitors the conditional selectivities of the filters by executing all filters for a small fixed-size data sample typically in the thousands obtained from a sliding window of the input tuples. The statistics computed are used to maintain a fixed execution strategy at any point in time, based on a greedy invariant. AGreedy handles only single pipelines of queries. In our work, we propose a MapReduce-tailored solution that can also handle shared execution of queries with correlated, expensive filters.

Filter reordering for batched query pipelines. CACQ [Madden et al. 2002] addresses the problem of evaluating continuous queries over streams based on Eddy. The authors propose a set of techniques for handling multiple transformations on a single input stream to produce multiple outputs. Our solution, `MRAdaptiveFilter`, resembles the approach adopted by CACQ [Madden et al. 2002] for data streams, adapted to a single-threaded setting where filters are executed in a serial fashion. This allows for reducing the overhead of storing the state information. In our work, we also address the problem of how the tuples need to be routed throughout the execution in presence of sharing.

Munagala et al. [2007] introduce the problem of shared execution of multiple conjunctive queries. They show that the problem is a probabilistic version of SetCover, and present a *query-coverage*-based greedy algorithm (QueryCoverGreedy). We will refer to this approach as QueryCoverGreedy. Liu et al. [2008b] propose an improved greedy adaptive strategy based on *edge coverage*. We will refer to this approach as EdgeCoverGreedy. Both techniques assume that filter selectivities are known and constant over time. Similar to the single-query case, in our work we depart from this assumption.

3. MAPREDUCE PRELIMINARIES

In this section we review the MapReduce [Dean and Ghemawat 2004] model. 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.

Execution flow. The computation is distributed across a cluster of hosts, with a designated *coordinator* and the remaining *slave* machines. MapReduce engines utilize a *Distributed File System* (DFS) for storing the input and the output. Delving into the MapReduce functionality, 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. This results in m map tasks $map_i^1, \ldots, map_i^m$ to be processed by the slaves. The tasks are scheduled based on the availability of resources. The set of keys produced by the map tasks is partitioned into a user-defined number of r partitions, resulting in r reduce tasks.

Map task. A map task of job J_i processes its input split in two phases.

—*Fetching.* The assigned input split is read and parsed into a set of input key-value pairs $\{(K_{i1}, V_{i1})\}$, according to the specified input format.
—*Mapping.* Each pair (K_{i1}, V_{i1}) is processed by the user-defined *map* function, producing a new set of key-value pairs $\{(K_{i2}, V_{i2})\}$ as the intermediate output. The intermediate map output is stored locally, in r partitions.

Reduce task. A reduce task of job J_i has three phases.

—*Copying.* A slave is assigned the task and copies output of each map task, but only from the corresponding partition. The partitions reside in the local disks of the slaves on which the map tasks were executed.

—*Sorting.* When all outputs are copied, sorting is conducted to group all occurrences of each key K_{i2}.
—*Reducing.* The user-defined *reduce* function is invoked for each group $(K_{i2}, \{(V_{i2})\})$ to produce the final output set of key-value pairs $\{(K_{i3}, V_{i3})\}$ (possibly of size 1). The output is placed 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*.

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

3.1. 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 [Thusoo et al. 2009]. Each input tuple represents a relational tuple with a specified schema.

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

This 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 colocate occurences of each key $T.a$. Then, the aggregate function *aggr*() is applied for each group and the final output tuples are produced.

It should be also noted here that high-level languages enable one to obtain much more detailed understanding of the semantics of the program and allow for easier static analysis, which is beyond the scope of this article. At the same time, such languages do not limit the ability to collect important characteristics of the input and output data, and preserve the notion of input and output tuples as defined for the low-level MapReduce model.

4. SHARING OPPORTUNITIES

In this section, we discuss sharing opportunities in the MapReduce pipeline. We 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 nontrivial *sharing opportunities*. These opportunities have also been exploited in Pig [Gates et al. 2009]. First, we concentrate on sharing scans. Then, we demonstrate the opportunity of sharing intermediate data between map and reduce stages. Finally, we show that user-defined map functions can be shared entirely or partially. For simplicity, we will use two jobs, J_i and J_j, for our presentation. 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 [Gates et al. 2009]. Given that, we can merge the two pipelines into a single pipeline and scan the input data

only once. 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 Eq. (2). Instead, they are merged into one pipeline.

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

To distinguish between the tuples originating from two different jobs, we use a *tagging* technique that enables evaluating multiple jobs within a single job. The details are deferred to Section 8.

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. We denote K_{i2} and K_{j2} by K_{ij2}. Similarly for V_{i2} and V_{j2}, we denote them by V_{ij2}. 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 in savings on sorting and copying intermediate data over the network. Observe that we only share map output when two identical map output tuples originate from *the same* input tuple.

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*. Observe that a single tuple may be pushed to multiple reduce functions.

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

Producing a smaller map output results in savings on sorting and copying intermediate data over the network. Clearly, this mechanism is easily generalized to more than two jobs.

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.

In particular, when the map functions map_i and map_j consist of a set of expensive filters, sharing the computation in principle can reduce the execution cost. It is possible that some of these filters are common, even if $map_i \neq map_j$. Let $map_{i \cap j}$ be the shared set of filters of map_i and map_j, and $map_{i \setminus j}$ be the set difference of map_i and map_j (i.e.,

$map_i \setminus map_{i \cap j}$). The shared map pipeline is described as follows.

$$mapping_{ij} : (K_{ij1}, V_{ij1}) \rightarrow map_{i \cap j} \rightarrow \frac{map_{j \setminus i}}{map_{i \setminus j}} \rightarrow \frac{tag(i) + (K_{i2}, V_{i2})}{tag(j) + (K_{j2}, V_{j2})}$$

Similar considerations can be presented for the reduce stage of such jobs.

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 yields CPU savings. The I/O costs due to reading, sorting, copying, etc., are usually dominant, and thus in the first part of this article we concentrate on the I/O sharing opportunities. In fact, when the map computation is relatively simple, the I/O costs are dominating, and optimizing for CPU savings is not likely to introduce new savings. However, when the map computation is complex, the CPU savings can be substantial. Hence, in the latter part of the article we introduce additional optimizations for jobs with expensive map computation. In the next section, we present examples for the sharing opportunities introduced in this section.

5. EXAMPLES OF SHARING OPPORTUNITIES

In this section we provide several useful examples of sharing opportunities. For illustration purposes we use SQL notation (please see Section 3). We note here that syntactical analysis of jobs, for instance, identifying similar expressions and predicate containment, are beyond the scope of this article.

Example 5.1 (*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 as follows.

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

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

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

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 5.2 (*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.

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

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.

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

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

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

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

Example 5.3 (*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

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

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

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

The reducing pipeline is similar to the previous examples.

Example 5.4 (*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.

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

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

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

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

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

We remark that sharing parts of map functions has many implications. It involves identifying common subexpressions [Finkelstein 1982; Sellis 1988] and filter reordering [Chaudhuri and Shim 1996, 1999], which are hard problems. When the computation in the map stage is relatively simple, the cost of the job is strongly dominated by the I/O costs, and hence sharing the map computation is not likely to introduce any benefits. Hence, `MRShare` in its basic form considers only sharing opportunities that introduce the I/O savings. Let us now describe in more detail situations in which reducing CPU cost might bring additional savings.

Example 5.5 (*Expensive Filters - Singleton Jobs*). As a base case, consider a single MapReduce job performing aggregation. Consider an input table $T(a, b, c)$ and the following Hive-style query.

```
SELECT   T.a, sum(T.b)
FROM     T
WHERE    F1(T.c) AND
         F2(T.a)
GROUP BY T.a
```

Two equivalent map pipelines are described as follows.

$$mapping_i : T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow F1(T.c), F2(T.a) \rightarrow (T.a, T.b)$$
$$mapping_i : T \rightarrow (\emptyset, (T.a, T.b, T.c)) \rightarrow F2(T.a), F1(T.c) \rightarrow (T.a, T.b)$$

The resulting map task parses the input into tuples $(T.a, T.b, T.c)$. Then, filters F1 and F2 are applied to each tuple. At this point, we assume that the filters are expensive in evaluation. A good example of such would be index probes. The map output tuples are of the form $(T.a, T.b)$. The reduce stage performs the aggregation. The order of the execution of F1 and F2 can impact the performance, and in extreme cases might dominate the I/O cost of executing this job.

Languages such as Hive or Pig enable one to apply arbitrary expensive operation (e.g., network connection) to each input tuple. Hence optimizing ordering of such calls is of very high importance.

Example 5.6 (*Expensive Filters - Batched Jobs*). A common scenario in data warehousing is to perform joins of fact tables with dimension tables [Blanas et al. 2010]. Consider a fact table $T(t_1, t_2, t_3)$ and dimension tables $B(b_1, b_2)$, $C(c_1, c_2)$, $D(d_1, d_2)$, with $|T| >> |B|, |C|, |D|$. Furthermore, consider two example MapReduce queries, Q_i evaluating $T \bowtie_{T.t_1=B.b_1} B \bowtie_{T.t_2=C.c_1} C$, and Q_j evaluating $T \bowtie_{T.t_1=B.b_1} B \bowtie_{T.t_3=D.d_1} D$.

```
SELECT     COUNT(*)              SELECT     COUNT(*)
FROM       T, B, C               FROM       T, B, D
WHERE      T.t1 = B.b1 AND       WHERE      T.t1 = B.b1 AND
           T.t2 = C.c1                      T.t3 = D.d1
GROUP BY   T.t3                  GROUP BY   T.t2
```

Broadcast joins [Blanas et al. 2010] are an efficient technique for evaluating such joins; the entire dimension tables are broadcasted to all machines and then joined locally with the fact table in the map stage, by probing indices built for the dimension tables at each map task. Hence, the joins can be modelled as expensive filters of the input fact table $|T|$. The aggregation, if present, is performed in the reduce stage of such job. In our example, the map tasks of both queries perform filtering, using the dimension tables, and read the same input fact table T, hence this computation can be shared among them. Again, the execution order of the (join) filters may have a significant impact on the performance.

The map pipelines are described as follows.

$$\begin{aligned} mapping_i : T \rightarrow (\emptyset, (T.t_1, T.t_2, T.t_3)) &\rightarrow probe(B.b_1 == T.t_1), probe(C.c_1 == T.t_2) \\ &\rightarrow (T.*, B.*, C.*) \\ mapping_j : T \rightarrow (\emptyset, (T.t_1, T.t_2, T.t_3)) &\rightarrow probe(B.b_1 == T.t_1), probe(D.d_1 == T.t_3) \\ &\rightarrow (T.*, B.*, D.*) \end{aligned}$$

Here, $probe(B.b_1 == T.t_1)$ can be shared between both jobs. The map pipelines can be merged as described in Example 5.4. In general, however, the order in which the (join) filters will be applied has significant impact on the execution cost of the job. Hence, we consider two possible variants of merged pipelines.

$$\begin{aligned} mapping_{ij} : T &\rightarrow (\emptyset, (T.t_1, T.t_2, T.t_3)) \rightarrow probe(B.b_1 == T.t_1) \\ &\rightarrow \frac{probe(C.c_1 == T.t_2)}{probe(D.d_1 == T.t_3)} \rightarrow \frac{tag(i) + (T.*, B.*, C.*)}{tag(j) + (T.*, B.*, D.*)} \\ mapping_{ij} : T &\rightarrow (\emptyset, (T.t_1, T.t_2, T.t_3)) \rightarrow \frac{probe(C.c_1 == T.t_2)}{probe(D.d_1 == T.t_3)} \\ &\rightarrow probe(B.b_1 == T.t_1) \rightarrow \frac{tag(i) + (T.*, B.*, C.*)}{tag(j) + (T.*, B.*, D.*)} \end{aligned}$$

6. 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 [Dean and Ghemawat 2004]. Actually, our cost model is appropriate for any MapReduce system in which communication between the map and the reduce stage is based on sorting.

6.1. 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 I. 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 assume that approximate job characteristics can be obtained from historical data, or by sampling the input dataset. 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 is

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

Cost components. Assuming that we have n jobs processing input of size $|I|$, the cost of reading the input data from the distributed file system is

$$T_{read}(\mathbb{J}) = C_r \cdot n \cdot |I|. \tag{4}$$

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 as follows.

$$\begin{aligned} T_{sort-map}(\mathbb{J}) &= C_l \cdot \Sigma_{i=1}^{n} \left(m \times |M_i| \left(2 + 2 \left(\left\lceil \log_B \frac{|M_i|}{(B+1)} \right\rceil \right) \right) \right) \\ &= C_l \cdot \Sigma_{i=1}^{n} \left(|D_i| \left(2 + 2 \left(\left\lceil \log_B \frac{|D_i|}{(B+1) \times m} \right\rceil \right) \right) \right) \\ &\approx C_l \cdot \Sigma_{i=1}^{n} (|D_i| \lceil 2(\log_B |D_i| - \log_B m) \rceil) \end{aligned} \tag{5}$$

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

$$T_{sort-red}(\mathbb{J}) = C_l \cdot \Sigma_{i=1}^{n} (r \times |R_i| \lceil 2 \log_B m \rceil) = C_l \cdot \Sigma_{i=1}^{n} (|D_i| \lceil 2 \log_B m \rceil). \tag{6}$$

Thus the total cost of sorting is

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

In the case that no sorting is performed at the map tasks [Dean and Ghemawat 2004], we consider a slightly different cost function for sorting.

$$T_{sort}(\mathbb{J}) = C_l \cdot \Sigma_{i=1}^{n} |D_i| \cdot 2 \cdot (\lceil \log_B |D_i| + log_B m - log_B r \rceil). \tag{8}$$

Finally, the cost of transferring intermediate data is.

$$T_{tr}(\mathbb{J}) = \Sigma_{i=1}^{n} C_t \cdot |D_i|. \tag{9}$$

Since we implemented `MRShare` in Hadoop, for the remainder of this article we use Eq. (7) to calculate the sorting cost. However, all our algorithms can be adjusted to handle the sorting cost (Eq. (8)) of the original MapReduce [Dean and Ghemawat 2004].

[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 nongrouping scenarios.

6.2. 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.

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, we have what follows.

$$\begin{aligned} T_{read}(J_G) &= C_r \cdot |I| \\ T_{sort}(J_G) &= C_l \cdot |X_G| \cdot 2(\lceil \log_B |X_G| - \log_B m \rceil + \lceil \log_B m \rceil) \\ T_{tr}(J_G) &= C_t \cdot |X_G| \\ T(J_G) &= T_{read}(J_G) + T_{sort}(J_G) + T_{tr}(J_G) \end{aligned} \tag{10}$$

We can determine if sharing is beneficial by comparing Eq. (3) with (10). 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 $d_i = |D_i|/|I|$, d_i be the *map output ratio* of the map stage of job J_i, and $x_G = |X_G|/|I|$ 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 Eqs. (3) and (10) we have.

$$f(n-1) + g\Sigma_{i=1}^{n} d_i + 2\Sigma_{i=1}^{n}(d_i \cdot p_i) - x_G(g + 2 \cdot p_G) \geq 0. \tag{11}$$

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 (11) 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 [Gates et al. 2009]. This is also verified by our experiments.

7. 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.

7.1. Sharing Scans Only

Here, we consider the scenario in which we share only scans. We first formulate the optimization problem (Section 7.1.1) and show that it is **NP**-hard (Section 7.1.2). In Section 7.1.3 we relax our original problem. In Section 7.1.4 we describe `SplitJobs`, an exact dynamic programming solution for the relaxed version, and finally in Section 7.1.5 we present the final algorithm `MultiSplitJobs`.

7.1.1. 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.

$$x_G = d_1 + \cdots + d_n \tag{12}$$

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.

$$\mathcal{SS}(G) = \Sigma_{i=1}^{n} \left(f - 2 \cdot d_i \cdot (p_j - p_i + \delta_G)\right) - f \tag{13}$$

Recall that J_j is the constituent job of group G, that is, 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 $\left(f - 2 \cdot d_i \cdot (p_j - p_i + \delta_G)\right)$, 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.

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

Problem 1 (*Scan-Shared Optimal Grouping*). Given a set of jobs $\mathbb{J} = \{J_1, \ldots, J_n\}$, group the jobs into S nonoverlapping groups $G_1, G_2, \ldots, G_S$, such that the overall sum of savings $\sum_s \mathcal{SS}(G_s)$ is maximized.

7.1.2. Problem 1 Hardness.

THEOREM 7.1. *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 $t = \frac{\sum_{i=1}^{n} a_i}{2}$. 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 $\forall i\ p_i = 0$, that is, 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 groups 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 $|X_{G_s}| > B$ (a.k.a. $x_{G_s} > 0.5$); 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 $\mathcal{SS}(G_s) = \left(n_{G_s} \cdot f - 2 \cdot x_{G_s} \cdot p_{G_s}\right) - f < 0$. 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. □

7.1.3. A Relaxation of the Problem. In its original form, Problem 1 is **NP**-hard. 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 and not on the rest of the jobs participating in the groups.

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

$$gain(i, j) = f - 2 \cdot d_i \cdot (p_j - p_i + \delta_j) \tag{14}$$

This 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 of one input scan (f) minus the cost of additional sorting of J_i's map output, that is, if the original number of passes p_i is lower than $p_j + \delta_j$.

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.

Relaxed Problem 1 Property.

THEOREM 7.2. *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 cannot 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. □

7.1.4. 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. 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 $GAIN(t,u) = \sum_{t \le i \le u} gain(i,u)$ to be the total gain of merging a sequence of jobs $J_t, \ldots, J_u$ into a single group. $GAIN(t,u)$ 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. Clearly, $p_u = max\{p_t, \ldots, p_u\}$. The overall group savings from merging jobs $J_t, \ldots, J_u$ become $GS(t,u) = GAIN(t,u) - f$. $GS(t,u)$ accounts for one additional input scan per group. 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 $J_1, \ldots, J_l$. If we know that the last group contains jobs $J_i, \ldots, J_l$, then we have: $c(l) = c(i-1) + GS(i,l)$. In general, $c(l) = max_{1 \le i \le l}\{c(i-1) + GS(i,l)\}$.

$c(l)$ 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 $c(l)$.

As a base case, for computing $c(1)$, we need $c(0)$. We set $c(0) = 0$, then $c(1) = GS(1,1)$, which is what we want. 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 $source()$. It points to the places where the c value was obtained. When $c(l)$ is computed, if $c(l)$ was obtained from $c(i-1)$, we set $source(l) = i$. After $c(k)$ (i.e., the total savings) is computed, we trace back to split the original sequence of jobs. The last group starts at job $J_{source(k)}$ and spans all jobs until J_k. The previous group starts at job $J_{source(source(k))}$, and goes up to job $J_{source(k)-1}$, etc. The pseudocode is shown in Algorithm 1.

ALGORITHM 1: SplitJobs

```
1 SplitJobs(J_1, ..., J_n)
2   compute GAIN(i,l) for 1 ≤ i ≤ l ≤ n.
3   compute GS(i,l) for 1 ≤ i ≤ l ≤ n.
4   compute c(l) and source(l) for 1 ≤ l ≤ n.
5   return c and source
```

`SplitJobs` has $O(n^2)$ time and space complexity, and can be evaluated by fast DP solvers. This is acceptable for our applications since the number of jobs in a batch is not expected to exceed the hundreds. Tables $c()$ and $source()$ are $O(n)$ in size. Filling each entry requires up to n iterations. Thus, the cost of filling the tables is $O(n^2)$.

Discussion. By assuming that the number of passes always behaves as in the worst case for a given J_j as the constituent job, we miss some sharing opportunities. Consider J_j as the constituting job for group G, then $GAIN(j,j) = f - 2 \cdot d_j$. This means that if $2 \cdot d_j > f$, we will never pick job J_j as a constituent job (increase in the number of passes will defeat the savings from scan). However, imagine that there is another job J_i which, merged with J_j, does not increase the number of passes and saves some work, hence by our pessimistic assumption this will not be considered by our program.

On the other hand, by assuming that the number of passes never increases (which is the best case) then $gain(j,j) = f$, we can return a suboptimal solution. Assume that

there is another job J_i which when merged with J_j increases the number of sorting passes by 1. Our best-case assumption would take $GAIN(i, j) = f - 2d_i(p_j - p_i)$. Then the $GS(i, j)$ will be $GAIN(i, j) + GAIN(j, j) - f$, which is equal to $gain(i, j)$. However, in reality it will be less because the number of passes will increase.

By taking the pessimistic assumption we ensure soundness, possibly missing some opportunities, but we avoid considering what happens to the number of passes for each possible combination of jobs. Also, our solution is applicable when we bound the number of jobs that can be shared by MAX. Then we obtain δ_j by looking what happens to the number of passes when job J_i is merged with MAX largest of all jobs J_i, such that $d_i < d_j$. The program needs to be modified by setting $GS(i, j) = -\infty$ for all $i \leq j - MAX$.

7.1.5. Improving SplitJobs. In this section, we make some observations about our relaxed version of Problem 1, that give more improvements. Thus, we propose `MultiSplitJobs`, a heuristic algorithm, that performs at least as well as `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, $(J_3J_4J_5)$, (J_6J_7), J_8, J_9, J_{10}, namely, J_1, J_2, J_8, J_9, and J_{10} are singletons. 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: (1) The iteration returns some new groups (e.g., $(J_1J_2J_8)$). 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). (2) 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` is shown in Algorithm 2. `MultiSplitJobs` yields a grouping at least as good as `SplitJobs` and runs in $O(n^3)$.

ALGORITHM 2: MultiSplitJobs

```
1 MultiSplitJobs(J_1, ..., J_n)
2   𝕁 ← {J_1, ..., J_n} (input jobs)
3   𝔾 ← ∅ (output groups)
4   while (𝕁 ≠ ∅)
5    compute δ_j for each J_j ∈ 𝕁
6    ALL = SplitJobs(𝕁)
7    𝔾 ← 𝔾 ∪ ALL.getNonSingletonGroups()
8    SINGLES ← ALL.getSingletonGroups()
9    if |SINGLES| < |𝕁| then
10     𝕁 ← SINGLES
11   else
12     J_x = SINGLES.theSmallest()
13     𝔾 ← 𝔾 ∪ {J_x}
14     𝕁 ← SINGLES \ J_x
15   end
16  end
17 return 𝔾 as the final set of groups.
```

7.2. Sharing Map Output

We move on to consider the problem of optimal grouping when jobs share not only scans, but also their map output. In Section 7.2.1 we formally define the Scan+Map-Shared Optimal Grouping problem. We show that the exact solution for the problem is prohibitive. Thus, in Section 7.2.2 we present a heuristic algorithm.

7.2.1. Problem Formulation. We now enable jobs to share map output as well as scans. The map output ratios d_is of the original jobs and x_G of the merged jobs no longer satisfy Eq. (12). Instead, we have

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

where the union operator takes into account lineage of map output tuples. By Eq. (11), the total savings from executing n jobs together are

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

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 nonoverlapping 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

$$\sum_{s=1}^{S} \left(f \cdot (n_{G_s} - 1) - x_{G_s}(g + 2(p_{G_s})\right), \tag{15}$$

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.

THEOREM 7.3. *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.*

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 the 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. □

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 article, we assume that all such containments have been identified.

7.2.2. 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

$$x_G = d_j + \gamma \sum_{i=1, i \neq j}^{n} d_i. \tag{16}$$

We remark that if $\gamma = 1$, then no map output is shared (Eq. (16) reduces to Eq. (12)). 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 nonoverlapping groups $G_1, G_2, \ldots, G_S$, such that the overall sum of savings is maximized.

By Eq. (11) the total savings from executing n jobs together are

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

As before, we define the gain.

$$\begin{aligned} gain(i, j) &= f - \big(g(\gamma - 1)d_i + 2 \cdot d_i \cdot (\gamma(p_j + \delta_j) - p_i)\big) \\ gain(j, j) &= f - 2 \cdot d_j \cdot \delta_j \end{aligned} \tag{17}$$

$gain(i, j)$ 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 nonshared scenario, where we sort the entire map output of J_i in p_i passes.

For $\gamma = 1$, Eq. (17) reduces to Eq. (14), and δ_x is obtained as before, by checking the worst possible case for each J_x. Notice that p_i is a monotone function of d_i, as the number of sorting passes increases monotonically with the size of the input. $g(\gamma - 1) \leq 0$ and is a constant for a given setting. $g(\gamma - 1)d_i$ is decreasing with increasing p_i; the same holds for $(\gamma(p_x + 1) - p_i)$. Hence, $gain(i, x)$ is a monotonically decreasing function of p_i for a given p_x, and monotonically decreasing function of p_x for a given p_i. For any given p_x and p_y, if $p_x \leq p_y$ then $gain(i, x) \geq gain(i, y)$. For any given p_i and p_j and p_x, if $p_i \leq p_j$ then $gain(i, x) \leq gain(j, x)$. The total savings of executing group G together versus executing each job separately are

$$savings(G) = \sum_{i=1}^{n} gain(i, j) - f. \tag{18}$$

Recall that J_j is the constituent job for a group G. Each job J_i ($i \neq j$), merged into G, saves one scan of the input table. It also saves on copying of a $(1 - \gamma)$ part of its map output. On the other hand, it incurs the cost of additional sorting if the number

of passes p_G is greater than the original number of passes p_i (but only for the γ part of its original map output). One scan of the input is added per group.

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 7.1.4 (Algorithm 2). *GAIN*, *GS*, and c are defined as in Section 7.1.4, but this time using Eq. (17) 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).

Parameter γ can be interpreted as the aggressiveness of merging. If it is set to 0, we assume that the entire map output is shared (optimistic scenario). If it is set to 1, we assume that no map output is shared among jobs (pessimistic scenario); the `MultiSplitJobs`$^{\gamma}$ reduces to the `MultiSplitJobs`.

8. IMPLEMENTING MRSHARE

We implemented our framework, `MRShare`, on top of Hadoop. However, it can be easily plugged into 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 [Agrawal et al. 2008]. Then, `MultiSplitJobs` is called to compute the optimal grouping of the jobs. Afterwards, the groups are *rewritten*, using a *metamap* and a *metareduce* function. These are `MRShare`-specific containers for merged map and reduce functions of multiple jobs. These are implemented as regular map and reduce functions, and their functionality relies on *tagging* (explained shortly). 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. Notice that the implementation of `MRShare` is orthogonal to the high-level languages Hive or Pig. For example, Hive uses SQL-like syntax which is compiled to map and reduce functions.

Next, we focus on the tagging technique that enables evaluating multiple jobs within a single job. In particular, in Section 8.1, we describe tagging for sharing scans, and then in Section 8.2 for sharing both scans and map output.

8.1. Tagging for Sharing-Only Scans

Recall from Section 4 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 associated with 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.

Example 8.1 (*Tagging for Sharing Scans*). Table I 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

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

Table I. 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	$reduce_1$

Table II. 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

Table III. 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 IV. 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	$reduce_1$; $reduce_2$; $reduce_3$

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. $B_6 \dots B_0$ determines the job in which the tuple belongs.

8.2. 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. Here is an example.

Example 8.2 (*Tagging for Sharing Map Output*). Table II describes tuples produced by n mapping pipelines from one input tuple, sorted according to *key*. The *tag*() field is byte long ($B7, \dots, B0$), as before.

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

Table IV 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.

In both cases, for sharing-scans-only and sharing map output, we do not consider separating the map output after the map stage; a single map output is produced for the merged jobs. 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 that more

files would need to be managed, which means more overhead and errors, and that running multiple reduce stages would cause network and resource contention.

9. EMPIRICAL EVALUATION OF MRSHARE

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 [Amazon 2006]. Our real-world 30GB text dataset consists of blog posts [Blogscope 2005]. In Section 9.3 we experimentally validate our cost model for MapReduce. Then, in Section 9.4, we establish that the GreedyShare policy is not always beneficial. Subsequently, in Section 9.5 we evaluate our MultiSplitJobs algorithm for sharing scans and show that our cost-based approach improves the overall time performance. In Section 9.6 we demonstrate that sharing intermediate data can introduce tremendous savings. In Section 9.7 we evaluate our heuristic algorithm MultiSplitJobs$^{\gamma}$ for sharing scans and intermediate data. Finally, in Section 9.8 we study the scalability of our techniques.

9.1. Experimental Setting

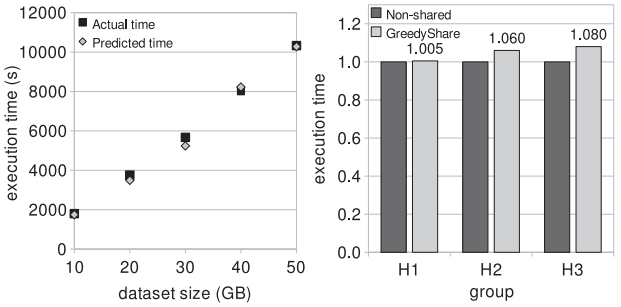
We ran all experiments on a cluster of 40 virtual machines, using Amazon EC2 [Amazon 2006] unless stated otherwise. We used the small-size instances (1 virtual core, 1.7GB RAM, 160GB of disk space). All settings for Hadoop were set to defaults. Our real-world 30GB text dataset consists of blog posts [BlogScope 2005]. We utilized approximately 8000 machine hours to evaluate MRShare. We ran each experiment three times and averaged the results.

We ran *wordcount* jobs that were modified to count only words containing given regular expressions (a.k.a. *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.

9.2. 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. Hence, we set $g = 2.3$, 2 accounting for double I/O cost, and 0.3 accounting for the network cost. We emphasize that these parameters are system dependent and need to be measured for each setting, especially in the presence of multiple jobs competing for resources. In particular, custom in-house clusters might offer better characteristics, but we chose EC2 since it potentially applies to more potential users.



(a) Cost Model (b) Non-shared vs `GreedyShare`.

Fig. 1. Cost model and naïve sharing.

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.

9.3. Validation of the Cost Model

First, we validate the cost model that we presented in Section 6. In order to estimate the system parameters, we used a text dataset of 10GB and a set of MapReduce jobs based on random *grep-wordcount* queries 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 50GB 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 6 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.

9.4. The GreedyShare Approach

Here, we demonstrate that the `GreedyShare` approach for sharing scans is not always beneficial. In fact, for large intermediate data, it can even lead to poor performance. We created three random series of 16 *grep-wordcount* jobs ($H1$, $H2$, $H3$), which we describe in Table V, with constraints on the average map output ratios (d_i). The maximum d_i was equal to 1.35[4].

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, for instance, for $H2$, the 6% increase incurs an overhead of more than 30 minutes.

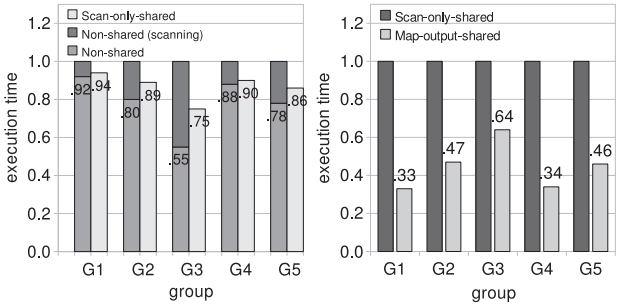
[4]This should not be confused with the *selectivity* in relational database systems, as in MapReduce the map output size may be arbitrarily larger than the input.

Table V. `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	$0.9 \leq d_i$

Table VI. `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



(a) sharing scans (b) sharing map output

Fig. 2. `MultiSplitJobs`.

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 that depend on the size of the intermediate data.

9.5. MultiSplitJobs Evaluation

Here, we study the effectiveness of the `MultiSplitJobs` algorithm for sharing scans. As discussed in Section 6, 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 VI. 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. We can clearly

observe that the larger the intermediate data sizes, the less the relative savings. The robustness of our method is also verified by the result for $G5$; we achieved savings of 14% for queries of random output size, which translated to reducing the running time by more than one hour in absolute values.

In every case, MultiSplitJobs yields substantial savings 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 VI 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.

Summarizing, in all experiments, MultiSplitJobs yielded substantial savings that were close to the best possible by using shared scans.

9.6. 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 VI). We used the same groupings obtained by MultiSplitJobs. We enable the sharing map outputs feature and we compare it to sharing scans only.

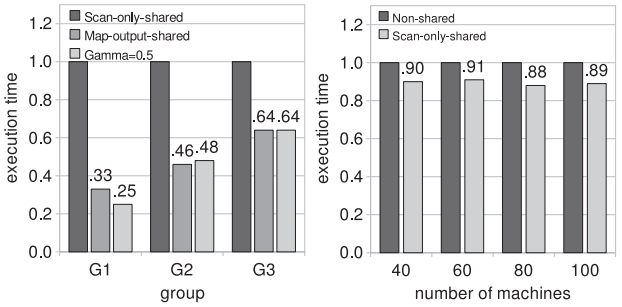
Our results are illustrated in Figure 2(b). Notice that, since we generated queries randomly, the degree of sharing among filtering predicates is most likely to follow the independence assumption. For $G1$, where map output sizes were large, the jobs shared large portions of intermediate data. The execution 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.

9.7. 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, for example, 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 article. However, we note that when gamma is set to 1 the scheme performs at least as well as MultiSplitJobs.

Figure 3(a) 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.



(a) `MultiSplitJobs`$^{\gamma}$ (b) MRShare-scale independence

Fig. 3. `MRShare` evaluation.

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`.

9.8. 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 VI) and measured the resulting savings for four cluster sizes, namely, 40, 60, 80, and 100. Figure 3(b) illustrates our results. In each case, the left bar represents the normalized execution time when no sharing occurs. The relative savings 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, for example, scans, do not depend on the cluster's size.

9.9. 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.

10. EXPENSIVE FILTER ORDERING FOR SINGLETON JOBS IN MAPREDUCE

So far, we have been considering jobs for which the main cost is influenced by the I/Os. In Section 5, we presented cases for which optimizing CPU performance might yield additional savings. Optimizing single- and batch-mode MapReduce queries, which are CPU intensive, becomes as important as optimizing the I/O-intensive case. The key distinctive characteristics of MapReduce come from the nature of the underlying data and the model of the computation itself. First, in the relational domain, it is natural to assume in-depth statistical knowledge about the input data, hence static query

optimization can be successfully applied in this domain. In MapReduce, however, the optimizations should primarily rely on the execution feedback [Olston et al. 2008a], especially in the presence of ad hoc data where statistical properties can vary greatly. This mandates the use of adaptive query optimization techniques. Second, the highly parallel nature of MapReduce, where possibly thousands of tasks are executed in parallel, makes the use of a central optimizer prohibitive in cost (e.g., cost of exchanging information). Hence, we advocate the use of highly adaptive optimization techniques that can be applied locally by the map or reduce tasks, and utilize only the local execution feedback. Moreover, independent tasks operating on small portions of the input should adapt very quickly to changing data properties, as they are lacking global view of the data.

`MRShare`, given a set of jobs, derives grouping of jobs into subsets that will be executed more efficiently. In particular, the subsets might be of size 1 – singletons. In this section, we introduce `MRAdaptiveFilter`, an extension of `MRShare` that allows for additional savings when the jobs are performing expensive operations on their input. Without loss of generality, assume that each job J_i is derived from a high-level query Q_i. This is a common scenario when using high-level languages (e.g., Hive), where SQL-like queries are translated into corresponding MapReduce jobs. Hive, for example, translates selection conditions into a series of filter operators executed in the map stage of the job. Here, we concentrate on execution of a single job with expensive filters. This will be the building block for the case of executing a group of queries, which will be presented in the next section. We consider the case where the expensive filters are applied in the map stage, since this is a more common use case. Similar analysis can be shown for the reduce stage.

10.1. Problem Formulation

Assume that the jobs are given in the form of high-level queries (e.g., Hive style). We say that job J_i evaluates query Q_i. Let the map pipeline map_i of a MapReduce job J_i be a conjunction of N commutative filters, $F_1, \ldots, F_N$, which are applied to every map input tuple $\mathcal{T}$ of job J_i. Each filter F_x is associated with a *cost* c_x per tuple that can be easily estimated. The probability that an input tuple $\mathcal{T}$ passes the filter F_x (i.e., $F_x(\mathcal{T}) = 1$) is p_x. Clearly, $F_x(\mathcal{T}) = 0$ with *drop probability* of $(1 - p_x)$. If the input tuple passes all filters, it is produced as a map output tuple and sent to the reduce stage. For a given ordering of filters $1, \ldots, N$ and a given tuple $\mathcal{T}$, we say that

$$C_x(\mathcal{T}) = \begin{cases} c_x & \text{if } \forall_{k \leq x} F_k(\mathcal{T}) = 1 \\ 0 & \text{otherwise} \end{cases}. \tag{19}$$

Then, the cost of executing the pipeline of filters in a single map task map_i^y of job J_i, operating on input split $\mathcal{I}_i^y$, under fixed ordering is

$$Cost_i^y = \sum_{\mathcal{T} \in \mathcal{I}_i^y} \sum_{x=1}^{N} C_x(\mathcal{T}). \tag{20}$$

By Eq. (20), the cost is only incurred by filters that are actually executed, since once a tuple is rejected no other filters need to be evaluated. Clearly, the ordering of filters has impact on $Cost_i^y$. The problem is to find the optimal ordering of the filters that minimizes the total execution cost of the map stage, hence $\sum_{y=1}^{m} Cost_i^y$. We assume that the costs of the filters are known since in practice they can be easily estimated via sampling.

The inapplicability of the techniques presented in Section 2, used in standard query optimizers, is caused by the nature of the underlying data and the model of the

computation itself, as discussed previously. Hence, in the next section we present a `MRAdaptiveFilter`, a MapReduce-tailored extension of `MRShare` for handling jobs with expensive filters. The filters in `MRAdaptiveFilter` are executed in sequential manner, which allows for simplicity of processing and low processing overheads. At any point in time, only one input tuple is processed, which preserves MapReduce task interfaces. `MRAdaptiveFilter` performs optimizations on a per-task basis, which eliminates the need of communication between tasks, preserves the MapReduce encapsulation, and avoids having a centralized optimizer. The immediate execution feedback allows for high adaptivity of our technique with little computation overhead.

10.2. MRAdaptiveFilter – Adaptive Filter Ordering in MapReduce

We recall that in `MRShare`, the map stage of each job is wrapped in a *metamap* function. `MRAdaptiveFilter` functionality is supported by an additional *MRAF* module stored in the metamap function of `MRShare`. Let us consider a MapReduce job processing a single pipeline of filters in the map stage. An example metamap function using `MRAdaptiveFilter` functionality is shown in Algorithm 3. The initialization stage creates the evaluation module, namely, the MRAF module. The map function passes the map input tuples to the module (line 9) and produces map output depending on the outcome (line 10).

ALGORITHM 3: Mapper

```
1 class Mapper()
2   function init()
3     MRAF eval = new MRAF(N) // create the evaluation module for a single query with N
            filters
4     eval.addFilter(1, new UserFilter(...)) // adding filters to the pipeline
5     eval.addFilter(2, new UserFilter(...))
6     ...
7
8     function Map(Key key, Value value)
9      if (eval.evaluate(k,v) == true)
10       output.collect(...) // produce output
```

The MRAF module stores information about all filters that need to be evaluated for a given map pipeline in a vector that we call *filters*. Each filter represents user-defined functionality (e.g., index probe). Each filter has a number of *tickets* associated with it, stored *tickets* vector. The number of tickets determines how likely a filter is to be chosen for execution. Updating this vector is discussed in depth later in this section. In addition, we maintain the *ready* vector which holds information about the candidate filters that can be executed next on an input tuple. An important observation is that the state information is not attached to each processed tuple but is stored on a per-task basis, which substantially limits overheads.

Algorithm 4 shows the implementation of the MRAF module. When a tuple is read from the input, it is then passed to the MRAF evaluation module (line 9). A lottery is held to determine which filter should be evaluated as the first one (line 12). We employ a lottery scheduling [Waldspurger and Weihl 1994] scheme for choosing filters to be evaluated. Depending on the status of the *ready* bits, we choose a filter with probability proportional to the number of tickets stored in *tickets*. After the filter is evaluated and if the tuple passes, the appropriate *ready* bit is set to 0 (line 16). In this case, we hold another lottery among the remaining filters, with *ready* set to 1. If the tuple fails, the *tickets* is updated for the filter by adding Δ (Line 14), and we return false

ALGORITHM 4: MRAF

```
1 class MRAF()
2     function addFilter(int x, Filter F)
3         filters[x] = F // add user–defined filter F
4
5     function init(int N)
6         filters[1..N] ← null // user-defined filters
7         tickets[1..N] ← 0 // number of tickets for each filter
8
9     function evaluate(Key key, Value value)
10        ready[1..N] ← 1 // "ready" flag for each filter
11        for k=1 to N do
12          x = chooseFilter()
13          if (filters[x](key,value) == false
14              tickets[x]+ = Δ
15              return false
16          ready[x] = 0
17          return true
18
19      function chooseFilter()
20          return x ∈ [1..N] with probability proportional to tickets[x], such that ready[x]==1
```

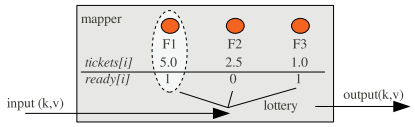


Fig. 4. `MRAdaptiveFilter` single pipeline.

immediately. Then, another tuple is read from the input and the *ready* table is reset. The algorithm shows the simplistic scenario when each filter returns only true or false on the input tuple. In general, user-defined filters can accommodate other functionality (e.g., probing a hash table, performing local join). The example in Figure 4 shows the state of a metamap task (i.e., the MRAF module) after an input tuple has passed filter $F2$, and filter $F1$ has been chosen to be evaluated next.

Now, we turn our attention to the crucial element of the solution. In Algorithm 4, when a tuple is discarded by a filter, the *tickets* vector is updated. The vector is used to determine the order of the evaluation, hence choosing the appropriate value of Δ will determine the performance gains. The value with which the suitable cell of the *tickets* vector is changed constitutes a *ticketing policy*.

10.2.1. Ticketing Policies. The ticketing policies constitute the core of the `MRAdaptiveFilter` framework. We propose the following policies.

SimpleTicketing. Every time filter F_x drops a tuple, it is assigned a number of tickets. In particular, we add Δ_{ST} tickets and add it to $tickets[x]$ cell, where

$$\Delta_{ST} = \frac{1}{c_x}. \tag{21}$$

By doing so, we promote inexpensive filters that discard tuples early. Hence filters with high drop probabilities $(1 - p_x)$ have higher chances of being awarded with a higher number of tickets, proportionately to their drop probability. This policy mirrors the static RANK ordering.

OrderedTickting. This time we also consider the number of filters that have passed the tuple before it was dropped. Namely, if F_x drops a tuple and it is evaluated as the k^{th} filter (Algorithm 4, line 11) for the given input tuple, $tickets[x]$ is increased by Δ_{OT}, where

$$\Delta_{OT} = \frac{1}{(k \times c_x)}. \tag{22}$$

The fact that $k-1$ other filters have been evaluated before F_x diminishes its "pruning power". Compared to *Simple Ticketing*, *Ordered Ticketing* favors more filters that discard tuples early. This policy is suitable for correlated filters, promoting filters with high unconditional pruning power.

10.2.2. History Inflation. Since filter selectivities are likely to be variable over time and across different map tasks, a *history* component is necessary. We enhance the `MRAdaptiveFilter` algorithm by introducing a *history inflation* heuristic. We decrease the number of tickets for each filter *inflationFactor* times every *inflationWindow* tuples. In our experiments we demonstrate that this policy improves the adaptivity of `MRAdaptiveFilter` with no significant overhead. This heuristic is orthogonal to the ticketing policies presented before. Algorithm 5 shows additional features within the MRAF module.

ALGORITHM 5: MRAF.eval()

```
1 class MRAF()
2     function eval(Key key, Value value)
3         if ++tupleCount % inflationWindow == 0
4             tickets = tickets · 1/inflationFactor
5         ready[1..N] ← 1
6         ...
```

In `MRAdaptiveFilter`, different policies can be adopted simultaneously by different map tasks. This opens a variety of optimization opportunities, where the decision on which policy should be used is made as the query progresses.

11. EXPENSIVE FILTER ORDERING FOR SHARED EXECUTION OF JOBS IN MAPREDUCE

In this section we present the final design of `MRAdaptiveFilter` for adaptive shared execution of multiple MapReduce jobs with map pipelines consisting of expensive user-defined filters. When utilized for a single-query evaluation, the solution reduces to the solution presented in Section 10. Similarly to the single-query case, the tuples are processed one at a time and the filters are evaluated in a sequential manner, which preserves MapReduce task interfaces and encapsulation and allows for minimizing processing overheads. Our starting point is a group of queries obtained from `MRShare` that potentially can be further optimized for CPU performance.

11.1. Problem Formulation

Consider an incoming stream of map input tuples, and q queries (MapReduce jobs) to be evaluated over the stream, that is, a single group obtained from `MRShare`. Our goal here is to further optimize their execution as a single job. Let $\mathcal{M}$ be the set of q map pipelines $\mathcal{M} = \{map_1, \ldots, map_q\}$ processing the input stream for each of the q MapReduce shared jobs. For clarification, a map pipeline map_i is the map pipeline of the original job J_i. As before, we assume that a job J_i is obtained from a high-level query Q_i, and all jobs are executed by a single `MRShare` job $J_{\mathcal{M}}$. Each pipeline is a conjunction

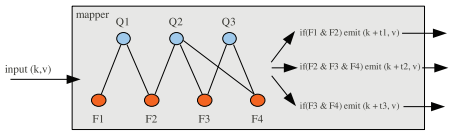


Fig. 5. Multiple-query execution in MapReduce.

of filters on the input tuple, $map_i = F_i^1, \ldots, F_a^X$. Some filters may be shared among the pipelines. Let $\mathcal{F} = \{F_1, \ldots, F_N\}$ be the set of N distinct filters occurring over the set of pipelines. As before, each filter F_x is associated with cost c_x. Figure 5 shows a *query filter* graph of a single map task of job $J_\mathcal{M}$ evaluating three queries ($Q1$, $Q2$, $Q3$) that share the underlying filters (i.e., $F1$, $F2$, $F3$, and $F4$). As one can see in the example, the evaluation of filters $F3$ and $F4$ can be shared for queries $Q2$ and $Q3$. In particular, if $F3$ discards an input tuple, no other filters for $Q2$ and $Q3$ need to be evaluated.

The goal is to minimize the execution time to evaluate $\mathcal{M}$ over the input dataset. For a given ordering of filters $1, \ldots, N$ and a given tuple $\mathcal{T}$, we say that

$$C_x(\mathcal{T}) = \begin{cases} c_x & \text{if } \exists_{map_l \in \mathcal{M}} \forall_{k \leq x, F_k \in map_l} F_k(\mathcal{T}) = 1 \\ 0 & \text{otherwise} \end{cases}, \tag{23}$$

namely under a fixed ordering of filters, we pay the cost of executing filter F_x, if there exists a pipeline for which the filters $F_1, \ldots, F_{x-1}$ did not discard a tuple. Then, the cost of executing $\mathcal{M}$ in a single map task $map_\mathcal{M}^y$, operating on input split $\mathcal{I}_\mathcal{M}^y$ of job $J_\mathcal{M}$, under fixed ordering is

$$Cost_\mathcal{M}^y = \sum_{\mathcal{T} \in \mathcal{I}_\mathcal{M}^y} \sum_{x=1}^{N} C_x(\mathcal{T}). \tag{24}$$

A shared execution policy for the set of filters $\mathcal{F}$ and the set of map pipelines $\mathcal{M}$ is a function that takes as an input the set of filters evaluated so far and decides the next filter to be evaluated, as the filters are evaluated in a serial fashion. Munagala et al. [2007] show that obtaining the optimal policy is **NP**-hard, even when the statistical knowledge about the underlying filters is complete. Hence, we devise a lightweight heuristic solution tailored to MapReducethat makes adaptive decisions by utilizing execution feedback.

11.2. MRAdaptiveFilter – for Shared Execution in MapReduce

Here, we detail the extension of `MRAdaptiveFilter` for handling shared execution of multiple MapReduce jobs with expensive user-defined filters. We build on top of the solution presented for adaptive filter ordering for the single-query scenario (Section 10).

Similar to the single-job case, the metamap function stores an MRAF module that implements `MRAdaptiveFilter` functionality. Let us consider a MapReduce job processing q pipelines of filters in the map stage, hence evaluating q distinct queries. An example metamap function using `MRAdaptiveFilter` functionality is shown in Algorithm 6.

In the initialization phase (lines 4–20), the filters are added to each query to construct the query filter graph. The map function calls the evaluation module for each input tuple (line 23) and retrieves the results. Depending on the outcome of the evaluation, map output tuples are produced for each query (line 28).

The MRAF module stores *filters* and *ready* vectors as before, with information about the filters. Each of the q map pipelines is represented in the $query\_filter$ matrix (e.g., Figure 5). Each of the q rows in this matrix contains information about the participation

ALGORITHM 6: Mapper

```
1 class Mapper()
2    function init()
3       MRAF eval = new MRAF(q,N) // create the evaluation module for q queries, with n
         filters
4       filter_1 = new UserFilter(...) // create n filters with user defined functionality
5       ...
6       filter_N = new UserFilter(...)
7
8       eval.initFilter( filter_1,1)
9       ...
10       eval.initFilter( filter_N,N)
11
12       eval.addFilter(1,1) // add filters to the first pipeline
13       ...
14       eval.addFilter(5,1)
15
16       ...
17
18       eval.addFilter(4,q) // add filters to the q^th pipeline
19       ...
20       eval.addFilter(9,q)
21
22    function Map(Key key,Value value)
23       eval.evaluate(key,value)
24       result = eval.getReturnValues()
25
26       for(int k = 1; k <= q; i++)
27         if (result[k] == true)
28               output . collect (..., k) // produce output for each query pipeline
```

of each filter in each pipeline. In particular $query\_filter_{(k,x)} == 1$ iff $F_x \in map_k$ – filter F_x belongs to pipeline map_k.

In addition, we store information about whether each pipeline has been evaluated in the *pipeline_done* vector and the *cover* vector for each filter, which indicates in how many queries the filter appears. As before, a lottery is held to decide which filter is evaluated next, based on the *tickets* and *ready* vectors. Notice that the entire state information is stored on a per-task basis, and does not need to be replicated for each input tuple. Algorithm 7 shows the implementation of the MRAF object.

In Algorithm 7, when an input tuple is passed to the evaluation function (line 13), first the state information is initialized, namely *ready*, *pipeline_done*, and *cover* vectors. The evaluation then iterates and chooses filters based on *tickets*, from filters that have not been evaluated so far (based on *ready*). When a tuple does not pass filter F_x, then:

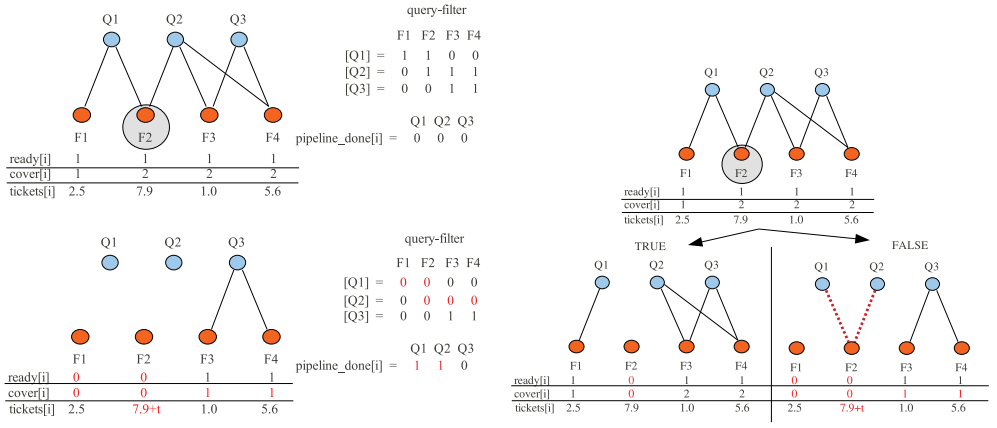
—based on the *query_cover* matrix, the *pipeline_done* is updated for all the pipelines that have been terminated (line 4), since the tuple did not pass;
—the *query_cover* rows for all terminated pipelines are cleared to denote that no other filters need to be evaluated for this pipeline, accordingly the *cover* needs to be updated for each filter (lines 43–45);
—*ready* vector is updated to denote that F_x has been evaluated (line 27).
—a number of tickets are granted to the filter for the future execution (line 22);

while; when a tuple passes the filter:

ALGORITHM 7: MRAF

```
1 class MRAF()
2   function initFilter(Filter f, int x)
3     filters[x] = f
4
5   function addFilter(int x, int q)
6     query_filter[q, x] = 1
7
8   function init(int N, int q)
9     filters[1..N] ← null
10    tickets[1..N] ← 0
11    query_filter[1..q, 1..N] ← 0
12
13  function evaluate(Key key,Value value)
14    ready[1..N] ← 1
15    pipeline_done[1..q] ← 0
16    initCover()
17
18    for k=1 to N do
19      x = chooseFilter()
20      if(filters[x](key,value) == false
21        updateStateInformationFalse(x)
22        tickets[x]+ = Δ+
23      else
24        updateStateInformationTrue(x)
25        tickets[x]+ = Δ−
26
27      ready[x] = 0
28
29    return result based on pipeline_done == 0
30
31  function chooseFilter()
32    return x ∈ [1..N] with probability proportional to tickets[x], such that ready[x]==1
33
34  function initCover()
35    for x=1 to N do
36      cover[x] = Σ_{k=1}^{q} query_filter[k, x]
37
38  function updateStateInformationFalse(int x)
39    for k=1 to q do
40      if query_cover[k, x] == 1 then
41      pipeline_done[k] = 1
42      for int l=1 to N
43        if query_cover[k,l]==1
44          query_cover[k,l]==0
45          cover[l]−
46
47  function updateStateInformationTrue(int x)
48    cover[x]=0
```

—only *ready* and *cover* vectors are updated, so the filter will not be chosen in future iterations;
—also in this case a number of tickets (possibly negative) can be granted to a filter (line 25).



(a) updating state information (b) QUERYCOVERTICKETING

Fig. 6. `MRAdaptiveFilter`.

When all filters have been evaluated and there are remaining entries set to 0 in *pipeline_done*, a map output tuple is emitted for the corresponding pipelines (line 29). Figure 6(a) presents the updates to the data structures when filter $F2$ is chosen and evaluates to false. Although only two queries contain $F2$, five edges are removed from the query filter graph.

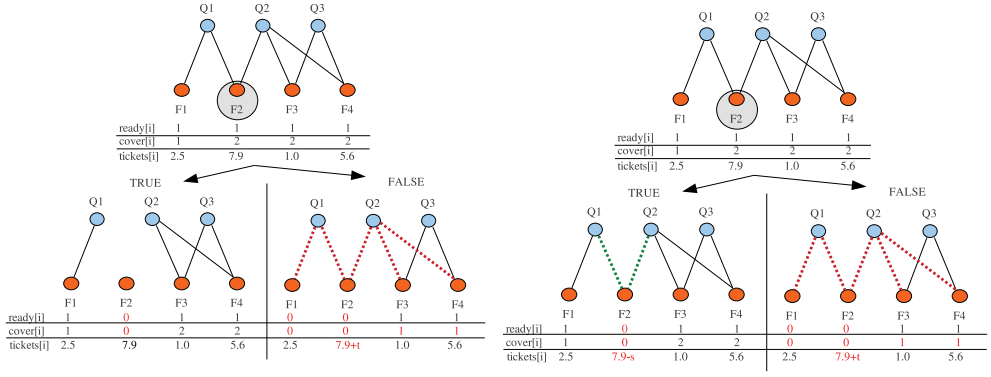
11.2.1. Ticketing Policies for Shared Execution. We propose ticketing policies resembling the QUERYCOVERGREEDY and EDGECOVERGREEDY heuristics, which are heuristic routing policies when the statistical properties of data are known. In our case, we utilize the rationale of both techniques for providing feedback for future execution when the statistical properties of data are unknown.

QueryCoverTicketing. The QUERYCOVERGREEDY approach attempts to choose filters that "cover" (evaluate to false) most queries per cost unit. In our solution we grant Δ^{+}_{QC} tickets to a filter when it discards a tuple, and $\Delta^{-}_{QC} = 0$ when a tuple passes the filter. Formally,

$$\Delta^{+}_{QC} = \frac{disc}{c_x}, \tag{25}$$

where $disc$ is the number of pipelines discarded by F_x, meaning that $cover[x] = disc$ *at the time* the filter is evaluated. Notice that $cover[x]$ can be updated as other filters are being evaluated, and hence Δ depends on when F_x was evaluated. We promote inexpensive filters that discard many pipelines, relying on the same rationale as the QUERYCOVERGREEDY heuristic. In Figure 6(b) we show the updates to the state information depending on the outcome of filter $F2$. In this example, t is equal to $\frac{2}{c_2}$.

EdgeCoverTicketing. Discarding a pipeline possibly decreases the need of evaluating other filters, as shown in the previous example. Based on the rationale of the EDGECOVERGREEDY strategy, we grant tickets based on the coverage of the filter in the graph. Assume that filter F_x occurs in $disc$ pipelines, hence $cover[x] = disc$, and it discards an input tuple. Also, assume that as a consequence $disc_e$ edges are removed from the query filter graph, which can be easily maintained (Algorithm 6, line 43). Then, the



(a) EDGECOVERTICKETING (b) HYBRIDTICKETING

Fig. 7. `MRAdaptiveFilter`ticketing policies.

filter is granted Δ^{+}_{EC} additional tickets.

$$\Delta^{+}_{EC} = \frac{disc_e}{c_x} \tag{26}$$

If the filter passes a tuple, it is granted $\Delta^{+}_{EC} = 0$ additional tickets. In the example in Figure 7(a), $disc_e$ is equal to 5 (red dotted edges).

HybridTicketing. Combining the QUERYCOVERTICKETING and EDGECOVERTICKETING policies, we propose a third policy, namely, HYBRIDTICKETING, that modifies the number of tickets both when a tuple passes or fails a filter. In the former case, we decrease the number of tickets for filter F_x that was passed by the tuple by

$$\Delta^{-}_{HC} = \frac{c_x}{disc}, \tag{27}$$

which is the reverse of Δ^{+}_{QC}, where $disc$ is the number of pipelines containing F_x. The more expensive the filter and the less edges it covers, the higher the penalty. On the other hand, when a filter drops an input tuple, it is granted tickets as in the EDGECOVERTICKETING. Hence

$$\Delta^{+}_{HC} = \frac{disc_e}{c_x}. \tag{28}$$

Figure 7(b) shows an example evaluation for filter $F2$, where s is equal to $\frac{c_2}{2}$, and t is equal to $\frac{5}{c_2}$.

OrderedTicketing Heuristic. Similar to single queries, we introduce a heuristic that modifies the number of assigned tickets based on the number of filters that the tuple passed before it was processed by the current filter. Namely, if F_x is the k^{th} filter to process a tuple, it is assigned $\frac{1}{k} \times X$, where X is the number of tickets corresponding to each of the three base ticketing policies. ORDEREDTICKETING favors filters that discard tuples early.

12. EMPIRICAL EVALUATION OF `MRADAPTIVEFILTER`

12.1. Evaluation Setup

We evaluated `MRAdaptiveFilter` in Hadoop using EC2 [Amazon 2006], as in the case of the base framework `MRShare`. Here, however, we used the medium-size instances, which offer 1.7GB of main memory and two virtual cores. Medium instances are known

to provide better performance isolation, and better repeatability than the small-size instances, which is crucial for accuracy of CPU measurements. We used 40 virtual machines.

Dataset. We used a 20GB synthetic dataset for better control of the input parameters. The split size, hence the amount of data processed by each map task, was set to 100MB. Each split contained about 1M rows, of variable random length. Each row was parsed into a single input tuple. We controlled the amount of data processed by each map task to make the size of the data approximately equal to obtain clear comparison among tasks. We used a synthetic dataset to be able to perform a more thorough analysis of our techniques for a variety of statistical data properties and obtain a clear analogy to experiments of Babu et al. [2004]. We executed star-join queries in MapReduce, commonly used in the data warehousing setting.

Queries. The input dataset served as the fact table A for the following type of joins.

```
SELECT    count(A.a)
FROM      A, B, C, ...
WHERE     A.k = B.k  AND  A.l = C.l  AND ...
GROUP BY  A.a
```

All star-join queries can be represented as a series of expensive filters applied to the fact table [Blanas et al. 2010; Afrati and Ullman 2010]. Such joins are executed in MapReduce by broadcasting the dimension tables to all processing nodes. Subsequently, for each dimension table an index is built, and used for filtering the input fact table in the map stage. We simulated the index probes by using synthetic filters, similarly to Babu et al. [2004], that compute an expensive operation and make a randomized selection decision with appropriate probabilities to match our desired statistical properties. The base cost of a filter was around 20 μs when measured in a standalone setting on EC2. For fairness across experiments we ensured that the filter outcome was determined by the contents of the input tuple. The filters were applied in the map stage of each job, to each input tuple. We use synthetic filters as we are primarily interested in time performance evaluation with respect to different filter selectivities, costs, and correlations among them.

Simulating filter properties. We tested all algorithms for both uncorrelated and correlated filters. The correlation was captured using the model used by Babu et al. [2004]. The N filters are divided into $\lceil N/\Gamma \rceil$ groups containing Γ filters each, where Γ is the *correlation factor*. Filters belonging to different groups are independent. Filters within each group are positively correlated and return the same result on 80% of input tuples. We utilized synthetic filters to be able to vary multiple characteristics. This setting has been commonly used in related research, as it is more likely to capture different data characteristics rather than concentrate on a particular application.

We have implemented the *Filter* class, which allows to abstract user-defined functions in terms of their cost and selectivity properties. A filter is defined in terms of its *randomSeed*, *cost*, *dropProbability*, and *correlationFactor*. *correlationFactor* specifies the correlation of filters with the same *randomSeed*. For example, $correlationFactor = 0.8$ means that two filters with the same *randomSeed* will return the same true/false result for 80% of the input tuples. This feature enables one to simulate correlated filters. Clearly, the unconditional filter selectivity is determined by the *dropProbability* parameter. This methodology for testing has been utilized before; see Babu et al. [2004] for further details.

Measurements. Since the filters were applied in the map stage of each job, we measured the map task execution times. The measurements were averaged over all map

Table VII. Evaluation Parameters

Parameter	Stable characteristics	Variable characteristics
number of filters	{2, 4, 6, 8, 10}	{2, 4, 6, 8, 10}
filter cost	random {1 − 4}	random {1 − 4}
filter selectivities	0.5	randomly altered every 10000 tuples
Γ	{1, 4}	{1, 4}

tasks belonging to each job. Notice that we concentrate on evaluating the per-task performance, not the performance of MapReduce itself. Since Hadoop can run at once up to M_{max} map tasks, the average map task performance depends primarily on the saturation of the MapReduce engine. In each case, the number of tasks running at once was set to default settings (i.e., two map and reduce tasks per node). In summary, each data point was obtained by processing 20GB of data by 200 concurrent map tasks. We measured the execution times including the startup and finalization of each task in Hadoop (i.e., reading the input, parsing, sorting the map output, etc.). In total, we executed approximately 1500 MapReduce jobs over the time of a few days. We were primarily interested in throughput for each technique, and hence for each experiment we compared the throughput of the baseline (normalized to 1) with other approaches.

12.2. Single-Query Experiments

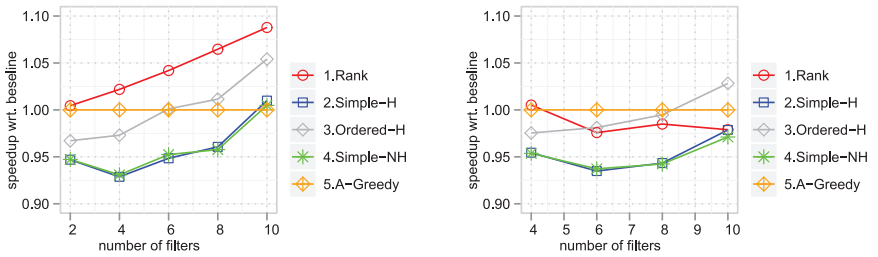
In this set of experiments, we were running singleton jobs using `MRShare`. Hence, the results do not include any benefits from sharing scans, nor sharing map output. The results show only additional benefits from expensive filter optimization introduced by `MRAdaptiveFilter`.

Compared approaches. For processing singleton jobs we compared the following approaches.

—*Rank*. The filters are ordered using their *rank*, assuming their independence. The selectivities of the filters are assumed to be known throughout the execution through an oracle (Rank).
—*`MRAdaptiveFilter` with SimpleTicketing*. This is with history inflation (Simple-H).
—*`MRAdaptiveFilter` with SimpleTicketing*. This is without history inflation (Simple-NH).
—*`MRAdaptiveFilter` with OrderedTicketing*. This is with history inflation (Ordered-H).
—*AGREEDY* [Babu et al. 2004] (baseline technique). The selectivities and the filter ordering are learned throughout the execution (AGreedy).

For the single-query case, we evaluated each algorithm in two scenarios, summarized in Table VII. For each experiment we varied the number of filters in the query. We generated 10 filters with random costs from the range of $\{1, \ldots, 4\}$, where 1 was equivalent to approximately 20 μs evaluation cost in a standalone setting. Recall that the processing speed is highly influenced by other factors (e.g., sorting), and there is a predicate cost lower bound below which there is no point in applying any reordering techniques. For each experiment we applied a number of filters to each query (see Table VII). For convergence experiments (stable filter selectivities), filter selectivities were constant over time and set to 0.5. For experiments with variable filter selectivities, filter selectivities were altered and set to a random value every 10000 input tuples. We normalized the performance of all other techniques by the performance of the baseline algorithm, namely, AGREEDY.

Stable filter characteristics. Figures 8(a) and 8(b) present our results for $\Gamma = 1$ and $\Gamma = 4$ respectively.



(a) uncorrelated filters $\Gamma = 1$ (b) correlated filters $\Gamma = 4$

Fig. 8. Single query: stable characteristics.

We plot the speedup with respect to AGREEDY as a function of the number of filters, for both uncorrelated and correlated filters, averaged over 200 map tasks. We emphasize that the curves do not necessarily need to be monotonic as the number of filters increases. This is due to the fact that, by increasing the number of filters, some new very expensive or less expensive filters can be added and therefore this can either improve or worsen the total running time.

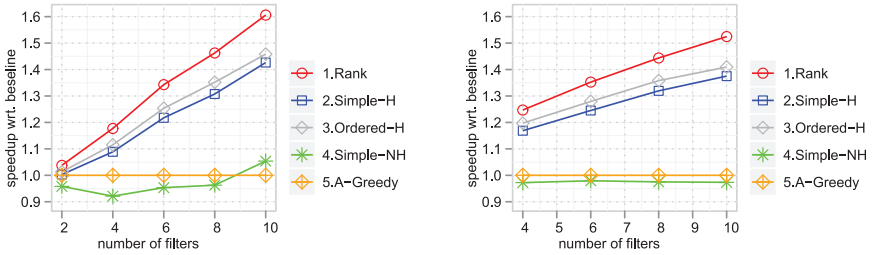
The results reveal several interesting observations. First, history inflation does not improve the performance of SIMPLETICKETING in `MRAdaptiveFilter` for the stable statistical characteristics. This is not surprising since the filter selectivities are constant over time, hence the burden of the history has no detrimental effect. Second, the performance of ORDEREDTICKETING is consistently better than the performance of SIMPLETICKETING, for both $\Gamma = 1$ and $\Gamma = 4$.

For the uncorrelated filters ($\Gamma = 1$), it can be observed that ORDEREDTICKETING stays in the range of 5% from the baseline approach. For larger number of filters, ORDEREDTICKETING outperforms AGREEDY. This is due to the fact that AGREEDY is introducing overhead by evaluating more filters that need not be evaluated. Notice that this is the optimal setting for the AGREEDY approach and is usually not true when processing ad hoc data; in the presence of stable filter characteristics it converges very fast to optimal ordering, and applies the same order of filters to each input tuple. In other words, the throughput is lower with respect to rank only because of the overhead of maintaining conditional probabilities. `MRAdaptiveFilter`, on the other hand, applies a different order of filters to each input tuple, hence uses suboptimal ordering. Furthermore, it can be observed that ORDEREDTICKETING follows the trend of the optimal ordering determined by RANK, staying in the range of 6% from the optimal ordering.

For the case of correlated filters with $\Gamma = 4$, the performance of `MRAdaptiveFilter` with ORDEREDTICKETING is very close to AGREEDY, and for greater number of filters it outperforms AGREEDY. Of course, RANK is not the optimal ordering in this case. Also in this case, the history inflation does not improve the results for the same reason as before. The experiments demonstrate that `MRAdaptiveFilter` is fully applicable in the presence of stable filter selectivities, and matches the performance of the state-of-the art techniques.

Variable filter characteristics. Now, we investigate a more common scenario where the statistical properties of the data (hence filter selectivities) change over time. In this set of experiments, we varied the filter selectivities as the tuples were processed; this variability is commonly present in ad hoc data. Results are shown in Figures 9(a) and 9(b) for respective values of Γ. The processing rate is the average over 200 map tasks.

First, we observe that `MRAdaptiveFilter` with ORDEREDTICKETING is consistently faster than `MRAdaptiveFilter` with SIMPLETICKETING and history inflation, and is the



(a) uncorrelated filters $\Gamma = 1$ (b) correlated filters $\Gamma = 4$

Fig. 9. Single query: variable characteristics.

best among our proposed policies. MRAdaptiveFilter with OrderedTicketing yields similar trends as in the previous case. For all cases, it follows the optimal Rank ordering and stays within the range of 10%. The speedup with respect to the baseline AGreedy reaches 45%. This trend can be observed for both values of Γ.

Observe also that SimpleTicketing without history inflation yields similar results as AGreedy. This is due to the fact that both algorithms adapt slowly to changing filter selectivities, as they are strongly burdened with history. Similar behavior can be observed for both uncorrelated ($\Gamma = 1$) and correlated filters ($\Gamma = 4$).

In principle, setting appropriate parameters for AGreedy (e.g., profiling probability) could possibly yield better results. However, this assumes knowledge of statistical properties of the data and filters in advance. MRAdaptiveFilter, on the other hand, does not rely on this assumption, rather it is capable of adapting to changing conditions.

We conclude that the MRAdaptiveFilter approach has strong experimental support for adaptive filter ordering in the case of a single query. It adapts quickly to changing filter selectivities. For the uncorrelated filters, we demonstrated that it stays within a close range of the optimal ordering. For the correlated case, we have shown that it significantly outperforms the AGreedy approach. At the same time, MRAdaptiveFilter matches the performance of other approaches for stable filter selectivities.

12.3. Multiple Queries

Approaches compared. As in the case of single queries, we evaluated MRAdaptiveFilter in a Hadoop environment. We ran a simulation of multiple star-join queries using MRShare, which allows for evaluation of a group of queries as a single job. We emphasize that here we were executing all queries in a single batch, to test *additional* savings when sharing expensive map computation; the results do not take into account the benefits of sharing the scan of the input. For the same reason, the map output of the queries was not shared, which would interfere with our measurements. The queries, as before, consist of expensive filters in the map stage, which simulate the index probes when executing star-joins. However here, some filters (index probes) were shared across the queries. We compared the performance of the following approaches:

—MRAdaptiveFilter. QueryCoverTicketing (MRAF-QC);
—MRAdaptiveFilter. EdgeCoverTicketing (MRAF-EC);
—MRAdaptiveFilter. HybridTicketing (MRAF-HC);
—MRAdaptiveFilter. HybridTicketing with OrderedTicketing heuristic (MRAF-HC-OT); and
—EdgeCoverGreedy (ECG-S). (baseline technique).

Baseline technique details – (ECG-S). For the multiple-query case, we compare MRAdaptiveFilter to the state-of-the-art approach for handling multiple queries,

Table VIII. Evaluation Parameters

Parameter	Stable characteristics	Variable characteristics
filters per query	{2, 4, 6, 8, 10}	{2, 4, 6, 8, 10}
filter cost	random {1, 2, 3, 4}	random {1, 2, 3, 4}
filter selectivities	0.5	randomly altered every 10000 tuples
Γ	{1, 4}	{1, 4}
history inflation	enabled	enabled

namely EdgeCoverGreedy [Liu et al. 2008b], which assumes knowledge of statistical properties of the filters and the data. We have implemented an extension of the EdgeCoverGreedy technique, inspired by the AGreedy approach. Our implementation is similar to AGreedy, however, it maintains only unconditional filter drop probabilities by executing all filters on a small sample of the input data.

We maintain a *profiling window* of size $windowSize$ of the input tuples, and estimate the selectivities of the filters based on the tuples within that window. Each tuple within the profiling window is associated with a $profileVector$ of true/false values for each of the n filters. For every input tuple, we insert the tuple into the profiling window with probability $profilingProbability$. When a new tuple is inserted, we remove the least recently inserted tuple from the window, execute *all* filters for the newly inserted tuple, and create its $profileVector$. Subsequently, all $profileVector$s of all the tuples within the window are used to estimate filter selectivities.

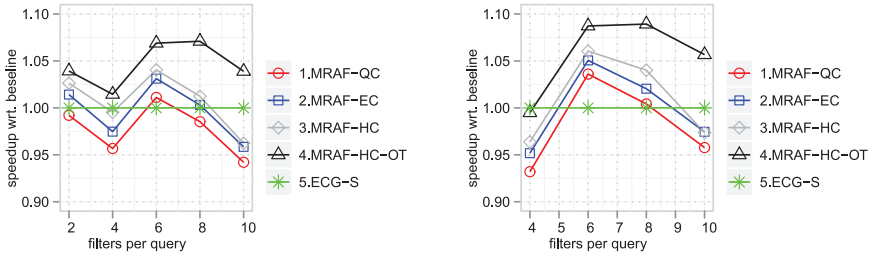
Clearly, the higher the $profilingProbability$, the more accurate statistics we can obtain, especially in the context of time-varying filter characteristics. However, increasing this parameter introduces overhead. Similarly, there is a trade-off between different values of $windowSize$. Larger window sizes increase the estimation accuracy, however, they introduce more profiling overhead. Equipped with this selectivity estimation scheme, we no longer need explicit knowledge of the filter selectivities. The estimated selectivities are then used to apply the EdgeCoverGreedy heuristic [Liu et al. 2008b]. We compare this sample-based approach to our solution.

The filter selectivities were obtained by keeping a window of $windowSize = 1000$ sampled input tuples. Each input tuple was inserted into the window with $profilingProbability = 0.02$. We set the parameters empirically. Further details about choosing such parameters can be found in Babu et al. [2004].

Parameters. We tested the performance using eight queries with variable number of filters per query. We generated filters with random costs, and for each experiment we assigned various randomly chosen filters to each query. The filters' "popularity" followed the Zipfian 1.00 distribution. We used uncorrelated and correlated filters. Similarly, we compared the algorithms for stable and variable filter selectivities. Each experiment was executed with history inflation enabled, since the single-query experiments confirmed its utility. The parameters are summarized in Table VIII.

Stable filter characteristics: Figures 10(a) and 10(b) show the performance of all other techniques normalized by the baseline, that is, EdgeCoverGreedy (ECG-S).

In both cases, `MRAdaptiveFilter` with HybridTicketing (MRAF-HC) yields the highest processing rates among our basic ticketing policies. The difference between MRAF-HC and our two other basic ticketing policies, MRAF-QC and MRAF-EC, was up to 5%. All three basic policies, however, process tuples at similar speed as the baseline (ECG-S), yielding processing rates within a range of 5% with respect to the baseline. In our experiments, we tuned the parameters empirically to obtain the highest throughput for our baseline technique. For the case of stable filter characteristics it is relatively easy.



(a) uncorrelated filters $\Gamma = 1$ (b) correlated filters $\Gamma = 4$

Fig. 10. Multiple queries: stable characteristics.

However, choosing the appropriate sampling technique to obtain accurate statistics for ad hoc data is not an easy task. Observe that using the OrderedTicketing heuristic (MRAF-HC-OT), is better by 10% than MRAF-HC and that the best technique consistently outperforms the baseline technique with a gain that in some cases is 10%. We conclude that `MRAdaptiveFilter` is fully applicable for stable filter characteristics. Even though in such scenarios it is easy to obtain statistical information about the underlying data, our best ticketing techniques still outperform the sampling-based approach.

Variable filter characteristics. Figures 11(a) and 11(b) show our results for variable filter selectivities.

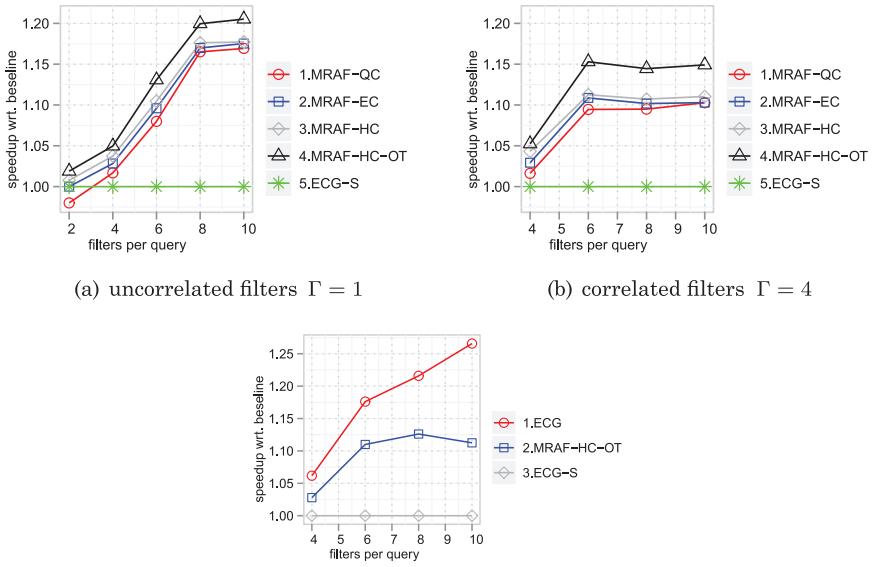
Consistently, the `MRAdaptiveFilter` approach with HybridTicketing (MRAF-HC) yields the highest processing rates among our basic ticketing policies, with the difference reaching 5%. The OrderedTicketing heuristic improves the performance of the best basic policy by 5%. In accordance with our intuition in Section 10, OrderedTicketing yields higher improvements for the correlated filters.

We can observe that the best policy of `MRAdaptiveFilter` consistently outperforms the sample-based approach in all cases, yielding up to 20% higher processing rates for uncorrelated filters and up to 15% for correlated filters. The gap between the sampling-based approach and `MRAdaptiveFilter` increases as with the number of filters. The baseline, the same as in the single-query case, is burdened with history, which causes its performance to degrade.

Comparison to an oracle-based approach. In the last set of experiments, we compared our best ticketing policy with an oracle-based EdgeCoverGreedy implementation. This oracle-based approach assumes that the filter selectivities are known at any time and applies the EdgeCoverGreedy heuristic. Hence, the entire overhead for sampling is avoided. Figure 11(c) presents the results for a random mix of queries with stable and variable filter characteristics, and also different values of the correlation factor, aggregated in all previous experiments. We plot three approaches: EdgeCoverGreedy with an oracle, EdgeCoverGreedy with sampling, and the `MRAdaptiveFilter` with HybridTicketing and OrderedTicketing heuristic.

We observe that, despite that the sampling rate was hand-tuned to achieve the highest throughput, the sampled-based EdgeCoverGreedy approach introduces a substantial overhead. The `MRAdaptiveFilter` approach, with the best ticketing policy, halves the gap between the oracle-based approach and the sampled-based one. Hence, we conclude that in the absence of statistical knowledge, `MRAdaptiveFilter` is the preferred technique.

Discussion. In this section we presented an extensive experimental study of various techniques for adaptive filter ordering in MapReduce. In particular, we evaluated the



(a) uncorrelated filters $\Gamma = 1$

(b) correlated filters $\Gamma = 4$

(c) overall comparison

Fig. 11. Multiple queries: variable characteristics and overall comparison.

MRAdaptiveFilter extension of MRShare for handling queries with expensive filters. MRAdaptiveFilter is primarily designed for handling ad hoc data with variable statistical properties. However, it performs at least as well as its competitors even in the case of stable statistical characteristics. Furthermore, MRAdaptiveFilter is parameter free. At the same time, it is much more flexible, as it allows for handling single and multiple queries in a unified manner. We have shown that MRAdaptiveFilter can yield tremendous savings in addition to MRShare when utilized for jobs/groups of jobs with expensive filters.

We have presented evidence that the appropriate ticketing policies can yield significant improvements. Our setup allows for having multiple ticketing policies within one job across different map tasks. The ticketing policies can be dynamically changed within a given task. Finally, we note that, although we applied MRAdaptiveFilter on the map side of each job, it is applicable on the reduce side as well.

Our experiments on EC2 utilized approximately 3000 machine hours, with a cost of $600. Introducing each percent of savings in the execution time would translate into $6, which even for our experimental setup is nonnegligible.

13. CONCLUSION AND FUTURE WORK

This article described MRShare, a principled analysis for automatic work sharing across multiple MapReduce jobs. Utilizing the fact that, in batch job processing, jobs overlap, we merge them appropriately. The merged jobs carry less redundancy and can be executed more efficiently. 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. We also studied the problem of expensive filter ordering for processing jobs in MRShare. We proposed MRAdaptiveFilter, an extension of MRShare for adaptive filter ordering tailored to MapReduce designed primarily for scenarios demanding high adaptivity. Our lightweight Eddies-based framework enables efficient expensive filter

evaluation for both single- and batch-query execution mode, which in turn minimizes the CPU overhead of the system. Central to our proposal are the `MRAdaptiveFilter` ticketing policies. We introduced two ticketing policies for the case of a single query, namely, *simple ticketing* and *ordered ticketing*, together with a *history inflation* heuristic enabling fast adaptivity of our solution. For the case of shared execution of multiple queries, we implemented three ticketing policies: *QueryCover Ticketing*, *EdgeCover Ticketing*, and *Hybrid Ticketing*. We evaluated `MRShare` and `MRAdaptiveFilter` experimentally and demonstrated the benefits of applying adaptive filter ordering strategies in the MapReduce environment. Our experiments on Amazon EC2 demonstrated that our approach yields significant savings.

REFERENCES

Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Alexander Rasin. 2009. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proc. VLDB Endow.* 2, 1, 922–933.

Foto Afrati and Jeffrey D. Ullman. 2010. Optimizing joins in a MapReduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT'10).* 99–110.

Parag Agrawal, Daniel Kifer, and Christopher Olston. 2008. Scheduling shared scans of large data files. *Proc. VLDB Endow.* 1, 1, 958–969.

Amazon. 2006. Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'00).* 261–272.

Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive ordering of pipelined stream filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD'04)*. 407–418.

Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. 2010. A comparison of join algorithms for log processing in MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD'10)*. 975–986.

Blogscope. 2005. BlogScope. http://www.blogscope.net/.

George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A scalable, predictable join operator for highly concurrent data warehouses. *Proc. VLDB Endow.* 2, 1, 277–288.

Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow* 1, 2, 1265–1276.

Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR'03).*

Surajit Chaudhuri and Kyuseok Shim. 1993. Query optimization in the presence of foreign functions. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB'93).* 529–542.

Surajit Chaudhuri and Kyuseok Shim. 1996. Optimization of queries with user-defined predicates. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB'96).* 87–98.

Surajit Chaudhuri and Kyuseok Shim. 1999. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.* 24, 2, 177–228.

Fa-Chung Fred Chen and Margaret H. Dunham. 1998. Common subexpression processing in multiple-query processing. *IEEE Trans. Knowl. Data Engin.* 10, 3, 493–499.

Jianjun Chen, David J. Dewitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'00).* 379–390.

Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, Yuan Yuan Yu, Gary Bradski, Andrew Y. Ng, and Kunie Olukotun. 2006. MapReduce for machine learning on multicore. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS'06).*

Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. 2009. MAD skills: New analysis practices for big data. *Proc. VLDB Endow.* 2, 2, 1481–1492.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI'04)*. 107–113.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Comm. ACM* 51, 1, 107–113.

Jens Dittrich, Jorge Quiane, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jorg Schad. 2010. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.* 3, 1.

Sheldon Finkelstein. 1982. Common expression analysis in database applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD'82)*. 235–245.

Eric Friedman, Peter Pawlowski, and John Cieslewicz. 2009. Sql/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.* 2, 2.

Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. 2009. Building a high-level dataflow system on top of MapReduce: The pig experience. *Proc. VLDB Endow.* 2, 2, 1414–1425.

Hadoop. 2007. Hadoop project. http://hadoop.apache.org/.

Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: A simultaneously pipelined relational query engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'00).* 383–394.

Joseph M. Hellerstein. 1994. Practical predicate placement. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD'94)*. 325–335.

Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate migration: Optimizing queries with expensive predicates. *ACM SIGMOD Rec.* 22, 2, 267–276.

Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A self-tuning system for big data analytics. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR'11).*

Eaman Jahani, Michael J. Cafarella, and Christopher Re. 2011. Automatic optimization for MapReduce programs. *Proc. VLDB Endow.* 4, 6.

Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. 2010. The performance of MapReduce: An in-depth study. *Proc. VLDB Endow.* 3, 1.

Ryan Johnson, Stavros Harizopoulos, Nikos Hardavellas, Kivanc Sabirli, Ippokratis Pandis, Anastassia Ailamaki, Naju G. Mancheril, and Babak Falsafi. 2007. To share or not to share? In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07).* 351–362.

Boduo Li, Edward Mazur, Yanlei Diao, Andrew Mcgregor, and Prashant J. Shenoy. 2012. SCALLA: A platform for scalable one-pass analytics using MapReduce. *ACM Trans. Database Syst.* 37, 4, 27.

Zhen Liu, Srinivasan Parthasarathy, Anand Ranganathan, and Hao Yang. 2008a. A generic flow algorithm for shared filter ordering problems. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'08).* ACM Press, New York, 79–88.

Zhen Liu, Srinivasan Parthasarathy, Anand Ranganathan, and Hao Yang. 2008b. Near-optimal algorithms for shared filter evaluation in data stream systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08).* ACM Press, New York, 133–146.

Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'02).* 49–60.

Kamesh Munagala, Utkarsh Srivastava, and Jennifer Widom. 2007. Optimization of continuous queries with shared expensive filters. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'07).* 215–224.

Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: Sharing across multiple queries in MapReduce. *Proc. VLDB Endow.* 3, 1, 494–505.

Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. 2008a. Automatic optimization of parallel dataflow programs. In *Proceedings of the Annual Technical Conference on Annual Technical Conference (ATC'08).* 267–273.

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008b. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08).* 1099–1110.

Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. 2009. PLANET: Massively parallel learning of tree ensembles with mapreduce. *Proc. VLDB Endow.* 2, 2.

Jooseok Park and Arie Segev. 1988. Using common subexpressions to optimize multiple queries. In *Proceedings of the 4th International Conference on Data Engineering (ICDE'88).* 311–319.

Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. Dewitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD'09)*.

Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.* 13, 4, 277–298.

Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. 2008. Main-memory scan sharing for multi-core CPUS. *Proc. VLDB Endow.* 1, 1, 610–621.

Daniel J. Rosenkrantz and Harry B. Hunt III. 1980. Processing conjunctive predicates and queries. In *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB'80).* 64–72.

Prasan Roy, Sridhar Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. *ACM SIGMOD Rec.* 29, 2, 249–260.

Timos Sellis. 1988. Multiple-query optimization. *ACM Trans. Database Syst.* 13, 1, 23–52.

Kyuseok Shim, Timos Sellis, and Dana Nau. 1994. Improvements on a heuristic algorithm for multiple-query optimization. *Data Knowl. Engin.* 12, 2, 197–222.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive – A warehousing solution over a MapReduce framework. *Proc. VLDB Endow.* 2, 2, 1626–1629.

Carl A. Waldspurger and William E. Weihl. 1994. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI'94).* 1–11.

Xiaodan Wang, Christopher Olston, Anish Das Sarma, and Randal Burns. 2011. CoScan: Cooperative scan sharing in the cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11).* 11:1–11:12.

Joel Wolf, Andrey Balmin, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Sujay Parekh, Kun-Lung Wu, and Rares Vernica. 2012. CIRCUMFLEX: A scheduling optimizer for MapReduce workloads with shared scans. *SIGOPS Oper. Syst. Rev.* 46, 1, 26–32.

Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. 2011. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11).* 12:1–12:13.

Hung-Chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. 2007. MapReduce-merge: Simplified relational data processing on large clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07).* 1029–1040.

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08).* 1–14.

Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. 2007. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD'07)*. 533–544.

Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2007. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07).* 723–734.

Received August 2012; revised September 2013; accepted December 2013