Minimal MapReduce Algorithms

Yufei Tao[1,2] Wenqing Lin[3] Xiaokui Xiao[3]

[1]Chinese University of Hong Kong, Hong Kong
[2]Korea Advanced Institute of Science and Technology, Korea
[3]Nanyang Technological University, Singapore

ABSTRACT

MapReduce has become a dominant parallel computing paradigm for *big data*, i.e., colossal datasets at the scale of tera-bytes or higher. Ideally, a MapReduce system should achieve a high degree of load balancing among the participating machines, and minimize the space usage, CPU and I/O time, and network transfer at each machine. Although these principles have guided the development of MapReduce algorithms, limited emphasis has been placed on enforcing serious constraints on the aforementioned metrics simultaneously. This paper presents the notion of *minimal algorithm*, that is, an algorithm that guarantees the best parallelization in multiple aspects at the same time, up to a small constant factor. We show the existence of elegant minimal algorithms for a set of fundamental database problems, and demonstrate their excellent performance with extensive experiments.

Categories and Subject Descriptors
F2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems

Keywords
Minimal algorithm, MapReduce, big data

1. INTRODUCTION

We are in an era of information explosion, where industry, academia, and governments are accumulating data at an unprecedentedly high speed. This brings forward the urgent need of *big data processing*, namely, fast computation over colossal datasets whose sizes can reach the order of tera-bytes or higher. In recent years, the database community has responded to this grand challenge by building massive parallel computing platforms which use hundreds or even thousands of commodity machines. The most notable platform, which has attracted a significant amount of research attention, is *MapReduce*.

Since its invention [16], MapReduce has gone through years of improvement into a mature paradigm (see Section 2 for a review). At a high level, a MapReduce system involves a number of share-nothing machines which communicate only by sending messages over the network. A MapReduce algorithm instructs these machines to perform a computational task collaboratively. Initially, the input dataset is distributed across the machines, typically in a non-replicate manner, i.e., each object on one machine. The algorithm executes in *rounds* (sometimes also called *jobs* in the literature), each having three phases: *map*, *shuffle*, and *reduce*. The first two enable the machines to exchange data: in the map phase, each machine prepares the information to be delivered to other machines, while the shuffle phase takes care of the actual data transfer. No network communication occurs in the reduce phase, where each machine performs calculation from its local storage. The current round finishes after the reduce phase. If the computational task has not completed, another round starts.

Motivation. As with traditional parallel computing, a MapReduce system aims at a high degree of load balancing, and the minimization of space, CPU, I/O, and network costs at each individual machine. Although these principles have guided the design of MapReduce algorithms, the previous practices have mostly been on a best-effort basis, paying relatively less attention to enforcing serious constraints on different performance metrics. This work aims to remedy the situation by studying algorithms that promise outstanding efficiency in multiple aspects simultaneously.

Minimal MapReduce Algorithms. Denote by S the set of input objects for the underlying problem. Let n, the *problem cardinality*, be the number of objects in S, and t be the number of machines used in the system. Define $m = n/t$, namely, m is the number of objects per machine when S is evenly distributed across the machines. Consider an algorithm for solving a problem on S. We say that the algorithm is *minimal* if it has all of the following properties.

- *Minimum footprint:* at all times, each machine uses only $O(m)$ space of storage.
- *Bounded net-traffic:* in each round, every machine sends and receives at most $O(m)$ words of information over the network.
- *Constant round:* the algorithm must terminate after a constant number of rounds.
- *Optimal computation:* every machine performs only $O(T_{seq}/t)$ amount of computation *in total* (i.e., summing over all rounds), where T_{seq} is the time needed to solve the same problem on a single sequential machine. Namely, the algorithm should achieve a speedup of t by using t machines in parallel.

It is fairly intuitive why minimal algorithms are appealing. First, *minimum footprint* ensures that, each machine keeps $O(1/t)$ of the

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

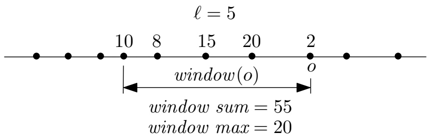


Figure 1: Sliding aggregates

dataset S at any moment. This effectively prevents *partition skew*, where some machines are forced to handle considerably more than m objects, as is a major cause of inefficiency in MapReduce [36].

Second, *bounded net-traffic* guarantees that, the shuffle phase of each round transfers at most $O(m \cdot t) = O(n)$ words of network traffic overall. The duration of the phase equals roughly the time for a machine to send and receive $O(m)$ words, because the data transfers to/from different machines are in parallel. Furthermore, this property is also useful when one wants to make an algorithm *stateless* for the purpose of fault tolerance, as discussed in Section 2.1.

The third property *constant round* is not new, as it has been the goal of many previous MapReduce algorithms. Importantly, this and the previous properties imply that there can be only $O(n)$ words of network traffic during the *entire* algorithm. Finally, *optimal computation* echoes the very original motivation of MapReduce to accomplish a computational task t times faster than leveraging only one machine.

Contributions. The core of this work comprises of neat minimal algorithms for two problems:

Sorting. The input is a set S of n objects drawn from an ordered domain. When the algorithm terminates, all the objects must have been distributed across the t machines in a sorted fashion. That is, we can order the machines from 1 to t such that all objects in machine i precede those in machine j for all $1 \leq i < j \leq t$.

Sliding Aggregation. The input includes

- a set S of n objects from an ordered domain, where every object $o \in S$ is associated with a numeric *weight*
- an integer $\ell \leq n$
- and a distributive aggregate function AGG (e.g., *sum*, *max*, *min*).

Denote by $window(o)$ the set of ℓ largest objects in S not exceeding o. The *window aggregate* of o is the result of applying AGG to the weights of the objects in $window(o)$. The *sliding aggregation problem* is to report the window aggregate of every object in S.

Figure 1 illustrates an example where $\ell = 5$. Each black dot represents an object in S. Some relevant weights are given on top of the corresponding objects. For the object o as shown, its window aggregate is 55 and 20 for AGG = *sum* and *max*, respectively.

The significance of sorting is obvious: a minimal algorithm for this problem leads to minimal algorithms for several fundamental database problems, including *ranking*, *group-by*, *semi-join* and *skyline*, as we will discuss in this paper.

The importance of the second problem probably deserves a bit more explanation. Sliding aggregates are crucial in studying time series. For example, consider a time series that records the Nasdaq index in history, with one value per minute. It makes good senses to examine *moving statistics*, that is, statistics aggregated from a sliding window. For example, a *6-month average/maximum* with respect to a day equals the average/maximum Nasdaq index in a 6-month period ending on that very day. The 6-month averages/maximums of *all* days can be obtained by solving a sliding aggregation problem (note that an average can be calculated by dividing a window sum by the period length ℓ).

Sorting and sliding aggregation can both be settled in $O(n \log n)$ time on a sequential computer. There has been progress in developing MapReduce algorithms for sorting. The state of the art is *TeraSort* [50], which won the Jim Gray's benchmark contest in 2009. *TeraSort* comes close to being minimal when a crucial parameter is set appropriately. As will be clear later, the algorithm requires manual tuning of the parameter, an improper choice of which can incur severe performance penalty. Sliding aggregation has also been studied in MapReduce by Beyer et al. [6]. However, as explained shortly, the algorithm is far from being minimal, and is efficient only when the window length ℓ is short – the authors of [6] commented that this problem is "notoriously difficult".

Technical Overview. This work was initialized by an attempt to justify theoretically why *TeraSort* often achieves excellent sorting time with only 2 rounds. In the first round, the algorithm extracts a random sample set S_{samp} of the input S, and then picks $t - 1$ sampled objects as the *boundary objects*. Conceptually, these boundary objects divide S into t segments. In the second round, each of the t machines acquires all the objects in a distinct segment, and sorts them. The size of S_{samp} is the key to efficiency. If S_{samp} is too small, the boundary objects may be insufficiently scattered, which can cause partition skew in the second round. Conversely, an over-sized S_{samp} entails expensive sampling overhead. In the standard implementation of *TeraSort*, the sample size is left as a parameter, although it always seems to admit a good choice that gives outstanding performance [50].

In this paper, we provide rigorous explanation for the above phenomenon. Our theoretical analysis clarifies how to set the size of S_{samp} to guarantee the minimality of *TeraSort*. In the meantime, we also remedy a conceptual drawback of *TeraSort*. As elaborated later, strictly speaking, this algorithm does not fit in the MapReduce framework, because it requires that (besides network messages) the machines should be able to communicate by reading/writing a common (distributed) file. Once this is disabled, the algorithm requires one more round. We present an elegant fix so that the algorithm still terminates in 2 rounds even by strictly adhering to MapReduce. Our findings of *TeraSort* have immediate practical significance, given the essential role of sorting in a large number of MapReduce programs.

Regarding sliding aggregation, the difficulty lies in that ℓ is not a constant, but can be any value up to n. Intuitively, when $\ell \gg m$, $window(o)$ is so large that the objects in $window(o)$ cannot be found on one machine under the *minimum footprint* constraint. Instead, $window(o)$ would potentially span many machines, making it essential to coordinate the searching of machines judiciously to avoid a disastrous cost blowup. In fact, this pitfall has captured the existing algorithm of [6], whose main idea is to ensure that *every* sliding window be sent to a machine for aggregation (various windows may go to different machines). This suffers from prohibitive communication and processing cost when the window length ℓ is long. Our algorithm, on the other hand, achieves minimality with a novel idea of *perfectly* balancing the input objects across the machines while still maintaining their sorted order.

Outline. Section 2 reviews the previous work related to ours. Section 3 analyzes *TeraSort* and modifies it into a minimal algorithm, which Section 4 deploys to solve a set of fundamental problems minimally. Section 5 gives our minimal algorithm for the sliding aggregation problem. Section 6 evaluates the practical efficiency of the proposed techniques with extensive experiments. Finally, Section 7 concludes the paper with a summary of findings.

2. PRELIMINARY AND RELATED WORK

In Section 2.1, we expand the MapReduce introduction in Section 1 with more details to pave the way for our discussion. Section 2.2 reviews the existing studies on MapReduce, while Section 2.3 points out the relevance of minimal algorithms to the previous work.

2.1 MapReduce

As explained earlier, a MapReduce algorithm proceeds in *rounds*, where each round has three phases: *map*, *shuffle*, and *reduce*. As all machines execute a program in the same way, next we focus on one specific machine $\mathcal{M}$.

Map. In this phase, $\mathcal{M}$ generates a list of key-value pairs (k, v) from its local storage. While the key k is usually numeric, the value v can contain arbitrary information. As clarified shortly, the pair (k, v) will be transmitted to another machine in the shuffle phase, such that the recipient machine is determined *solely* by k.

Shuffle. Let L be the list of key-value pairs that all the machines produced in the map phase. The shuffle phase distributes L across the machines adhering to the constraint that, pairs with the same key must be delivered to the same machine. That is, if $(k, v_1), (k, v_2), ..., (k, v_x)$ are the pairs in L having a common key k, all of them will arrive at an identical machine.

Reduce. $\mathcal{M}$ incorporates the key-value pairs received from the previous phase into its local storage. Then, it carries out whatever processing as needed on its local data. After all machines have completed the reduce phase, the current round terminates.

Discussion. It is clear from the above that, the machines communicate only in the shuffle phase, whereas in the other phases each machine executes the algorithm sequentially, focusing on its own storage. Overall, parallel computing happens mainly in reduce. The major role of map and shuffle is to swap data among the machines, so that computation can take place on different combinations of objects.

Simplified View for Our Algorithms. Let us number the t machines of the MapReduce system arbitrarily from 1 to t. In the map phase, all our algorithms will adopt the convention that $\mathcal{M}$ *generates a key-value pair* (k, v) *if and only if it wants to send* v *to machine* k. In other words, the key field is explicitly the id of the recipient machine.

This convention admits a conceptually simpler modeling. In describing our algorithms, we will combine the map and shuffle phases into one called *map-shuffle*. By saying succinctly that "*in the map-shuffle phase,* $\mathcal{M}$ *delivers* v *to machine* k", we mean that $\mathcal{M}$ creates (k, v) in the map phase, which is then transmitted to machine k in the shuffle phase. The equivalence also explains why the simplification is only at the logical level, while physically all our algorithms are still implemented in the standard MapReduce paradigm.

Statelessness for Fault Tolerance. Some MapReduce implementations (e.g., Hadoop) place the requirement that, at the end of a round, each machine should send all the data in its storage to a *distributed file system* (DFS), which in our context can be understood as a "disk in the cloud" that guarantees consistent storage (i.e., it never fails). The objective is to improve the system's robustness in the scenario where a machine collapses during the algorithm's execution. In such a case, the system can replace this machine with another one, ask the new machine to load the storage of the old machine at the end of the previous round, and re-do the current round (where the machine failure occurred). Such a system is called *stateless* because intuitively no machine is responsible for remembering any state of the algorithm [58].

The four minimality conditions defined in Section 1 ensure efficient enforcement of statelessness. In particular, *minimum footprint* guarantees that, at each round, every machine sends $O(m)$ words to the DFS, as is still consistent with *bounded traffic*.

2.2 Previous Research on MapReduce

The existing investigation on MapReduce can be coarsely classified into two categories, which focus on improving the internal working of the framework, and employing MapReduce to solve concrete problems, respectively. In the sequel, we survey each category separately.

Framework Implementation. *Hadoop* is perhaps the most popular open-source implementation of MapReduce nowadays. It was first described by Abouzeid et al. [1], and has been improved significantly by the collective findings of many studies. Specifically, Dittrich et al. [18] provided various user-defined functions that can substantially reduce the running time of MapReduce programs. Nykiel et al. [47], Elghandour and Agoulnaga [19] achieved further performance gains by allowing a subsequent round of an algorithm to re-use the outputs of the previous rounds. Eltabakh et al. [20] and He et al. [27] discussed the importance of keeping relevant data at the same machine in order to reduce network traffic. Floratou et al. [22] presented a column-based implementation and demonstrated superior performance in certain environments. Shinnar et al. [53] proposed to eliminate disk I/Os by fitting data in memory as much as possible. Gufler et al. [26], Kolb et al. [33], and Kwon et al. [36] designed methods to rectify *skewness*, i.e., imbalance in the workload of different machines.

Progress has been made towards building an *execution optimizer* that can automatically coordinate different components of the system for the best overall efficiency. The approach of Herodotou and Babu [28] is based on profiling the cost of a MapReduce program. Jahani et al. [29] proposed a strategy that works by analyzing the programming logic of MapReduce codes. Lim et al. [40] focused on optimizing as a whole multiple MapReduce programs that are interconnected by a variety of factors.

There has also been development of administration tools for MapReduce systems. Lang and Patel [37] suggested strategies for minimizing energy consumption. Morton et al. [46] devised techniques for estimating the progress (in completion percentage) of a MapReduce program. Khoussainova et al. [32] presented a mechanism to facilitate the debugging of MapReduce programs.

MapReduce, which after all is a computing framework, lacks many features of a database. One, in particular, is an expressive language that allows users to describe queries supportable by MapReduce. To fill this void, a number of languages have been designed, together with the corresponding *translators* that convert a query to a MapReduce program. Examples include SCOPE [9], Pig [49], Dremel [43], HIVE [55], Jaql [6], Tenzing [10], and SystemML [24].

Algorithms on MapReduce. Considerable work has been devoted to processing joins on relational data. Blanas et al. [7] compared the implementations of traditional join algorithms in MapReduce. Afrati and Ullman [3] provided specialized algorithms for multiway equi-joins. Lin et al. [41] tackled the same problem utilizing column-based storage. Okcan and Riedewald [48] devised algorithms for reporting the cartesian product of two tables. Zhang et al. [62] discussed efficient processing of multiway theta-joins.

Regarding joins on non-relational data, Vernica et al. [59], Metwally and Faloutsos [44] studied *set-similarity join*. Afrati et al. [2] re-visited this problem and its variants under the constraint that an algorithm must terminate in a single round. Lu et al. [42], on the other hand, investigated *k nearest neighbor join* in Euclidean space.

MapReduce has been proven useful for processing massive graphs. Suri, Vassilvitskii [54], and Tsourakakis et al. [56] considered *triangle counting*, Morales et al. [45] dealt with *b-matching*, Bahmani et al. [5] focused on the discovery of *densest subgraphs*, Karloff et al. [31] analyzed computing connected components and spanning trees, while Lattanzi et al. [39] studied *maximal matching*, *vertex/edge cover*, and *minimum cut*.

Data mining and statistical analysis are also popular topics on MapReduce. Clustering was investigated by Das et al. [15], Cordeiro et al. [13], and Ene et al. [21]. Classification and regression were studied by Panda et al. [51]. Ghoting et al. [23] developed an integrated toolkit to facilitate machine learning tasks. Pansare et al. [52] and Laptev et al. [38] explained how to compute aggregates over a gigantic file. Grover and Carey [25] focused on extracting a set of samples satisfying a given predicate. Chen [11] described techniques for supporting operations of data warehouses.

Among the other algorithmic studies on MapReduce, Chierichetti et al. [12] attacked approximation versions of the *set cover problem*. Wang et al. [60] described algorithms for the simulation of real-world events. Bahmani et al. [4] proposed methods for calculating personalized page ranks. Jestes et al. [30] investigated the construction of wavelet histograms.

2.3 Relevance to Minimal Algorithms

Our study of minimal algorithms is orthogonal to the *framework implementation* category as mentioned in Section 2.2. Even a minimal algorithm can benefit from clever optimization at the system level. On the other hand, a minimal algorithm may considerably simplify optimization. For instance, as the minimal requirements already guarantee excellent load balancing in storage, computation, and communication, there would be little skewness to deserve specialized optimization. As another example, the cost of a minimal algorithm is by definition highly predictable, which is a precious advantage appreciated by cost-based optimizers (e.g., [28, 40]).

This work belongs to the *algorithms on MapReduce* category. However, besides dealing with different problems, we also differ from the existing studies in that we emphasize on an algorithm's minimality. Remember that the difficulty of designing a minimal algorithm lies in excelling in *all* the four aspects (see Section 1) at the same time. Often times, it is easy to do well in only certain aspects (e.g., *constant rounds*), while losing out in the rest. Parallel algorithms on classic platforms are typically compared under multiple metrics. We believe that MapReduce should not be an exception.

From a theoretical perspective, minimal algorithms are reminiscent of algorithms under the *bulk synchronous parallel* (BSP) *model* [57] and *coarse-grained multicomputer* (CGM) *model* [17]. Both models are well-studied branches of theoretical parallel computing. Our algorithmic treatment, however, is system oriented, i.e., easy to implement, while offering excellent performance in practice. In contrast, theoretical solutions in BSP/CGM are often rather involved, and usually carry large hidden constants in their complexities, not to mention that they are yet to be migrated to MapReduce. It is worth mentioning that there has been work on extending the MapReduce framework to enhance its power so as to solve difficult problems efficiently. We refer the interested readers to the recent work of [34].

3. SORTING

In the *sorting problem*, the input is a set S of n objects from an ordered domain. For simplicity, we assume that objects are real values because our discussion easily generalizes to other ordered domains. Denote by $\mathcal{M}_1, ..., \mathcal{M}_t$ the machines in the MapReduce system. Initially, S is distributed across these machines, each storing $O(m)$ objects where $m = n/t$. At the end of sorting, all objects in $\mathcal{M}_i$ must precede those in $\mathcal{M}_j$ for any $1 \le i < j \le t$.

3.1 TeraSort

Parameterized by $\rho \in (0, 1]$, *TeraSort* [50] runs as follows:

Round 1. *Map-shuffle*(ρ)

Every $\mathcal{M}_i$ ($1 \le i \le t$) samples each object from its local storage with probability ρ independently. It sends all the sampled objects to $\mathcal{M}_1$.

Reduce (only on $\mathcal{M}_1$)

1. Let S_{samp} be the set of samples received by $\mathcal{M}_1$, and $s = |S_{samp}|$.
2. Sort S_{samp}, and pick $b_1, ..., b_{t-1}$ where b_i is the $i\lceil s/t \rceil$-th smallest object in S_{samp}, for $1 \le i \le t-1$. Each b_i is a *boundary object*.

Round 2. *Map-shuffle* (assumption: $b_1, ..., b_{t-1}$ have been sent to all machines)

Every $\mathcal{M}_i$ sends the objects in $(b_{j-1}, b_j]$ from its local storage to $\mathcal{M}_j$, for each $1 \le j \le t$, where $b_0 = -\infty$ and $b_t = \infty$ are dummy boundary objects.

Reduce:

Every $\mathcal{M}_i$ sorts the objects received in the previous phase.

For convenience, the above description sometimes asks a machine $\mathcal{M}$ to send data to itself. Needless to say, such data "transfer" occurs internally in $\mathcal{M}$, with no network transmission. Also note the assumption at the map-shuffle phase of Round 2, which we call the *broadcast assumption*, and will deal with later in Section 3.3.

In [50], ρ was left as an open parameter. Next, we analyze the setting of this value to make *TeraSort* a minimal algorithm.

3.2 Choice of ρ

Define $S_i = S \cap (b_{i-1}, b_i]$, for $1 \le i \le t$. In Round 2, all the objects in S_i are gathered by $\mathcal{M}_i$, which sorts them in the reduce phase. For *TeraSort* to be minimal, it must hold:

$\boldsymbol{\mathcal{P}_1}$. $s = O(m)$.
$\boldsymbol{\mathcal{P}_2}$. $|S_i| = O(m)$ for all $1 \le i \le t$.

Specifically, $\mathcal{P}_1$ is because $\mathcal{M}_1$ receives $O(s)$ objects over the network in the map-shuffle phase of Round 1, which has to be $O(m)$ to satisfy *bounded net-traffic* (see Section 1). $\mathcal{P}_2$ is because

$\mathcal{M}_i$ must receive and store $O(|S_i|)$ words in Round 2, which needs to be $O(m)$ to qualify *bounded net-traffic* and *minimum footprint*.

We now establish an important fact about *TeraSort*:

THEOREM 1. *When* $m \geq t\ln(nt)$, $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *hold simultaneously with probability at least* $1 - O(\frac{1}{n})$ *by setting* $\rho = \frac{1}{m}\ln(nt)$.

PROOF. We will consider $t \geq 9$ because otherwise $m = \Omega(n)$, in which case $\mathcal{P}_1$ and $\mathcal{P}_2$ hold trivially. Our proof is based on the Chernoff bound[1] and an interesting bucketing argument.

First, it is easy to see that $\mathbf{E}[s] = m\rho t = t\ln(nt)$. A simple application of Chernoff bound results in:

$$\mathbf{Pr}[s \geq 1.6 \cdot t\ln(nt)] \leq \exp(-0.12 \cdot t\ln(nt)) \leq 1/n$$

where the last inequality used the fact that $t \geq 9$. The above implies that $\mathcal{P}_1$ can fail with probability at most $1/n$. Next, we analyze $\mathcal{P}_2$ under the event $s < 1.6t\ln(nt) = O(m)$.

Imagine that S has been sorted in ascending order. We divide the sorted list into $\lfloor t/8 \rfloor$ sub-lists as evenly as possible, and call each sub-list a *bucket*. Each bucket has between $8n/t = 8m$ and $16m$ objects. We observe that $\mathcal{P}_2$ *holds if every bucket covers at least one boundary object*. To understand why, notice that under this condition, no bucket can fall between two consecutive boundary objects (counting also the dummy ones)[2]. Hence, every S_i, $1 \leq i \leq t$, can contain objects in at most 2 buckets, i.e., $|S_i| \leq 32m = O(m)$.

A bucket β definitely includes a boundary object if β covers more than $1.6\ln(nt) > s/t$ samples (i.e., objects from S_{samp}), as a boundary object is taken every $\lceil s/t \rceil$ consecutive samples. Let $|\beta| \geq 8m$ be the number of objects in β. Define random variable x_j, $1 \leq j \leq |\beta|$, to be 1 if the j-th object in β is sampled, and 0 otherwise. Define:

$$X = \sum_{j=1}^{|\beta|} x_j = |\beta \cap S_{samp}|.$$

Clearly, $\mathbf{E}[X] \geq 8m\rho = 8\ln(nt)$. We have:

$$\begin{aligned}
\mathbf{Pr}[X \leq 1.6\ln(nt)] &= \mathbf{Pr}[X \leq (1-4/5)8\ln(nt)] \\
&\leq \mathbf{Pr}[X \leq (1-4/5)\mathbf{E}[X]] \\
\text{(by Chernoff)} &\leq \exp\left(-\frac{16}{25}\frac{\mathbf{E}[X]}{3}\right) \\
&\leq \exp\left(-\frac{16}{25}\cdot\frac{8\ln(nt)}{3}\right) \\
&\leq \exp(-\ln(nt)) \\
&\leq 1/(nt).
\end{aligned}$$

We say that β *fails* if it covers no boundary object. The above derivation shows that β fails with probability at most $1/(nt)$. As there are at most $t/8$ buckets, the probability that at least one bucket fails is at most $1/(8n)$. Hence, $\mathcal{P}_2$ can be violated with probability at most $1/(8n)$ under the event $s < 1.6t\ln(nt)$, i.e., at most $9/8n$ overall.

Therefore, $\mathcal{P}_1$ and $\mathcal{P}_2$ hold at the same time with probability at least $1 - 17/(8n)$. □

[1]Let $X_1, ..., X_n$ be independent Bernoulli variables with $\mathbf{Pr}[X_i = 1] = p_i$, for $1 \leq i \leq n$. Set $X = \sum_{i=1}^{n} X_i$ and $\mu = \mathbf{E}[X] = \sum_{i=1}^{n} p_i$. The Chernoff bound states (i) for any $0 < \alpha < 1$, $\mathbf{Pr}[X \geq (1+\alpha)\mu] \leq \exp(-\alpha^2\mu/3)$ while $\mathbf{Pr}[X \leq (1-\alpha)\mu] \leq \exp(-\alpha^2\mu/3)$, and (ii) $\mathbf{Pr}[X \geq 6\mu] \leq 2^{-6\mu}$.

[2]If there was one, the bucket would not be able to cover any boundary object.

Discussion. For large n, the success probability $1 - O(1/n)$ in Theorem 1 is so high that the failure probability $O(1/n)$ is negligible, i.e., $\mathcal{P}_1$ and $\mathcal{P}_2$ are almost never violated.

The condition about m in Theorem 1 is tight within a logarithmic factor because $m \geq t$ is an implicit condition for *TeraSort* to work, noticing that both the reduce phase of Round 1 and the map-shuffle phase of Round 2 require a machine to store $t-1$ boundary objects.

In reality, typically $m \gg t$, namely, the memory size of a machine is significantly greater than the number of machines. More specifically, m is at the order of at least 10^6 (this is using only a few *mega* bytes per machine), while t is at the order of 10^4 or lower. Therefore, $m \geq t\ln(nt)$ is a (very) reasonable assumption, which explains why *TeraSort* has excellent efficiency in practice.

Minimality. We now establish the minimality of *TeraSort*, temporarily ignoring how to fulfill the broadcast assumption. Properties $\mathcal{P}_1$ and $\mathcal{P}_2$ indicate that each machine needs to store only $O(m)$ objects at any time, consistent with *minimum footprint*. Regarding the network cost, a machine $\mathcal{M}$ in each round sends only objects that were already on $\mathcal{M}$ when the algorithm started. Hence, $\mathcal{M}$ sends $O(m)$ network data per round. Furthermore, $\mathcal{M}_1$ receives only $O(m)$ objects by $\mathcal{P}_1$. Therefore, *bounded-bandwidth* is fulfilled. *Constant round* is obviously satisfied. Finally, the computation time of each machine $\mathcal{M}_i$ ($1 \leq i \leq t$) is dominated by the cost of sorting S_i in Round 2, i.e., $O(m\log m) = O(\frac{n}{t}\log n)$ by $\mathcal{P}_2$. As this is $1/t$ of the $O(n\log n)$ time of a sequential algorithm, *optimal computation* is also achieved.

3.3 Removing the Broadcast Assumption

Before Round 2 of *TeraSort*, $\mathcal{M}_1$ needs to *broadcast* the boundary objects $b_1, ..., b_{t-1}$ to the other machines. We have to be careful because a naive solution would ask $\mathcal{M}_1$ to send $O(t)$ words to every other machine, and hence, incur $O(t^2)$ network traffic overall. This not only requires one more round, but also violates *bounded net-traffic* if t exceeds $\sqrt{m}$ by a non-constant factor.

In [50], this issue was circumvented by assuming that all the machines can access a distributed file system. In this scenario, $\mathcal{M}_1$ can simply write the boundary objects to a file on that system, after which each M_i, $2 \leq i \leq t$, gets them from the file. In other words, a brute-force *file accessing step* is inserted between the two rounds. This is allowed by the current Hadoop implementation (on which *TeraSort* was based [50]).

Technically, however, the above approach destroys the elegance of *TeraSort* because it requires that, besides sending key-value pairs to each other, the machines should also communicate via a distributed file. This implies that the machines are not share-*nothing* because they are essentially sharing the file. Furthermore, as far as this paper is concerned, the artifact is inconsistent with the definition of minimal algorithms. As sorting lingers in all the problems to be discussed later, we are motivated to remove the artifact to keep our analytical framework clean.

We now provide an elegant remedy, which allows *TeraSort* to still terminate in 2 rounds, and retain its minimality. The idea is to give *all* machines a copy of S_{samp}. Specifically, we modify Round 1 of *TeraSort* as:

Round 1. *Map-shuffle*(ρ)

> After sampling as in *TeraSort*, each $\mathcal{M}_i$ sends its sampled objects to all machines (not just to $\mathcal{M}_1$).

Reduce

> Same as *TeraSort* but performed on all machines (not just on $\mathcal{M}_1$).

Round 2 still proceeds as before. The correctness follows from the fact that, in the reduce phase, every machine picks boundary objects in exactly the same way from an identical S_{samp}. Therefore, all machines will obtain the same boundary objects, thus eliminating the need of broadcasting. Henceforth, we will call the modified algorithm *pure TeraSort*.

At first glance, the new map-shuffle phase of Round 1 may seem to require a machine $\mathcal{M}$ to send out considerable data, because every sample necessitates $O(t)$ words of network traffic (i.e., $O(1)$ to every other machine). However, as every object is sampled with probability $\rho = \frac{1}{m}\ln(nt)$, the number of words sent by $\mathcal{M}$ is only $O(m \cdot t \cdot \rho) = O(t\ln(nt))$ in expectation. The lemma below gives a much stronger fact:

LEMMA 1. *With probability at least* $1 - \frac{1}{n}$*, every machine sends* $O(t\ln(nt))$ *words over the network in Round 1 of pure TeraSort.*

PROOF. Consider an arbitrary machine $\mathcal{M}$. Let random variable X be the number of objects sampled from $\mathcal{M}$. Hence, $\mathbf{E}[X] = m\rho = \ln(nt)$. A straightforward application of Chernoff bound gives:

$$\mathbf{Pr}[X \geq 6\ln(nt)] \leq 2^{-6\ln(nt)} \leq 1/(nt).$$

Hence, $\mathcal{M}$ sends more than $O(t\ln(nt))$ words in Round 1 with probability at most $1/(nt)$. By union bound, the probability that this is true for all t machines is at least $1 - 1/n$. □

Combining the above lemma with Theorem 1 and the minimality analysis in Section 3.2, we can see that *pure TeraSort* is a minimal algorithm with probability at least $1 - O(1/n)$ when $m \geq t\ln(nt)$.

We close this section by pointing out that, the fix of *TeraSort* is of mainly theoretical concerns. Its purpose is to convince the reader that the broadcast assumption is not a technical "loose end" in achieving minimality. In practice, *TeraSort* has nearly the same performance as our *pure* version, at least on Hadoop where (as mentioned before) the brute-force approach of *TeraSort* is well supported.

4. BASIC MINIMAL ALGORITHMS IN DATABASES

A minimal sorting algorithm also gives rise to minimal algorithms for other database problems. We demonstrate so for *ranking*, *group-by*, *semi-join*, and 2D *skyline* in this section. For all these problems, our objective is to terminate in *one more round* after sorting, in which a machine entails only $O(t)$ words of network traffic where t is the number of machines.

As before, each of the machines $\mathcal{M}_1, ..., \mathcal{M}_t$ is permitted $O(m)$ space of storage where $m = n/t$, and n is the problem cardinality. In the rest of the paper, we will concentrate on $m \geq t\ln(nt)$, i.e., the condition under which *TeraSort* is minimal (see Theorem 1).

4.1 Ranking and Skyline

Prefix Sum. Let S be a set of n objects from an ordered domain, such that each object $o \in S$ carries a real-valued *weight* $w(o)$. Define *prefix*(o, S), the *prefix sum* of o, to be the total weight of the objects $o' \in S$ such that $o' < o$. The *prefix sum problem* is to report the prefix sums of all objects in S. The problem can be settled in $O(n\log n)$ time on a sequential machine. Next, we present an efficient MapReduce algorithm.

First, sort S with *TeraSort*. Let S_i be the set of objects on machine $\mathcal{M}_i$ after sorting, for $1 \leq i \leq t$. We solve the prefix sum problem in another round:

Map-shuffle (on each $\mathcal{M}_i$, $1 \leq i \leq t$)

$\mathcal{M}_i$ sends $W_i = \sum_{o \in S_i} w(o)$ to $\mathcal{M}_{i+1}, ..., \mathcal{M}_t$.

Reduce (on each $\mathcal{M}_i$):

1. $V_i = \sum_{j \leq i-1} W_j$.
2. Obtain *prefix*(o, S_i) for $o \in S_i$ by solving the prefix sum problem on S_i locally.
3. *prefix*$(o, S) = V_i +$ *prefix*(o, S_i) for each $o \in S_i$.

In the above map-shuffle phase, every machine sends and receives exactly $t - 1$ values in total: precisely, $\mathcal{M}_i$ ($1 \leq i \leq t$) sends $t - i$ and receives $i - 1$ values. This satisfies *bounded net-traffic* because $t \leq m$. Furthermore, the reduce phase takes $O(m) = O(n/t)$ time, by leveraging the sorted order of S_i. Omitting the other trivial details, we conclude that our prefix sum algorithm is minimal.

Prefix Min. The *prefix min problem* is almost the same as prefix sum, except that *prefix*(o, S) is defined as the *prefix min* of o, which is the minimum weight of the objects $o' \in S$ such that $o' < o$. This problem can also be settled by the above algorithm minimally with three simple changes: redefine (i) $W_i = \min_{o \in S_i} w(o)$ in the map-shuffle phase, (ii-iii) $V_i = \min_{j \leq i-1} W_j$ at Line 1 of the reduce phase, and *prefix*$(o, S) = \min\{V_i,$ *prefix*$(o, S_i)\}$ at Line 3.

Ranking. Let S be a set of objects from an ordered domain. The *ranking problem* reports the *rank* of each object $o \in S$, which equals $|\{o' \in S \mid o' \leq o\}|$; in other words, the smallest object has rank 1, the second smallest rank 2, etc. This can be solved as a special prefix sum problem where all objects have weight 1 (i.e., prefix count).

Skyline. Let x_p (y_p) be the x- (y-) coordinate of a 2D point p. A point p *dominates* another p' if $x_p \leq x_{p'}$ and $y_p \leq y_{p'}$. For a set P of n 2D points, the *skyline* is the set of points $p \in P$ such that p is not dominated by any other point in P. The *skyline problem* [8] is to report the skyline of P, and admits a sequential algorithm of $O(n\log n)$ time [35].

The problem is essentially prefix min in disguise. Specifically, let $S = \{x_p \mid p \in P\}$ where x_p carries a "weight" y_p. Define the *prefix min* of x_p as the minimum "weight" of the values in S preceding[3] x_p. It is rudimentary to show that p is in the skyline of P, if and only if the prefix min of x_p is strictly greater than y_p. Therefore, our prefix min algorithm also settles the skyline problem minimally.

4.2 Group By

Let S be a set of n objects, where each object $o \in S$ carries a *key* $k(o)$ and a *weight* $w(o)$, both of which are real values. A *group* G is a maximal set of objects with the same key. The *aggregate* of G is the result of applying a distributive[4] aggregate function AGG to the weights of the objects in G. The *group-by* problem is to report the aggregates of all groups. It is easy to do so in $O(n\log n)$ time on a sequential machine. Next, we discuss MapReduce, assuming for simplicity AGG = *sum* because it is straightforward to generalize the discussion to other AGG.

---

[3]Precisely, given points p and p', x_p precedes $x_{p'}$ if (i) $x_p < x_{p'}$ or (ii) $x_p = x_{p'}$ but $y_p < y_{p'}$.

[4]An aggregate function AGG is *distributive* on a set S if AGG(S) can be obtained in constant time from AGG(S_1) and AGG(S_2), where S_1 and S_2 form a partition of S, i.e., $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$.

The main issue is to handle large groups that do not fit in one machine. Our algorithm starts by sorting the objects by keys, breaking ties by ids. Consider an arbitrary machine $\mathcal{M}$ after sorting. If a group G is now completely in $\mathcal{M}$, its aggregate can be obtained locally in $\mathcal{M}$. Motivated by this, let $k_{min}(\mathcal{M})$ and $k_{max}(\mathcal{M})$ be the smallest and largest keys on $\mathcal{M}$ currently. Clearly, groups of keys k where $k_{min}(\mathcal{M}) < k < k_{max}(\mathcal{M})$ are entirely stored in $\mathcal{M}$, which can obtain their aggregates during sorting, and remove them from further consideration.

Each machine $\mathcal{M}$ has at most 2 groups remaining, i.e., with keys $k_{min}(\mathcal{M})$ and $k_{max}(\mathcal{M})$, respectively. Hence, there are at most $2t$ such groups on all machines. To handle them, we ask each machine to send at most 4 values to $\mathcal{M}_1$ (i.e., to just a single machine). The following elaborates how:

Map-shuffle (on each $\mathcal{M}_i, 1 \le i \le t$):

1. Obtain the total weight $W_{min}(\mathcal{M}_i)$ of group $k_{min}(\mathcal{M}_i)$, i.e., by considering only objects in $\mathcal{M}_i$.
2. Send pair $(k_{min}(\mathcal{M}_i), W_{min}(\mathcal{M}_i))$ to $\mathcal{M}_1$.
3. If $k_{min}(\mathcal{M}_i) \neq k_{max}(\mathcal{M}_i)$, send pair $(k_{max}(\mathcal{M}_i), W_{max}(\mathcal{M}_i))$ to $\mathcal{M}_1$, where the definition of $k_{max}(\mathcal{M}_i)$ is similar to $k_{min}(\mathcal{M}_i)$.

Reduce (only on $\mathcal{M}_1$):

Let $(k_1, w_1), ..., (k_x, w_x)$ be the pairs received in the previous phase where x is some value between t and $2t$. For each group whose key k is in one of the x pairs, output its final aggregate $\sum_{j|k_j=k} w_j$.

The minimality of our group-by algorithm is easy to verify. It suffices to point out that the reduce phase of the last round takes $O(t \log t) = O(\frac{n}{t} \log n)$ time (since $t \le m = n/t$).

Categorical Keys. We have assumed that the key $k(o)$ of an object is numeric. This is in fact unnecessary because the key ordering does not affect the correctness of group by. Hence, even if $k(o)$ is categorical, we can simply sort the keys alphabetically by their binary representations.

Term Frequency. MapReduce is often introduced with the *term frequency problem*. The input is a document D, which can be regarded as a multi-set of strings. The goal is to report, for every distinct string $s \in D$, the number of occurrences of s in D. In their pioneering work, Dean and Ghemawat [16] gave an algorithm which works by sending all occurrences of a string to an identical machine. The algorithm is not minimal in the scenario where a string has an exceedingly high frequency. Note, on the other hand, that the term frequency problem is merely a group-by problem with every distinct string representing a group. Hence, our group-by algorithm provides a minimal alternative to counting term frequencies.

4.3 Semi-Join

Let R and T be two sets of objects from the same domain. Each object o in R or T carries a key $k(o)$. The *semi-join problem* is to report all the objects $o \in R$ that have a *match* $o' \in T$, i.e., $k(o) = k(o')$. The problem can be solved in $O(n \log n)$ time sequentially, where n is the total number of objects in $R \cup T$.

In MapReduce, we approach the problem in a way analogous to how group-by was tackled. The difference is that, now objects with the same key do not "collapse" into an aggregate; instead, we must output all of them if their (common) key has a match in T. For this reason, we will need to transfer more network data than group by, as will be clear shortly, but still $O(t)$ words per machine.

Define $S = R \cup T$. We sort the objects of the mixed set S by their keys across the t machines. Consider any machine $\mathcal{M}$ after sorting. Let $k_{min}(T, \mathcal{M})$ and $k_{max}(T, \mathcal{M})$ be the smallest and largest keys respectively, among the T-objects stored on $\mathcal{M}$ (a T*-object* is an object from T). The semi-join problem can be settled with an extra round:

Map-shuffle (on each $\mathcal{M}_i, 1 \le i \le t$):

Send $k_{min}(T, \mathcal{M}_i)$ and $k_{max}(T, \mathcal{M}_i)$ to *all* machines.

Reduce (on each $\mathcal{M}_i$):

1. K_{border} = the set of keys received from the last round.
2. $K(\mathcal{M}_i)$ = the set of keys of the T-objects stored in $\mathcal{M}_i$.
3. For every R-object o stored in $\mathcal{M}_i$, output it if $k(o) \in K(\mathcal{M}_i) \cup K_{border}$.

Every machine sends and receives $2t$ keys in the map-shuffle phase. The reduce phase can be implemented in $O(m + t \log t) = O(\frac{n}{t} \log n)$ time, using the fact that the R-objects on $\mathcal{M}_i$ are already sorted. The overall semi-join algorithm is minimal.

5. SLIDING AGGREGATION

This section is devoted to the *sliding aggregation problem*. Recall that the input is: (i) a set S of n objects from an ordered domain, (ii) an integer $\ell \le n$, and (iii) a distributive aggregate function AGG. We will focus on AGG = *sum* because extension to other AGG is straightforward. Each object $o \in S$ is associated with a real-valued *weight* $w(o)$. The *window* of o, denoted as $window(o)$, is the set of ℓ largest objects not exceeding o (see Figure 1). The *window sum* of o equals

$$win\text{-}sum(o) = \sum_{o' \in window(o)} w(o')$$

The objective is to report $win\text{-}sum(o)$ for all $o \in S$.

5.1 Sorting with Perfect Balance

Let us first tackle a variant of sorting which we call the *perfect sorting problem*. The input is a set S of n objects from an ordered domain. We want to distribute them among the t MapReduce machines $\mathcal{M}_1, ..., \mathcal{M}_t$ such that $\mathcal{M}_i$, $1 \le i \le t-1$, stores *exactly* $\lceil m \rceil$ objects, and $\mathcal{M}_t$ stores all the remaining objects, where $m = n/t$. In the meantime, the sorted order must be maintained, i.e., all objects on $\mathcal{M}_i$ precede those on $\mathcal{M}_j$, for any $1 \le i < j \le t$. We will assume that m is an integer; if not, simply pad at most $t-1$ dummy objects to make n a multiple of t.

The problem is in fact nothing but a small extension to ranking. Our algorithm first invokes the ranking algorithm in Section 4.1 to obtain the rank of each $o \in S$, denoted as $r(o)$. Then, we finish in one more round:

Map-shuffle (on each $\mathcal{M}_i, 1 \le i \le t$):

For each object o currently on $\mathcal{M}_i$, send it to $\mathcal{M}_j$ where $j = \lceil r(o)/m \rceil$.

Reduce: No action is needed.

The above algorithm is clearly minimal.

5.2 Sliding Aggregate Computation

We now return to the sliding aggregation problem, assuming that S has been perfectly sorted across $\mathcal{M}_1, ..., \mathcal{M}_t$ as described earlier. The objective is to settle the problem in just one more round. Once again, we assume that n is a multiple of t; if not, pad at most $t-1$ dummy objects with zero weights.

By virtue of the perfect balancing, the objects on machine i form a *rank range* $[(i-1)m+1, im]$, for $1 \leq i \leq t$. Consider an object o with $window(o) = [r(o)-\ell+1, r(o)]$, i.e., the range of ranks of the objects in $window(o)$. Clearly, $window(o)$ intersects the rank ranges of machines from α to β, where $\alpha = \lceil (r(o)-\ell+1)/m \rceil$ to $\beta = \lceil r(o)/m \rceil$. If $\alpha = \beta$, *win-sum*(o) can be calculated locally by $\mathcal{M}_\beta$, so next we focus on $\alpha < \beta$. Note that when $\alpha < \beta - 1$, $window(o)$ *spans* the rank ranges of machines $\alpha+1, ..., \beta-1$.

Let W_i be the total weight of all the objects on $\mathcal{M}_i$, $1 \leq i \leq t$. We will ensure that every machine knows $W_1, ..., W_t$. Then, to calculate *win-sum*(o) at $\mathcal{M}_\beta$, the only information $\mathcal{M}_\beta$ does *not* have locally is the objects on $\mathcal{M}_\alpha$ enclosed in $window(o)$. We say that those objects are *remotely relevant* to $\mathcal{M}_\beta$. Objects from machines $\alpha+1, ..., \beta-1$ are not needed because their contributions to *win-sum*(o) have been summarized by $W_{\alpha+1}, ..., W_{\beta-1}$.

The lemma below points out a crucial fact.

LEMMA 2. *Every object is remotely relevant to at most 2 machines.*

PROOF. Consider a machine $\mathcal{M}_i$ for some $i \in [1, t]$. If a machine $\mathcal{M}_j$ stores at least an object remotely relevant to $\mathcal{M}_i$, we say that $\mathcal{M}_j$ is *pertinent* to $\mathcal{M}_i$.

Recall that the left endpoint of $window(o)$ lies in machine $\alpha = \lceil (r(o)-\ell+1)/m \rceil$. When $r(o) \in [(i-1)m+1, im]$, i.e., the rank range of $\mathcal{M}_i$, it holds that

$$\left\lceil \frac{(i-1)m+1-\ell+1}{m} \right\rceil \leq \alpha \leq \left\lceil \frac{im-\ell+1}{m} \right\rceil \Rightarrow$$
$$(i-1) - \left\lfloor \frac{\ell-1}{m} \right\rfloor \leq \alpha \leq i - \left\lfloor \frac{\ell-1}{m} \right\rfloor \quad (1)$$

where the last step used the fact that $\lceil x - y \rceil = x - \lfloor y \rfloor$ for any integer x and real value y.

There are two useful observations. First,integer α has only two choices satisfying (1), namely, at most 2 machines are pertinent to $\mathcal{M}_i$. Second, as i grows by 1, the two permissible values of α both increase by 1. This means that each machine can be pertinent to at most 2 machines, thus completing the proof. □

COROLLARY 1. *Objects in $\mathcal{M}_i$, $1 \leq i \leq t$, can be remotely relevant only to*

- *machine $i+1$, if $\ell \leq m$*
- *machines $i + \lfloor(\ell-1)/m\rfloor$ and $i+1+\lfloor(\ell-1)/m\rfloor$, otherwise.*

In the above, if a machine id exceeds m, ignore it.

PROOF. Directly from (1). □

We are now ready to explain how to solve the sliding aggregation problem in one round:

Map-shuffle (on each $\mathcal{M}_i$, $1 \leq i \leq t$):

1. Send W_i to all machines.
2. Send all the objects in $\mathcal{M}_i$ to one or two machines as instructed by Corollary 1.

Reduce (on each $\mathcal{M}_i$):

For each object o already in $\mathcal{M}_i$ after perfect sorting:

1. $\alpha = \lceil (r(o)-\ell+1)/m \rceil$
2. w_1 = the total weight of the objects in $\mathcal{M}_\alpha$ that fall in $window(o)$ (if $\alpha < i$, such objects were received in the last phase).
3. $w_2 = \sum_{j=\alpha+1}^{i-1} W_j$.
4. If $\alpha = i$, set $w_3 = 0$; otherwise, w_3 is the total weight of the objects in $\mathcal{M}_i$ that fall in $window(o)$.
5. $win\text{-}sum(o) = w_1 + w_2 + w_3$.

We now analyze the algorithm's minimality. It is clear that every machine sends and receives $O(t+m) = O(m)$ words of data over the network in the map-shuffle phase. Hence, each machine requires only $O(m)$ storage. It remains to prove that the reduce phase terminates in $O(\frac{n}{t}\log n)$ time. We create a *range sum* structure[5] respectively on: (i) the local objects in $\mathcal{M}_i$, (ii) the objects received from (at most) two machines in the map-reduce phase, and (iii) the set $\{W_1, ..., W_t\}$. These structures can be built in $O(m \log m)$ time, and allow us to compute w_1, w_2, w_3 in Lines 2-4 using $O(\log m)$ time. It follows that the reduce phase takes $O(m \log m) = O(\frac{n}{t}\log m)$ time.

6. EXPERIMENTS

This section experimentally evaluates our algorithms on an in-house cluster with one master and 56 slave nodes, each of which has four Intel Xeon 2.4GHz CPUs and 24GB RAM. We implement all algorithms on Hadoop (version 1.0), and allocate 4GB of RAM to the Java Virtual Machine on each node (i.e., each node can use up to 4GB of memory for a Hadoop task). Table 1 lists the Hadoop parameters in our experiments.

Parameter Name	Value
fs.block.size	128MB
io.sort.mb	512MB
io.sort.record.percentage	0.1
io.sort.spill.percentage	0.9
io.sort.factor	300
dfs.replication	3

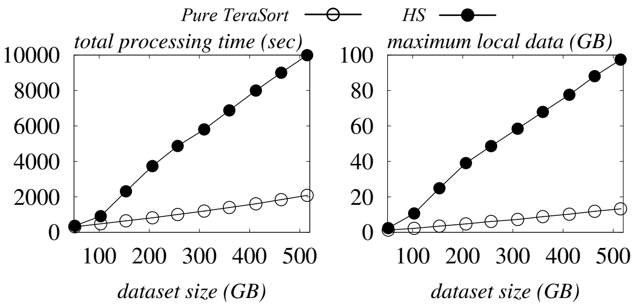
Table 1: Parameters of Hadoop

We deploy two real datasets named LIDAR[6] and PageView[7], respectively. 514GB in size, LIDAR contains 7.35 billion records, each of which is a 3D point representing a location in North Carolina. We use LIDAR for experiments on sorting, skyline, group by, and semi-join. PageView is 332GB in size and contains 11.8 billion tuples. Each tuple corresponds to a page on Wikipedia, and records the number of times the page was viewed in a certain hour during Jan-Sep 2012. We impose a total order on all the tuples by their timestamps, and use the data for experiments on sliding aggregation. In addition, we also generate synthetic datasets to

[5] Let S be a set of n real values, each associated with a numeric weight. Given an interval I, a *range sum* query returns the total weight of the values in $S \cap I$. A simple augmented binary tree [14] uses $O(n)$ space, answers a query in $O(\log n)$ time, and can be built in $O(n \log n)$ time.

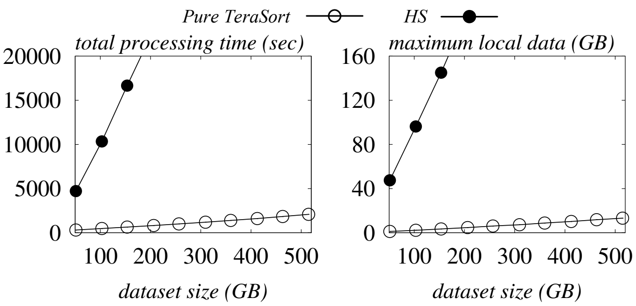
[6] *Http://www.ncfloodmaps.com.*

[7] *Http://dumps.wikimedia.org/other/pagecounts-raw.*



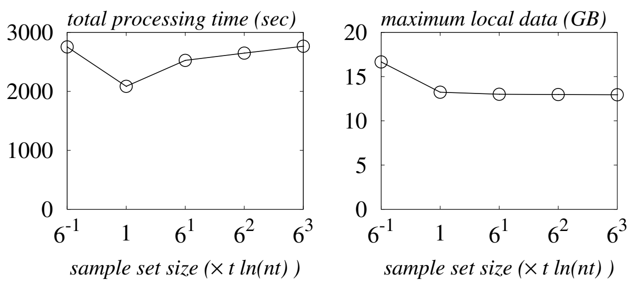
(a) Total time (b) Max. data volume on a slave

Figure 2: *Pure TeraSort* vs. *HS* on LIDAR.



(a) Total time (b) Max. data volume on a slave

Figure 3: *Pure TeraSort* vs. *HS* on modified LIDAR



(a) Total time (b) Max. data volume on a slave

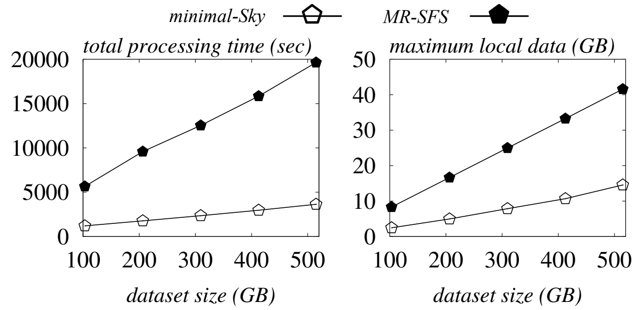
Figure 4: Effects of sample size on *pure TeraSort*

investigate the effect of data distribution on the performance of different algorithms. In each experiment, we run an algorithm 5 times and report the average reading.

6.1 Sorting

The first set of experiments compares *pure TeraSort* (proposed in Section 3.3) with Hadoop's default sorting algorithm, referred to as *HS* henceforth.

Given a dataset of k blocks long in the Hadoop Distributed File System (HDFS), *HS* first asks the master node to gather the first $\lceil 10^5/k \rceil$ records of each block into a set S – call them the *pilot records*. Next, the master identifies $t_{slave} - 1$ boundary points $b_1, b_2, \ldots, b_{t_{slave}-1}$, where b_i is the $i\lceil 10^5/t_{slave} \rceil$-th smallest record in S, and t_{slave} is the number of slave nodes. The mater then launches a one-round algorithm where all records in $(b_{i-1}, b_i]$ are sent to the i-th ($i \in [1, t_{slave}]$) slave for sorting, where $b_0 = 0$ and $b_t = \infty$ are dummies. Clearly, the efficiency of *HS* relies on the distribution of the pilot records. If their distribution is the same as the whole dataset, each slave sorts approximately an equal number of tuples. Otherwise, certain slaves may receive an excessive amount of data and thus become the bottleneck of sorting.



(a) Total time (b) Max. data volume on a slave

Figure 5: *Minimal-Sky* vs. *MR-SFS* on LIDAR

We implement *pure TeraSort* in a way similar to *HS* with the difference in how pilot records are picked. Specifically, the master now forms S by randomly sampling $t \ln t$ records from the dataset. Figure 2a illustrates the running time of *HS* and *pure TeraSort* in sorting LIDAR by its first dimension, when the dataset size varies from 51.4GB to 514GB (a dataset with size smaller than 514GB consists of random tuples from LIDAR, preserving their original ordering.) *Pure TeraSort* consistently outperforms *HS*, with the difference becoming more significant as the size grows. To reveal the reason behind, we plot in Figure 2b the maximum data amount on a slave node in the above experiments. Evidently, while *pure TeraSort* distributes the data evenly to the slaves, *HS* sends a large portion to a single slave, thus incurring enormous overhead.

To further demonstrate the deficiency of *HS*, Figure 3a shows the time taken by *pure TeraSort* and *HS* to sort a modified version of LIDAR, where tuples with small first coordinates are put to the beginning of each block. The efficiency of *HS* deteriorates dramatically, as shown in Figure 3b, confirming the intuition that its cost is highly sensitive to the distribution of pilot records. In contrast, the performance of *pure TeraSort* is not affected, owning to the fact that its sampling procedure is not sensitive to original data ordering at all.

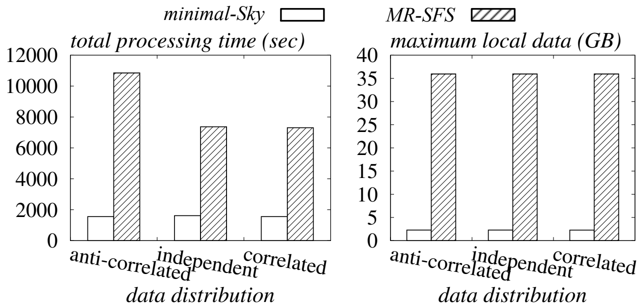
To demonstrate the effect of sample size, Figure 4a shows the cost of *pure Terasort* on LIDAR as the number of pilot tuples changes. The result suggests that $t \ln(nt)$ is a nice choice. When the sample size decreases, *pure Terasort* is slower due to the increased unbalance in the distribution of data across the slaves, as can be observed from Figure 4b. On the opposite side, when the sample size grows, the running time also lengthens because sampling itself is more expensive.

6.2 Skyline

The second set of experiments evaluates our skyline algorithm, referred to as *minimal-Sky*, against *MR-SFS* [61], a recently developed method for skyline computation in MapReduce. We use exactly the implementation of *MR-SFS* from its authors. Figure 5a compares the cost of *minimal-Sky* and *MR-SFS* in finding the skyline on the first two dimensions of LIDAR, as the dataset size increases. *Minimal-Sky* significantly outperforms *MR-SFS* in all cases. The reason is that *MR-SFS*, which is not a minimal algorithm, may force a slave node to process an excessive amount of data, as shown in Figure 5b.

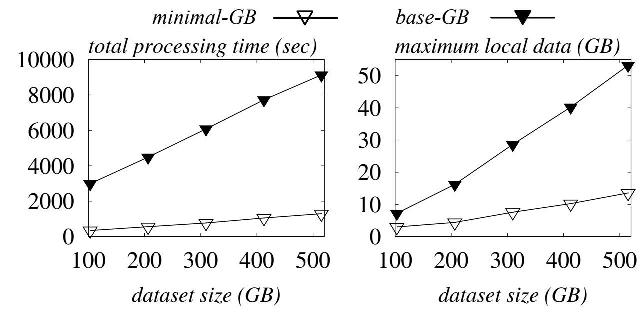
Figure 6 illustrates the performance of *minimal-Sky* and *MR-SFS* on three synthetic datasets that follow a correlated, anti-correlated, and independent distribution, respectively. 120GB in size, each dataset contains 2.5 billion 2D points generated by a publicly available toolkit[8]. Clearly, *MR-SFS* is rather sensitive to the dataset

[8] *Http://pgfoundry.org/projects/randdataset.*



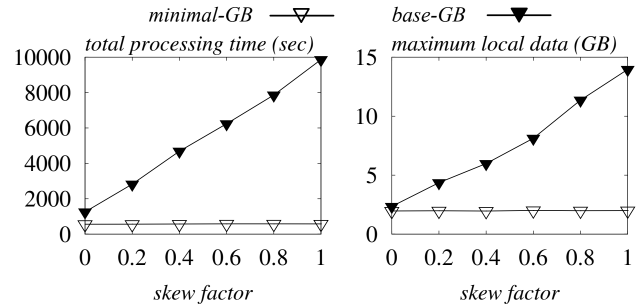
(a) Total time (b) Max. data volume on a slave

Figure 6: *Minimal-Sky* vs. *MR-SFS* on synthetic data



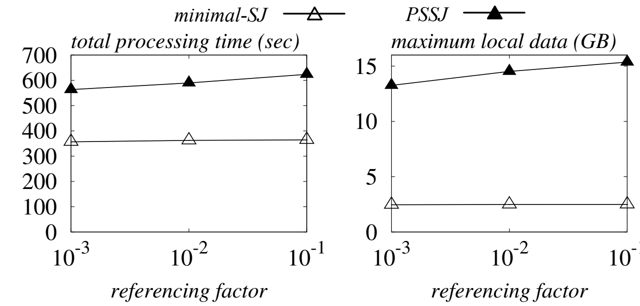
(a) Total time (b) Max. data volume on a slave

Figure 7: *Minimal-GB* vs. *base-GB* on LIDAR



(a) Total time (b) Max. data volume on a slave

Figure 8: *Minimal-GB* vs. *base-GB* on synthetic data



(a) Total time (b) Max. data volume on a slave

Figure 9: *Minimal-SJ* vs. *PSSJ* on various referencing factors

distribution, whereas the efficiency of *Minimal-Sky* is not affected at all.

6.3 Group By

Next, we compare our group by algorithm, referred to as *minimal-GB*, with a baseline approach called *base-GB*. Suppose that we are to group a dataset D by an attribute A. *Base-GB* first invokes a map phase where each tuple $t \in D$ spawns a key-value pair $(t[A], t)$, where $t[A]$ is the value of t on A. Then, all key-value pairs are distributed to the slave nodes using Hadoop's *Partitioner* program. Finally, every slave aggregates the key-value pairs it receives to compute the group by results.

Figure 7a presents the cost of *minimal-GB* and *base-GB* in grouping LIDAR by its first attribute. Regardless of the dataset size, *minimal-GB* is considerably faster than *base-GB* which, as shown in Figure 7b, is because Hadoop's *Partitioner* does not distribute data across the slaves as evenly as *minimal-GB*.

To evaluate the effect of dataset distribution, we generate 2D synthetic datasets where the first dimension (i) has an integer domain $[1, 2.5 \times 10^8]$, and (ii) follows a Zipf distribution with a skew factor between 0 and 1.[9] Each dataset contains 5 billion tuples and is 90GB in size. Figure 8 illustrates the performance of *minimal-GB* and *base-GB* on grouping the synthetic datasets by their first attributes. The efficiency of *base-GB* deteriorates as the skew factor increases. This is because *base-GB* always sends tuples with an identical group-by key to the same slave node. When the group-by keys are skewed, the data distribution is very uneven on the slaves, leading to severe performance penalty. In contrast, *minimal-GB* is completely insensitive to data skewness.

6.4 Semi-Join

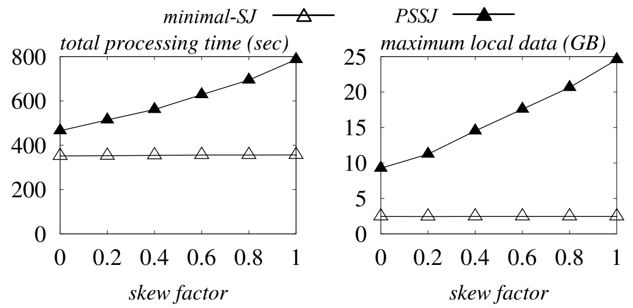
We now proceed to evaluate our semi-join algorithm, referred to as *minimal-SJ*, with *Per-Split Semi-Join* (*PSSJ*) [7], which is the best existing MapReduce semi-join algorithm. We adopt the implementation of *PSSJ* that has been made available online at *sites.google.com/site/hadoopcs561*. Following [7], we generate synthetic tables T and R as follows. The attributes of T are A_1 and A_2, both of which have an integer domain of $[1, 2.5 \times 10^8]$. T has 5 billion tuples whose A_1 values follow a Zipf distribution (some tuples may share an identical value). Their A_2 values are unimportant and arbitrarily decided. Similarly, R has 10 million tuples with integer attributes A_1 and A_3 of domain $[1, 2.5 \times 10^8]$. A fraction r of the tuples in R carry A_1 values present in T, while the other tuples have A_1 values absent from T. We refer to r as the *referencing factor*. Tuples' A_3 values are unimportant and arbitrarily determined.

Figure 9 compares *minimal-SJ* and *PSSJ* under different referencing factors, when the skew factor of $T.A_1$ equals 0.4. In all scenarios, *Minimal-SJ* beats *PSSJ* by a wide margin. Figure 10a presents the running time of *minimal-SJ* and *PSSJ* as a function of the skew factor of $T.A_1$, setting the reference factor r to 0.1. The efficiency of *PSSJ* degrades rapidly as the skew factor grows which, as shown Figure 10b, is because *PSSJ* fails to distribute the workload evenly among the slaves. *Minimal-SJ* is not affected by skewness.

6.5 Sliding Aggregation

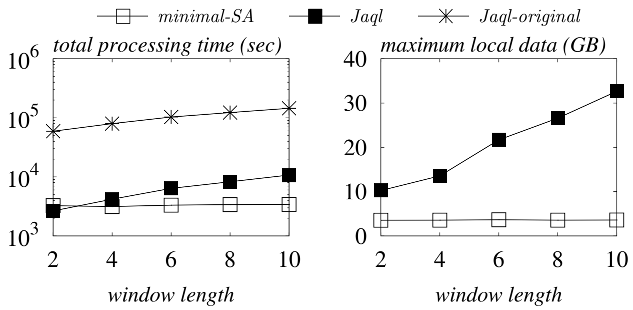
In the last set of experiments, we evaluate our sliding aggregation algorithm, referred to as *minimal-SA*, against a baseline solution referred to as *Jaql*, which corresponds to the algorithm proposed in [6]. Suppose that we want to perform sliding aggregation over a set S of objects using a window size $l \le n$. *Jaql* first sorts S with our ranking algorithm (Section 4.1). It then maps each record $t \in S$ to l key-value pair $(t_1, t), ..., (t_l, t)$, where $t_1, ..., t_l$ are the l largest objects not exceeding t. Then, *Jaql* distributes the key-value pairs to the slaves by applying Hadoop's *Partitioner*, and

[9] Data are more skewed when the skew factor is higher. In particular, when the factor is 0, the distribution degenerates into uniformity.



(a) Total time (b) Max. data volume on a slave

Figure 10: *Minimal-SJ* vs. *PSSJ* on various skew factors



(a) Total time (b) Max. data volume on a slave

Figure 11: Sliding aggregation on small window sizes

instructs each slave to aggregate the key-value pairs with the same key. Besides our own implementation of *Jaql*, we also examine the original implementation released by the authors of [6], henceforth called *Jaql-original*.

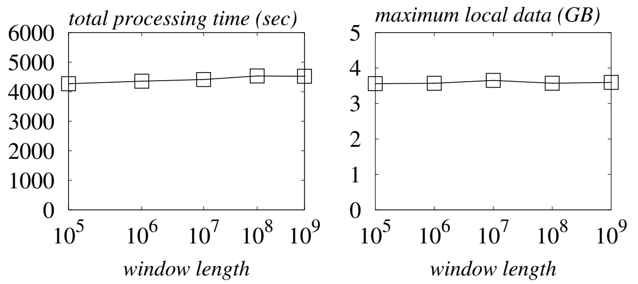
Figure 11 demonstrates the performance of *minimal-SA*, *Jaql*, and *Jaql-original* on the PageView dataset, varying l from 2 to 10. *Minimal-SA* is superior to *Jaql* in all settings, except for a single case $l = 2$. In addition, *minimal-SA* is not affected by l, while *Jaql* deteriorates linearly. *Jaql-original* is slower than the other two methods by a factor of over an order of magnitude. It is not included in Figure 11b because it needs to keep almost the entire database on a single machine, which becomes the system's bottleneck.

Focusing on large l, Figure 12 plots the running time of *minimal-SA* when l increases from 10^5 to 10^9. We omit the Jaql implementations because they are prohibitively expensive, and worse than *minimal-SA* by more than a thousand times.

7. CONCLUSIONS

MapReduce has grown into an extremely popular architecture for large-scaled parallel computation. Even though there have been a great variety of algorithms developed for MapReduce, few are able to achieve the ideal goal of parallelization: balanced workload across the participating machines, and a speedup over a sequential algorithm linear to the number of machines. In particular, currently there is a void at the conceptual level as to what it means to be a "good" MapReduce algorithm.

We believe that a major contribution of this paper is to fill the aforementioned void with the new notion of "minimal MapReduce algorithm". This notion puts together for the first time four strong criteria towards (at least asymptotically) the highest parallel degree. At first glance, the conditions of minimality appear to be fairly stringent. Nonetheless, we prove the existence of simple yet elegant algorithms that minimally settle an array of important database problems. Our extensive experimentation demonstrates



(a) Total time (b) Max. data volume on a slave

Figure 12: *Minimal-SA* on large window sizes

the immediate benefit brought forward by minimality that, the proposed algorithms significantly improve the existing state of the art for all the problems tackled.

ACKNOWLEDGEMENTS

Yufei Tao was supported in part by (i) projects GRF 4166/10, 4165/11, and 4164/12 from HKRGC, and (ii) the WCU (World Class University) program under the National Research Foundation of Korea, and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007). Wenqing Lin and Xiaokui Xiao were supported by the Nanyang Technological University under SUG Grant M58020016, and by the Agency for Science, Technology, and Research (Singapore) under SERC Grant 102-158-0074. The authors would like to thank the anonymous reviewers for their insightful comments.

8. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[2] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, and J. D. Ullman. Fuzzy joins using mapreduce. In *ICDE*, pages 498–509, 2012.

[3] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *TKDE*, 23(9):1282–1298, 2011.

[4] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized pagerank on mapreduce. In *SIGMOD*, pages 973–984, 2011.

[5] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *PVLDB*, 5(5):454–465, 2012.

[6] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[7] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986, 2010.

[8] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

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

[10] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a sql implementation on the mapreduce framework. *PVLDB*, 4(12):1318–1327, 2011.

[11] S. Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *PVLDB*, 3(2):1459–1468, 2010.

[12] F. Chierichetti, R. Kumar, and A. Tomkins. Max-cover in map-reduce. In *WWW*, pages 231–240, 2010.

[13] R. L. F. Cordeiro, C. T. Jr., A. J. M. Traina, J. Lopez, U. Kang, and C. Faloutsos. Clustering very large multi-dimensional datasets with mapreduce. In *SIGKDD*, pages 690–698, 2011.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[15] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, pages 271–280, 2007.

[16] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[17] F. K. H. A. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *SoCG*, pages 298–307, 1993.

[18] J. Dittrich, J.-A. Quiane-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1):518–529, 2010.

[19] I. Elghandour and A. Aboulnaga. Restore: Reusing results of mapreduce jobs. *PVLDB*, 5(6):586–597, 2012.

[20] M. Y. Eltabakh, Y. Tian, F. Ozcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.

[21] A. Ene, S. Im, and B. Moseley. Fast clustering using mapreduce. In *SIGKDD*, pages 681–689, 2011.

[22] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *PVLDB*, 4(7):419–429, 2011.

[23] A. Ghoting, P. Kambadur, E. P. D. Pednault, and R. Kannan. Nimble: a toolkit for the implementation of parallel data mining and machine learning algorithms on mapreduce. In *SIGKDD*, pages 334–342, 2011.

[24] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[25] R. Grover and M. J. Carey. Extending map-reduce for efficient predicate-based sampling. In *ICDE*, pages 486–497, 2012.

[26] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *ICDE*, pages 522–533, 2012.

[27] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *ICDE*, pages 1199–1208, 2011.

[28] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[29] E. Jahani, M. J. Cafarella, and C. Re. Automatic optimization for mapreduce programs. *PVLDB*, 4(6):385–396, 2011.

[30] J. Jestes, F. Li, and K. Yi. Building wavelet histograms on large data in mapreduce. In *PVLDB*, pages 617–620, 2012.

[31] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.

[32] N. Khoussainova, M. Balazinska, and D. Suciu. Perfxplain: Debugging mapreduce job performance. *PVLDB*, 5(7):598–609, 2012.

[33] L. Kolb, A. Thor, and E. Rahm. Load balancing for mapreduce-based entity resolution. In *ICDE*, pages 618–629, 2012.

[34] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, pages 223–234, 2011.

[35] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *JACM*, 22(4):469–476, 1975.

[36] Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD*, pages 25–36, 2012.

[37] W. Lang and J. M. Patel. Energy management for mapreduce clusters. *PVLDB*, 3(1):129–139, 2010.

[38] N. Laptev, K. Zeng, and C. Zaniolo. Early accurate results for advanced analytics on mapreduce. *PVLDB*, 5(10):1028–1039, 2012.

[39] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA*, pages 85–94, 2011.

[40] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *PVLDB*, 5(11):1196–1207, 2012.

[41] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *SIGMOD*, pages 961–972, 2011.

[42] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *PVLDB*, 5(10):1016–1027, 2012.

[43] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[44] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.

[45] G. D. F. Morales, A. Gionis, and M. Sozio. Social content matching in mapreduce. *PVLDB*, 4(7):460–469, 2011.

[46] K. Morton, M. Balazinska, and D. Grossman. Paratimer: a progress indicator for mapreduce dags. In *SIGMOD*, pages 507–518, 2010.

[47] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *PVLDB*, 3(1):494–505, 2010.

[48] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, pages 949–960, 2011.

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

[50] O. O'Malley. Terabyte sort on apache hadoop. Technical report, Yahoo, 2008.

[51] B. Panda, J. Herbach, S. Basu, and R. J. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. *PVLDB*, 2(2):1426–1437, 2009.

[52] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 4(11):1135–1145, 2011.

[53] A. Shinnar, D. Cunningham, B. Herta, and V. A. Saraswat. M3r: Increased performance for in-memory hadoop jobs. *PVLDB*, 5(12):1736–1747, 2012.

[54] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

[55] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[56] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *SIGKDD*, pages 837–846, 2009.

[57] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[58] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac. Adaptive mapreduce using situation-aware mappers. In *EDBT*, pages 420–431, 2012.

[59] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

[60] G. Wang, M. A. V. Salles, B. Sowell, X. Wang, T. Cao, A. J. Demers, J. Gehrke, and W. M. White. Behavioral simulations in mapreduce. *PVLDB*, 3(1):952–963, 2010.

[61] B. Zhang, S. Zhou, and J. Guan. Adapting skyline computation to the mapreduce framework: Algorithms and experiments. In *DASFAA Workshops*, pages 403–414, 2011.

[62] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. *PVLDB*, 5(11):1184–1195, 2012.