On the Optimality of TeraSort in MapReduce

by

Fei Xia
B.Mgt, Tsinghua University, 2013

A Master's Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

On the Optimality of TeraSort in MapReduce

by

Fei Xia
B.Mgt, Tsinghua University, 2013

Supervisory Committee

Dr. Alex Thomo, Co-Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-Supervisor
(Department of Computer Science)

Supervisory Committee

Dr. Alex Thomo, Co-Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-Supervisor
(Department of Computer Science)

ABSTRACT

MapReduce is a scalable, reliable and easy-to-program parallel computation framework for massive data processing. The key for a MapReduce algorithm to be efficient is the balance of workloads on the participating machines. Building on the notion of *minimal* MapReduce algorithms, this project report discusses the sampling and partitioning techniques used in TeraSort. For one of them, we improve the bound on partition sizes to one of asymptotic optimality in terms of increasing number of partitions. In light of the wide applicability of this partition technique, our result potentially strengthens the worst case performance guarantee in other algorithms. We show the application in top-k and k-selection problems as an example.

Contents

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

My supervisors **Dr. Alex Thomo** and **Dr. Venkatesh Srinivasan**
for their kind support academically and personally

Instructors
for conveyance of knowledge and ideas

University of Victoria
for granting me the fellowship

DEDICATION

To Emily.

Chapter 1

Introduction

MapReduce [7, 6] was invented and has become the de facto standard to process massive data sets. An typical MapReduce implementation consists of a set of network-linked computation nodes, upon which jobs and tasks defined by MapReduce algorithms are executed. Because MapReduce handles data sets that is unlikely to reside in any single computation node, a distributed storage management scheme almost always come hand in hand with the computation part of the implementation. For example, the widely adopted open source implementation Hadoop utilizes the distributed file system HDFS [26] and variants to manage storage. It is essential that data in the MapReduce system is robustly handled: when errors occur such as the failure of a computation node, the objective is always to safely cope with them with no loss of data.

MapReduce provides an easy and reliable way to program solutions to a wide range of problems related to massive data processing. In overview, a MapReduce algorithm executes in rounds, while each round consists of three phases - map, shuffle, reduce. The map phase transforms the input to key-value pairs of desired format. Pairs are transferred into different groups in the shuffle phase. In the end, each group becomes a reducer that outputs the result of this round by performing local computation - in the reduce phase, no network traffic occurs between reducers.

Among the various operations applicable on data sets, sorting is of particular significance. It could be the key to a wide field of problems because of the strong consistency between order and physical location that it places on the data, the relative low cost of space and time and the rich collection of simple algorithms. Sorting is the basis for a handful of solutions to database operations - selection, table join, grouping and many others [11], and thus the candidate to tackle challenges of massive

data management. In a sequential machine, the generic swap based algorithms have achieved the theoretical lower bound $O(n \log n)$ time complexity, while others which operate on limited types of data could go below $n \log n$ (e.g. radix sort on integers consumes $O(n)$ time).

The state-of-the-art sorting algorithm in MapReduce is TeraSort [12, 21, 20] which won Jim Gray's sort benchmark in 2009 and 2013. The algorithm conceptually consists of three steps:

1. *Sample*. First of all, a sample of the input data set is constructed by certain strategy.

2. *Partition*. Compute t partition elements for the data set from the sample. In most cases, it is conducted in this way: imagine the sample elements lie in order and pick the t splitters that evenly divide.

3. *Sort*. Partition elements form an segmentation of the data set by order. Sort each partition. Normally each partition is processed in a reducer. Since one partition is either entirely larger or smaller than another, the algorithm has sorted the input data set.

Clearly, the construction of the sample in TeraSort is crucial to efficiency, as it determines the evenness of the reducers in size. For it to be a good sketch of the data set, it must not be too small and skewed; on the other hand, while it usually implies better sketching, large samples could incur expensive overheads. We present a technical overview of the sampling strategies in TeraSort in Section 1.2 and focus on one of them, self-sampling, in the rest of the report.

The problem of selecting the k-th smallest or largest element out of a set/multi-set has been studied since the early times of computer science; Hoare's Find [13] is one well-known linear time solution in sequential computation. In the domain of massive data sets, selecting the top k elements (top-k problem) is a common scenario. The top-k problem is trivial when k is very small; we could simply let each reducer output a set of the top k local elements and, in the next round, compute the top k elements of the data set by aggregating all the local sets in a reducer. However, when k is large, the last reducer has inevitably heavy workload and thus undesirably long running time.

The approach of partition in TeraSort offers a simple solution to the top-k and k-selection problem. Once we know the sizes of the partitions, we can locate the

k-th element in one together with its local offset. Then we are able to "select" it (k-selection) or all elements higher than it (top-k).

1.1 Previous Work on MapReduce Algorithms

1.1.1 MapReduce Algorithms for Fundamental Problems

Databases. [1] studies multi-way joins in MapReduce and the trade-off between communication cost and speed in various schedules of joins. [18] [36] studies efficient MapReduce multi-way *theta-joins*[1]. Efficient *Set-similarity joins* is proposed in [32] . [17] put forth V-smart-join in for the problem of all-pair set similarity joins in MapReduce. [3] compares MapReduce join algorithms in log processing and also offers a solution to semi-joins in MapReduce.

High-level languages are developed with emphasis on program reusability and data management. Queries written in the high-level languages are translated to MapReduce operations and executed. To name a few, Hive[30], Pig[19], Jaql[2], Dremel[16], SCOPE[4]. While expressiveness drives the development of high-level query languages, the flexibility of the MapReduce operations is also appreciated in traditional databases, as seen in [27], an Oracle in-database hadoop implementation in favour of the MapReduce programming style.

Graph. MapReduce has proved effective in processing massive graphs. [28] [31] have studied *triangle counting*; the problem of *triangle enumeration* is studied in [5] [22] have studied the slightly . Finding the *minimal spanning tree* is investigated in [15] via edge set split and in [14] via vertex set split. [15] also provides solutions to *max matching* and *minimum cut*.

1.1.2 Models for MapReduce

Models have been introduced to capture the notion of time and space efficiency of MapReduce algorithms. We recall them here.

Minimal MapReduce algorithms. [29] proposes the notion of minimal MapReduce algorithms. Denote by n the input size of the problem and by t the number

[1] An extended join operation in which the relation is in addition allowed to be inequality.

of machines used in the MapReduce system; let $m = \frac{n}{t}$. A minimal MapReduce algorithm is characterized by the limited computation and communication resources it consumes:

- *minimal footprint*. Every machine uses $O(m)$ space.
- *bounded traffic*. In each round, every machine sends and receives $O(m)$ size of information.
- *constant round*. The algorithm finishes in constant rounds.
- *optimal computation*. The time of computation at every machine is $O(T_{seq}/t)$, where T_{seq} is the time of solving the problem on a single sequential machine.

Minimal MapReduce algorithms exhibit good scalability: not only do they have balanced workloads and traffic, but also they are well accelerated through the use of multiple machines, as indicated by the fourth point. The requirement of constant rounds in fact implies that throughout the entire algorithm, only $O(n)$ communication occurs.

[29] shows that TeraSort is minimal with appropriate parameters.

The MapReduce Class ($\mathcal{MRC}$). [14] puts forth the notion of $\mathcal{MRC}$, a class of MapReduce algorithms computable by a MapReduce system characterized with certain amount of resources. Let n be the length of the input, class $\mathcal{MRC}^i$ is able to run on the following systems,

- *mapper and reducer*. Mappers and reducers are implemented by RAMs with $O(\log n)$-length words, have $O(n^{1-\epsilon})$ space and runs in time polynomial to n. Mappers and reducers can be randomized.
- *map output*. The total size of the output of the map phase in any round is $O(n^{2-2\epsilon})$.
- *round*. The number of rounds is $O(\log^i n)$.
- *machines*. The total number of machines available is $\Theta(n^{1-\epsilon})$.

When the algorithm is randomized, it must output the correct answer with probability at least 3/4. $\mathcal{MRC}$ is defined by the union of $\mathcal{MRC}^i$ over i. The deterministic subset of $\mathcal{MRC}$ is called $\mathcal{DMRC}$.

The sublinear constraints in the resources available in mappers and reducers and the number of machines are crucial, because the input size n could be very large and it is unrealistic to assume our capability of manufacturing such powerful machines economically. On the other hand, if machines are capable of holding $O(n)$ size of items, then there is no need to solve the problem in MapReduce anymore; the problem could just be solved on a single machine by sequential algorithm.

Not all algorithms in $\mathcal{MRC}$ are efficient; rather it offers to characterize them. One would expect efficient ones in $\mathcal{MRC}^i$ with small i and large ϵ in the constraints. Indeed $\mathcal{MRC}^0$ consists of constant-round algorithms.

[14] shows that a variety of problems have solutions in $\mathcal{MRC}^0$ and $\mathcal{MRC}^1$, such as finding an MST of a dense graph, frequency moments, undirected s-t connectivity. It also proves that certain CREW PRAM algorithms can be simulated by $\mathcal{MRC}$ algorithms.

Massive, unordered, distributed (MUD) algorithms. MUD is a class of MapReduce algorithms proposed by [9] to compute a *distributed* stream. The MUD algorithm consists of three components,

- *local function* $\Phi : \Sigma \to Q$. It takes a single input data item and output a message.
- *aggregator* $\oplus : Q \times Q \to Q$. The aggregation function combines two messages into one. It is applied repeatedly on a group of messages until there is only one message, which is the result of the aggregation. The result may depend on the order of application T for which we denote by $m_T(\cdot)$ the computation process.
- *post-processing* $\eta : Q \to \Sigma$. The overall output of the algorithm is obtained by mapping the output of aggregation via η. The algorithm designer need to ensure that the overall output is independent of the order of application of the aggregator.

The communication complexity of a MUD algorithm is $\log |Q|$. The space/time complexity is the maximum of the space/time complexity of the components $\Phi, \oplus, \eta$.

The connection between MUD and MapReduce framework is obvious. The local function could be implemented by mappers. The independence of $\eta \circ m_T$ and the order of application T implies that we could divide and conquer the output of local function O. Indeed, we arbitrarily split O into k reducers and aggregate within each reducers.

The result is k messages. We may repeat this process several times - corresponding to several rounds in MapReduce - until the result reduces to only one message. In the end, the message is transformed by η into the overall output.

It is clear from the above construction that MUD algorithms can be computed very efficiently in a MapReduce system and mostly independent of the underlying computing capability. Since we are able to split and aggregate the output of local functions in *any* manner, we could always choose an execution plan with balanced reducers of appropriate size.

[9] shows that any deterministic streaming algorithm that computes a symmetric (order-invariant) function $\Sigma^n \to \Sigma$ can be simulated by a MUD algorithm with the same communication complexity and the square of space complexity.

1.2 Strategies of Sampling and Partitioning

Sampling and partitioning is the central idea in TeraSort. As the performance of TeraSort is sensitive to the quality of the partition, it is worth examining such strategies for a highly even one. Techniques discussed in this section might not be suitable for the MapReduce framework; implementing them incurs high overhead due to the lack of communication-related features assumed efficient by the strategies. For example, the majority of them were originally developed in the context of inter-connected machines where, rather than following a shuffling (grouping) pattern, nodes send messages to one another freely. They also assume that an element is physically located in a machine while in MapReduce, the location and execution of reducers is beyond the control of algorithm designers.

In practice, however, MapReduce systems are very likely to have implemented a broader range of distributed computation interfaces. They are built on primitive operations of general inter-connected machines, so in addition to carrying out MapReduce algorithms, they provide practical extensions to MapReduce computation. For instance, Hadoop fuses mappers and reducers; thus mappers *remember* the history of processed items. Although the communication still follows the shuffling pattern, the more powerful mappers may shorten the number of rounds and remove certain overheads found in pure MapReduce algorithms. The sampling techniques can potentially be implemented in the practical MapReduce systems with much less overhead.

Putting elements into buckets is very useful in sorting in distributed systems. Elements in a bucket are all smaller or larger than another. Each computation node

in the system holds one or more buckets and all nodes can perform sorting within the local buckets in parallel. The running time depends heavily on the maximum amount of elements among all nodes: it takes the longest time in local sort and also likely in receiving the elements via network. Denote by n the number of elements in the data set, by t the number of partitions, and by $m := \frac{n}{t}$ the optimal load of nodes, i.e. the average amount of elements across all nodes. We measure the *unevenness* of the approximate partition as a ratio of the maximum load over m; *evenness* is the opposite.

The buckets are usually identified by partitioning the data set, for which the idea of sampling goes a long way. It first takes a sample from the data set and then pick the even-splitters of the sample as the partition elements of the data set. The idea is not new. Sample-sort [10], a generalized quick-sort, chooses pivots that divide a random sample of the input. In distributed systems, the idea inspired a variety of sorting algorithms, [8],[24],[35],[23], to name a few.

We remark that these sampling strategies and the coupled partition techniques are drop-in candidates for the first two steps in TeraSort. The third step, sort, relies solely on the partition. Each option presumes certain efficient operations in the MapReduce system. For example, self-sampling may demand high-quality yet inexpensive random number generator; in some systems, sampling without replacement across the entire data set might not be as costly as it generally is, and becomes a viable approach. The vanilla strategy and the variants are practical but there is no worst-case guarantee.

In this report, we focus on analyzing TeraSort with self-sampling.

Vanilla TeraSort. In the current implementation of TeraSort (shipped in Hadoop as a code example), the sample is created by reading a elements in total from b locations which are evenly spread across the input data set[2]. a and b are configurable by users. At each location, $\frac{a}{b}$ elements are read. No solid guarantee exists that such sampling scheme yields good partitioning. In fact, there are bad cases for every a, b that the partitions are extremely unbalanced. When the sample comprises elements concentrated in a few small ranges, it may well lead to uneven buckets.

Variants of the approach of vanilla TeraSort are widely adopted in the participants of Jim Gray's sort benchmark[3] and are effective for what is perceived in the

[2]Technically, the input data set is stored in splits, which can be thought as basic blocks. Therefore, instead of picking locations "into" a split, the algorithm simply picks splits evenly.

[3]The benchmark does not limit the computation model.

experiments. For example, the latest winner FuxiSort [34] adds randomness into the selection of read locations.

Self-Sampling. [29] discusses the strategy of *self-sampling*, by which we mean each element is picked into the sample independently at the same probability. Self-sampling fits well into the MapReduce framework as mappers are assumed having no other knowledge than the input item currently being processed. It is the focus of this report. As will be shown later, self-sampling has very appealing probabilistic properties and it actually achieves asymptotic optimal evenness with high probability.

Sampling with/without replacement. Towards the its end, [29] also reports the experiment results on another strategy, sampling without replacement. The results itself is promising and comparable to self-sampling: the unevenness remains stable at a low level when the sample size is no less than the expected size in self-sampling, regardless of how much it excesses the latter. However, there is no further investigation on how the evenness is affected by the layout of the input data set, nor is provided a uniform bound on all layouts.

[33] discusses sufficient sample sizes for this sampling strategy to generate a ϵ-accurate range partition with probability at least $1 - 1/\delta$.[4] Their model is general; here we phrase a slightly simplified version.[5] The input data set is stored across k sites (this describes the reality that in distributed systems data is stored across machines), and sampling is carried out in three steps - onsite sample, merge, partition. To generate a partition of t buckets,

- *Sample.* At site i, create a random sample S_i of size s/t. Also keep the number of elements n_i on this site.
- *Merge.* A total sample is constructed by adding n_i copies of every element in S_i, for all i's.
- *Partition.* Partition the total sample in the usual way to produce $t-1$ partition elements.

[4] An ϵ-accurate range partition is one in which the maximum size of partitions is less than $(1+\epsilon)m$, m is defined above as the average load.

[5] The model considers, besides what are mentioned here, unequal partition sizes and data replication.

The conclusion is that when the total sample size

$$s \geq \frac{(b^2+3)t-4}{2(\epsilon^2-\frac{1}{t})} \cdot (\log(\frac{1}{\delta}) + \log(t) + k\log(n))$$

where $b := \max_i \frac{n_i}{n}$ and we require $\epsilon t > 1$, the generated partition is ϵ-accurate with probability at least $1-\delta$. Let $\delta = 1/n$ and $\epsilon > 1/t$ be any constant , we find that s can take $\Theta(kt\log(n))$, a reasonable figure in many cases.

Note that, when $k = 1$ we are directly sampling across the entire data set and the above sample size reduces to $\Theta(t\log(n))$.

Regular sampling. [25] proposes a deterministic technique called regular sampling that aids the construction of approximate partition when data is store across t machines and we are only allowed to sample locally. To produce a partition of t buckets, the algorithm first sort the data locally at each machine and picks $t-1$ even splitters. Then after all $t(t-1)$ splitters are sent into one machine, within them select elements y_j's of ranks: $\left\{ \frac{t}{2} + t(j-1) \right\}_{j=1}^{t-1}$. It argues that irrelevant of the layout of the input data set, the partition sizes are at most $2m$.

We describe a slightly generalized version of the algorithm and show how regular sampling manages to reduce the influence of data layout. Instead of picking $t-1$ splitters at each machine, we pick r. We also suppose that each of the k machines store $m' := \frac{n}{k}$ elements and the goal is to partition the data set into t buckets; for now we assume $(t-1)|rk$. Within the rk splitters, select y_j's of ranks: $\left\{ \frac{c}{2} + c(j-1) \right\}_{j=1}^{t-1}$, where $c = \frac{rk}{t-1}$. The partition then produces: $\left\{ [y_{j-1}, y_j)_{j=1}^{t} \right\}$ with $y_0 := -\infty$ and $y_t := +\infty$.

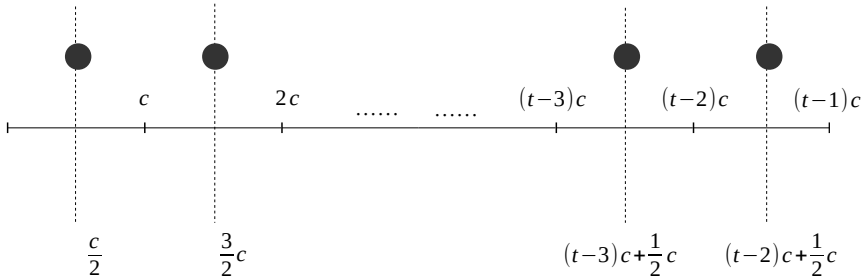

Figure 1.1: Picks of partition elements in regular sampling. $c = \frac{rk}{t-1}$. The original version is $c = t$.

Now we consider how many elements could reside in each partition.

1. $j = 1$. For the first partition $(-\infty, \frac{c}{2})$, there are $rk - \frac{c}{2}$ splitters larger than it. Any splitter x comes from a machine. Denote the next splitter on that machine as x' (or $x' = +\infty$ if x is the last). There are $\frac{m'}{r+1}$ elements in $[x, x')$ on the machine. The correspondence of elements in $[x, x')$ to x is a function, meaning that no element maps to two splitters. Therefore, in the data set, there are at least $(rk - \frac{c}{2}) \cdot \frac{m'}{r+1}$ elements outside the first partition. In turn, there are at most $n - (rk - \frac{c}{2}) \cdot \frac{m'}{r+1} = m(\frac{t}{r+1} + \frac{1}{2}\frac{r}{r+1}\frac{t}{t-1})$ elements in the partition.

2. $j = t$. For the last partition $[y_{t-1}, +\infty)$, similar analysis to $j = 1$ yields the same upper bound.

3. $2 \leq j \leq t - 1$. For partition $[y_{j-1}, y_{j+2})$, there are at least $left$ splitters smaller than it, indicating $(left + 1) \cdot \frac{m'}{r+1}$ elements (we consider $-\infty$ as the image of the $\frac{m'}{r+1}$ elements smaller than the first splitter). There are also at least $right := rk - (left + 1 + t) + 1$ splitters larger and thus $right \cdot \frac{m'}{r+1}$ elements. Combining them leads to an upper bound on the size of the partition: $m(\frac{t}{r+1} + \frac{t}{t-1}(\frac{r}{r+1} - \frac{t-1}{k(r+1)}))$.

When we remove the assumption $t-1|rk$ and switch to a variant of a scheme called ordered even partition (Definition 2 introduced in Section 3.1), the factors before m in the above bounds increase at most $\frac{1}{r+1}$ in case 1,2 and remains the same in case 3.[6]

It is easy to see that when k, r, t are of the same order, regular sampling ensures that every partition has $O(m)$ number of elements. In particular, when $k = t, r = t-1$, we fall back to the original version of regular sampling with bound $2m$.

1.3 Motivation and contribution

In TeraSort, the evenness of the partition determines the maximum cost of network traffic and storage on any single computation unit, and hence the execution time of the whole algorithm. [29] concludes that TeraSort is load-balanced within constant multiplicative factors. That is, if the data set has n elements and t machines are

[6]Here we provide the full detail of our construction. We calculate the sizes of each bucket, enough to determine the partition. Let $y_1, y_2, ..., y_{t-1}$ be the sizes of the buckets of an ordered even $(t-1)$-partition (not t) of the s elements. Then the sizes for our need are $\lceil \frac{y_1}{2} \rceil, y_2, ..., y_{t-1}, \lfloor \frac{y_1}{2} \rfloor$. Comparing the sizes here with those in the argument for $t - 1|rk$ leads to the increase of constant factors in the bounds.

used in the algorithm, then every machine has $O(\frac{n}{t})$ workload. The constant factor in the proof is relatively large (16 to 32). Moreover, the proof itself does not extend to substantially smaller bounds.

Asymptotic optimality. This report seeks more accurate description of the performance of TeraSort with self-sampling. Under the same constraints in [29], it gives a series of bounds which describes the trade-off among the number of machines and the evenness of the partition (hence the worst case maximum workload on a single machine). Larger numbers of machines allow more even partitions. Out of these bounds, an asymptotically optimal one is stated explicitly. As t grows, the workload on any machine converges to m (not only $O(m)$) with probability at least $1 - O(\frac{1}{m})$.

Top-k/k-selection. The report also extends the use of the partition technique to the top-k and k-selection problems in the MapReduce world.

1.4 Outline

Chapter 2 reviews the basics MapReduce and declares the conventions adopted in this paper. Chapter 3 focuses on a careful analysis of the probabilistic structure of self-sampling in TeraSort. It first proves the minimality already shown in [29] but with a different analysis of probability. Then it shows that parameters meeting certain condition, there is a strong guarantee on the evenness of the partition scheme. One assignment of parameters leads to a nearly optimal bound. Chapter 5 describes an efficient MapReduce algorithm to the top-k/k-selection problem in which the same random partition scheme is applied. Chapter 6 concludes the paper.

Chapter 2

Preliminaries

2.1 MapReduce

The MapReduce computation model describes an algorithm in rounds, each of which consists of three phases, map, shuffle and reduce. Rounds are sequentially connected by feeding the output of the reduce phase to the map phase of the next round. Figure 2.1 depicts a high-level workflow of a MapReduce round.

Map. Workers called mappers are given input of items. For every item, a set of key-value pairs $(k; v)$ is generated into the output. We emphasize that mappers keep no track of any information of the history of processed data.

Shuffle. In this phase, pairs output by the map phase are conceptually shuffled into reducers, which are workers in the reduce phase. The destination of a pair $(k; v)$ solely depends on the value of k; that is, pairs sharing the same key value will end up in the same reducer.

Reduce. Each reducer perform local computation upon the set of received key-value pairs. As opposed to mappers, the reducer operate on a set of items rather than one at a time. It may generate any items either as the input of the map phase of the next round, or as the output of the entire algorithm if the current round is the last.

We take table joining as an example of MapReduce algorithms. Suppose that data tables are stored as collections of records (i.e. data rows) and it is very efficient to retrieve the value of the column named c from a record. (This fits the typical implementations of data tables in practice.) The MapReduce algorithm which computes

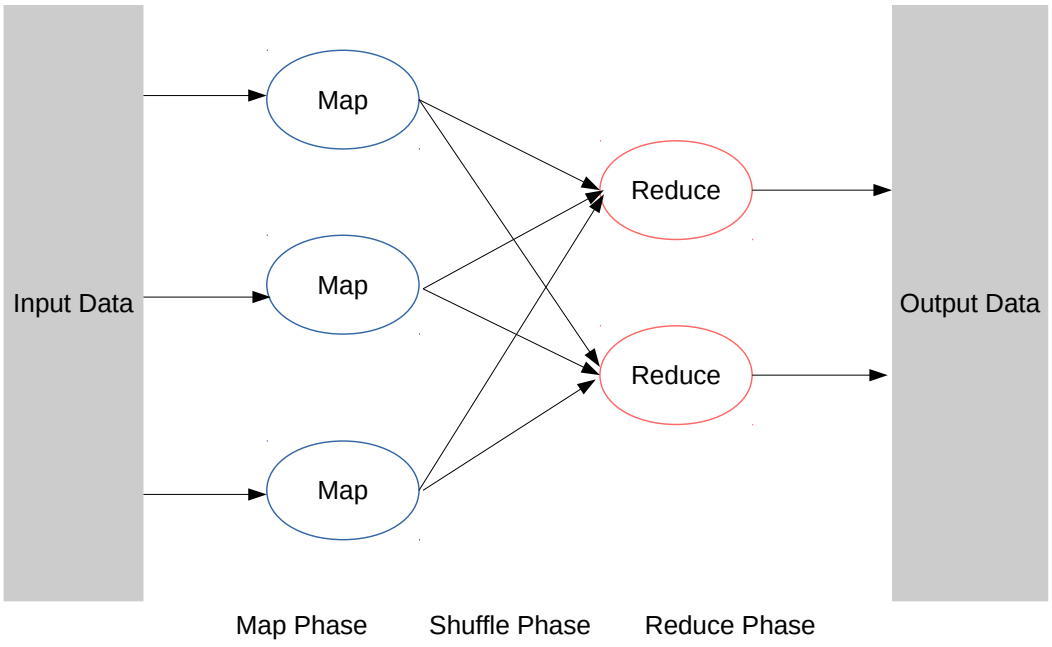

Figure 2.1: One round of MapReduce computation

the join of table T_1 and T_2 on $T_1[c_1] = T_2[c_2]$ is described below. Let h be randomly picked from a universal hash family with sufficient range $[N]$, in order to numerate $r[c_i]$ and avoid reducer of undesirably large size.

Map-Shuffle 1 Records of $T_i, i = 1, 2$ are read from the storage of tables. Each record $r \in T_i$ is mapped to $(r[c_i]; i, r)$, which is then shuffled into reducer $h(r[c_i])$.

Reduce 1 Denote all received pairs in a reducer as R; R may contain pairs with different keys. Let $R_i(k), i = 1, 2$ denote the (multi-)set $\{\, r \mid (k; i, r) \in R \,\}$. $R_i(k)$ in fact consists of all records of T_i whose column c evaluates to k. In this way, we compute the join of $R_1(k)$ and $R_2(k)$ for every k appearing in the reducer as the output. The output of all reducers is the join of T_1 and T_2 expressed as collection of records.

Mappers and reducers are logical constructs. In an implementation of MapReduce, they are scheduled onto physical computation nodes. In cases where there are more reducers than available computation nodes, for lower overall processing time, the scheduler may keep a single node exclusive to one large reducer while having several smaller reducers share the same node. Simple greedy scheduling algorithms serve this purpose well. This level of scheduling is invisible to MapReduce algorithms.

2.2 Settings of The Report

We introduce the notation and convention in this paper. Instead of mappers and reducers, we call the workers as machines. This reflects the fact that the TeraSort algorithm is applicable to a wide range of distributed computation models and is most naturally phrased in terms of computation nodes (machines). Fitting the algorithm into MapReduce implementations such as Hadoop is easy.

Suppose that t machines store the data set of interest. They are indexed from 1 to t, namely $\mathcal{M}_1, ..., \mathcal{M}_t$. $\mathcal{M}_i$ sending element x to $\mathcal{M}_j$ corresponds to, in the MapReduce language, $\mathcal{M}_i$ mapping x to $(j; x)$ which is shuffled to reducer j executed on $\mathcal{M}_j$. To accommodate the statelessness of mappers and reducers, we assume that unmentioned data in the algorithm is "carried forward" implicitly to the reducer with the same index in the next round.[1]

As mentioned earlier, the focus of our analysis is on TeraSort with self-sampling.

[1] In a MapReduce system with explicit control over access to shared storage, it is not necessary to "carry forward" unused data in implementing TeraSort. Take Hadoop as an example. The partition

2.3 Probability Tools

In this section we briefly review the two major tools that we use in analyzing probabilities in this paper - the union bound and Chernoff bounds.

Union bound. The union bound reflects the fact that the probability of a union of events is always smaller than the sum of probabilities of the events. Formally, let E_i be events, we have,

$$\mathbf{Pr}\{\cup_i E_i\} \leq \sum_i \mathbf{Pr}\{E_i\}$$

The union bound is especially useful in limiting the probability of bad events, which in turn gives a desired lower bound of probability of good events.

Chernoff bounds. Chernoff bounds restricts from above the tail probability of sums of independent Bernoulli random variables. There are several forms/variants of Chernoff bounds of similar restrictive power. In this report, we use the following form:

$$\mathbf{Pr}\left\{\sum_{i=1}^{n} X_i > (1+\delta)\mu\right\} \leq \exp\left\{-\frac{\delta^2\mu}{\delta+2}\right\},\ \delta > 0$$

$$\mathbf{Pr}\left\{\sum_{i=1}^{n} X_i < (1-\delta)\mu\right\} \leq \exp\left\{-\frac{\delta^2\mu}{2}\right\},\ 0 < \delta < 1$$

where $X_i = 1, 0$ with probability $p_i, 1-p_i$ respectively and $\mu = \sum_{i=1}^{n} p_i$; X_i's are independent.

elements are written into a shared file at the end of the partition step. The data set can then be processed in a second MapReduce job which utilizes the partition elements in mappers.

Chapter 3

A New Proof of TeraSort's Minimality

3.1 TeraSort

In TeraSort, we use the notion of ordered even t-partition, which divides a set as evenly as possible, offering a close approximation to the ideal case of absolute evenness.

Definition 1 (ordered t-partition)**.** An ordered t-partition divides an ordered set of s elements into t subgroups. Elements of subgroup i is smaller than those of subgroup j for all $i < j$.

The first element of every partition except for the first one is called t-partition element as they describe the subgroups in full.

Definition 2 (ordered even t-partition)**.** An ordered even t-partition is an ordered t-partition in which sizes of the subgroups differ by at most 1.

An ordered even partition always exist for a data set. In fact, we may construct it in the following way. Let $s = ts_1 + s_2$ where $s_1 = \left\lfloor \frac{s}{t} \right\rfloor$. The indices of the partition elements are

$$d_j = s_1 + jI_{\{j \leq s_2\}},\ I_{\{\cdot\}} \text{ is the indicator function}$$

Another popular scheme, in which ranks of partition elements are $\left\lceil \frac{s}{t} \right\rceil, 2\left\lceil \frac{s}{t} \right\rceil, ..., (t-1)\left\lceil \frac{s}{t} \right\rceil$, produces less balanced buckets: when $s \equiv 1 \pmod{t}$, the difference is as large as $t - 2$. Furthermore, this scheme simply fails if $t > \sqrt{s} + 1$ when $s \equiv 1 \pmod{t}$

because the rank of the $(t-1)$-th partition element is already beyond s. Such circumstances *can* occur in TeraSort. In Jim Gray's sort benchmark, the size of data set to sort is $100TB$ and the winning TeraSort solutions uses above 1000 reducers ($t > 1000$). Considering that the expected sample size is $t\ln(nt)$ (n is the size of input data set, shown later) in TeraSort, the failure happens as long as $\ln(nt) \mod t$ coincides to 1. In summary, the scheme fails to capture the notion of "partition".

The TeraSort algorithm sorts n elements across t machines $\mathcal{M}_1, ..., \mathcal{M}_t$ in 2 rounds.

Map 1 Each element is selected into the sample S with probability ρ.

Reduce 1 S is sent to $\mathcal{M}_1$. On this machine, b_j, $j = 1, ..., t-1$ form an ordered even t-partition of S. On all other machines, no operation is performed.

Map 2 Assume that b_j's have been broadcast to all machines.[1] Element x is sent to $\mathcal{M}_j$ if $b_{j-1} \leq x < b_j$, where $b_0 := -\infty$ and $b_n := +\infty$.

Reduce 2 On each machine, sort elements locally.

In the rest of this section, we give a new proof of the claims in [29] to prove the minimality of TeraSort, but by a more extensible probabilistic method that is capable of generating a series of tight bounds (i.e. the constant coefficient converging to one) shown in the next Chapter.

The minimality of TeraSort is equivalent to the following claims. Claim 1 limits the size of the sample S and thus the amount of traffic that machines send and receive in the first round. Claim 2 limits the reducer sizes and the network input in the second round, since the map phase of round 2 never violates minimality as long as every machine holds $O(m)$ elements at the beginning of the algorithm.

Claim 1. *With selected ρ, in Map 1, $|S| = O(m)$, w.h.p.*

Claim 2. *With selected ρ, in Reduce 2, every machine ends up with $O(m)$ elements, w.h.p.*

[1] As noted in [29], the broadcast assumption may incur a network outflow of size $\Phi(t^2)$ at $\mathcal{M}_1$, which would make TeraSort unminimal when t^2 is no longer $O(m)$. However, in practice the broadcast can be implemented in Hadoop as $\mathcal{M}_1$ writing to a shared file which is then read by all machines. This way, the broadcast cost is evenly distributed among machines.

3.2 Proof of Minimality

It is straightforward to show Claim 1.

Lemma 1. *With* $\rho \geq \frac{1}{m} \ln nt$ *and* $k \geq 3$,

$$\mathbf{Pr}\{\, |S| > kn\rho \,\} \leq \left(\frac{1}{nt}\right)^t$$

Proof. $|S|$ is the sum of n Bernoulli random variables of probability ρ; $\mathbb{E}[\,|S|\,] = n\rho$. By Chernoff bounds, we have

$$\begin{aligned}\mathbf{Pr}\{\, |S| > kn\rho \,\} &\leq \exp\left\{ -\frac{(k-1)^2}{k+1} n\rho \right\} \\ &\leq \left(\frac{1}{nt}\right)^t\end{aligned}$$

□

Theorem 1 (Claim 1)**.** *By setting* $\rho \geq \frac{1}{m} \ln nt$ *and assuming* $m > n\rho$*, the Claim 1 hold with probability* $1 - O(\frac{1}{(nt)^t})$.

Proof. By Lemma 1, $|S|$ is bound w.h.p,

$$\begin{aligned}\mathbf{Pr}\{\, |S| > 3m \,\} &\leq \mathbf{Pr}\{\, |S| > 3n\rho \,\} \\ &\leq \left(\frac{1}{nt}\right)^t\end{aligned}$$

The event that Claim 1 holds is within the complement of the above. Therefore the claim holds with high probability. □

Claim 2 is more involved in that the identification of the partition elements b_i's depend on the entire sample process: if one more element is included in the sample, we may end up with multiple different partition elements. For example, suppose $|S| \equiv -1 \pmod{t}$. Adding one more element to S as the smallest of S leads to an entire new set of partition elements. This phenomenon is occur in all partition schemes which aims for evenness.

We present our approach in the form of a few interesting lemmas. In the first place, we formulate a problem closely related to Claim 2.

Problem 1 (Sample-Partition)**.** Let A, called the total set, denote the set of n elements from an ordered universe; a_j denotes the $(j+1)$-th smallest element in A. Construct a sample $S \subseteq A$ by independently picking each element with probability ρ. Let $b_1, b_2, ..., b_{t-1} \in S$ be the ordered even t-partition elements.

Question: how evenly does b_i's partition A?

Problem 1 captures the probabilistic structure of TeraSort with self-sampling. Clearly, the set of the elements on $\mathcal{M}_i$ in Reduce 2 is exactly $A \cap [b_{i-1}, b_i)$, independent of how the data set (the total set A in the definition of the problem) is spread across machines. The answer of Problem 1 that the partitions are all $O(m)$ in size proves Claim 2.

Suppose an ordered partition of A. Every partition is a bucket. It is easy to observe that if every bucket contains a marked element, then the distance between any two adjacent marked elements is less than the sum of the sizes of the buckets in which they exist. The observation will lead to Claim 2 if we additionally ensure that buckets are $O(m)$ in size. This is the approach in [29], as illustrated in Figure 3.1.

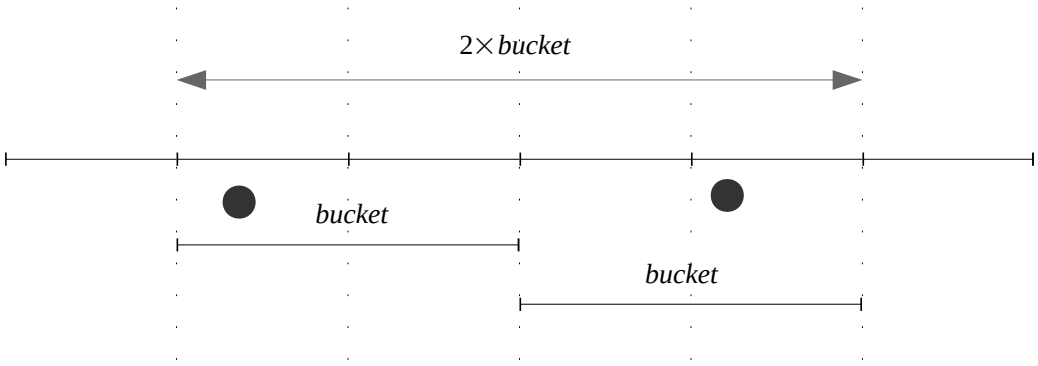

Figure 3.1: If every bucket contains a marked element, then the distance between no two adjacent marked elements is less than the sum of the sizes of the buckets in which they exist.

We extend it with a stronger observation: if we overlay buckets with one another, we will end up having more buckets and a promise of shorter distance between adjacent marked elements. Formally, consider an ordered even t-partition of A. Let $d(i)$ be the index in A of the i-th smallest partition element; we also manually set $d(0) := 1$. Let $I_j := [a_{d(j)}, a_{d(j)+lm})$. $l > 0$ controls the length of the interval. I_j's are well defined in this way for all $j \geq 0$ with $d(j) + lm \leq n - 1$. If the largest a few elements are still left, we cover them with one additional interval $[a_{n-lm}, a_n]$.

The intervals form a cover of A. There could be at most t intervals. Figure 3.2 3.3 illustrate such intervals.

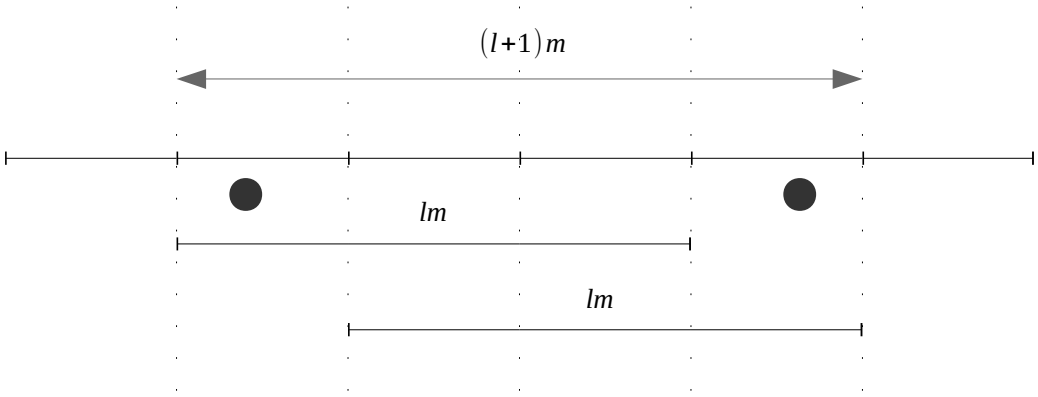

Figure 3.2: This compacts the placement of buckets in Figure 3.1, but still only works in l being integers.

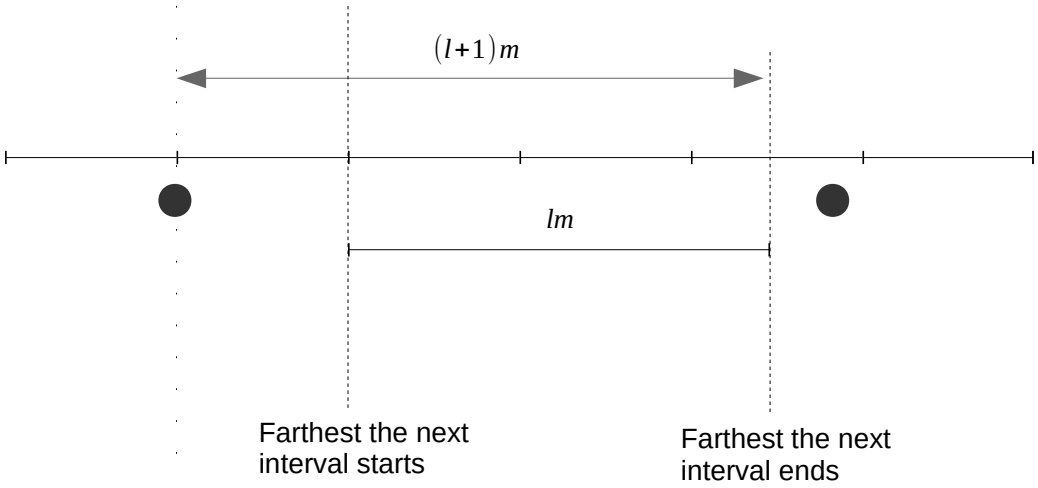

Figure 3.3: Lemma 2 extends our intuition to l being any positive real. The next interval contains no marked element if the next marked element is more than $(l+1)m$ away.

Lemma 2. *Suppose we marked a few elements in A. If every interval I_j has at least one marked element, then no two marked elements are more than $(l+1)m$ away from each other, $l > 0$.*

Proof. Let $\{ e_i \}_{i \geq 1}$ represent the marked elements, in increasing order. Suppose the opposite, that $|[e_i, e_{i+1}]| > (l+1)m$, then there exists an interval $I_j \subseteq [e_i, e_{i+1}]$ which contains no partition element.

Indeed, we start at e_i and walk towards larger elements by $(l+1)m$ steps. Since $|[e_i, e_{i+1}]| > (l+1)m$, we must have not met another marked elements yet. The next interval of the one containing e_i starts at most m away from e_i (this is the spacing of intervals), so its end cannot pass where we are now. This interval contain no marked element. □

Note that Lemma 2 even applies to intervals whose sizes are not multiples of m. (l could be any positive real.)

Now we only need to put a ceiling on the probability that some interval contains no marked element, which, in our case, refers to the partition elements b_j's. They are shown in Lemma 3 and Lemma 4. Also note that S is the random sample taken from A (Problem 1).

Lemma 3. *Fix an arbitrary subset $B(x)$ of size x of A. Then $B(x) \cap S$ denotes the set of selected elements in $B(x)$. With $\rho \geq \frac{1}{m} \ln nt$ and $l \geq 7$,*

$$\mathbf{Pr}\left\{ |B(lm) \cap S| < \frac{|S|}{t} \right\} \leq \frac{1}{nt} + \left(\frac{1}{nt}\right)^t$$

.

Proof. Take condition on $|S| > kn\rho, 3 \leq k \leq l$ and decompose the probability,

$$\begin{aligned}
&\mathbf{Pr}\left\{ |B(lm) \cap S| < \frac{|S|}{t} \right\} \\
&\leq \mathbf{Pr}\left\{ |B(lm) \cap S| < \frac{|S|}{t} \text{ and } |S| \leq kn\rho \right\} + \mathbf{Pr}\left\{ |B(lm) \cap S| < \frac{|S|}{t} \text{ and } |S| > kn\rho \right\} \\
&\leq \mathbf{Pr}\left\{ |B(lm) \cap S| < \frac{kn\rho}{t} \right\} + \mathbf{Pr}\left\{ |S| > kn\rho \right\}
\end{aligned}$$

By Lemma 1 we are able to bind the above. When $k := 3$ and $l \geq 7$,

$$\begin{aligned}
\mathbf{Pr}\left\{ |B(lm) \cap S| < \frac{|S|}{t} \right\} &\leq \exp\left\{ -\frac{(l-k)^2}{2l} m\rho \right\} + \left(\frac{1}{nt}\right)^t \\
&\leq \frac{1}{nt} + \left(\frac{1}{nt}\right)^t
\end{aligned}$$

□

Lemma 4. *b_i's are partition elements defined above. Then for any $0 \leq i \leq t-1$,*

$$\mathbf{Pr}\left\{\, |A \cap [b_i, b_{i+1}]| \leq 8m \,\right\} \leq O\left(\frac{1}{n}\right)$$

, where $b_0 := -\infty$ and $b_t := +\infty$.

Proof. In A, if a block of consecutive elements in the ordered manner contain no partition element, then they must contribute no more than $\left\lceil \frac{|S|}{t} \right\rceil$ to the sample S. Along with Lemma 3,

$$\begin{aligned}
\mathbf{Pr}\left\{\, I_j \text{ has no partition element} \,\right\} &\leq \mathbf{Pr}\left\{\, |B(lm) \cap S| < \left\lceil \frac{|S|}{t} \right\rceil \,\right\} \\
&= \mathbf{Pr}\left\{\, |B(lm) \cap S| < \frac{|S|}{t} \,\right\} \\
&\leq \frac{1}{nt} + \left(\frac{1}{nt}\right)^t
\end{aligned}$$

By union bound, with probability $1 - O(\frac{1}{n})$ every interval covers at least one partition element given that $l \geq 7$. By Lemma 2, this leads to the lemma. □

Theorem 2 (Claim 2)**.** *By setting $\rho \geq \frac{1}{m} \ln nt$, Claim 2 hold with probability $1 - O(\frac{1}{n})$.*

Chapter 4

Optimal Bounds for TeraSort

4.1 A series of bounds

At this point, we ask ourselves, whether a fixed constant coefficient before m is the farthest we could go with the tools in hand; moreover, whether there is a point in pursuing better guarantees of evenness. In the particular problem of TeraSort, since the initial layout of the data set also affect and possibly dominates the network throughput of machines, one may be already content with fixed constant coefficients. However, we see the problem of Sample-Partition as a simple, generic routine potentially used for various problems. Thus it is rewarding to pushing the bound downwards.

Our analysis is based on a re-examination of the proof of Lemma 3 and Lemma 4. We already obtain the link of the event that an interval having no partition element and the one that it contributes no more than $\left\lceil \frac{|S|}{t} \right\rceil$ elements to the sample S. Yet in decoupling $|B(lm) \cap S|$ and $\left\lceil \frac{|S|}{t} \right\rceil$, we loosely transform $\left\lceil \frac{|S|}{t} \right\rceil$ to $\lceil \frac{kn\rho}{t} \rceil$, losing the information that the partition are even. It turns out that adding it back results in surprisingly tight bounds.

We focuses on tighter bounds in Claim 2 (the evenness of partition), though as is shown later the choice of parameters also ensures Claim 1.

Theorem 3. *Following the denotation in Lemma 3, with* $\rho \geq \frac{1}{m} \ln nt$ *and* l, t *satisfying the constraints below,*

$$\begin{cases} (k-1)^2(t-l) \geq (k+1) \\ ((t-1)l-(t-l)k)^2 \geq 2(t-1)l \\ (t-1)l > (t-l)k \\ k,t>1,\ l>0 \end{cases} \tag{4.1}$$

We have,

$$\mathbf{Pr}\left\{ |B(lm) \cap S| < \frac{|S|}{t} \right\} \leq \frac{2}{nt}$$

Proof. Let Y_j be the indicator random variable representing whether element $a_j \in A$ is picked into S. Let $W(x)$ denotes the sum of x independent Bernoulli(ρ) random variables; we require that there is no dependency between the random variables underlying two $W(\cdot)$ expressions.

$$\begin{aligned} \mathbf{Pr}\left\{ |B(lm) \cap S| < \frac{|S|}{t} \right\} &= \mathbf{Pr}\left\{ \sum_{j:a_j \in B(lm)} Y_j < \sum_{j=0}^{n-1} Y_j \right\} \\ &= \mathbf{Pr}\left\{ (t-1) \sum_{j:a_j \in B(lm)} Y_j < \sum_{j:a_j \notin B(lm)} Y_j \right\} \\ &= \mathbf{Pr}\left\{ (t-1)W(lm) < W(n-lm) \right\} \end{aligned} \tag{4.2}$$

To set an upper bound on (4.2), we use the same decoupling technique in Lemma 3 again. Let,

$$\begin{aligned} W_1 &= W(lm) \\ W_2 &= W(n-lm) \end{aligned}$$

we have,

$$\begin{aligned} &\mathbf{Pr}\left\{ (t-1)W(lm) < W(n-lm) \right\} \\ &\leq \mathbf{Pr}\left\{ W_1 < \frac{W_2}{t-1},\ W_2 \leq k(n-lm)\rho \right\} + \mathbf{Pr}\left\{ W_1 < \frac{W_2}{t-1}, W_2 > k(n-lm)\rho \right\} \\ &\leq \mathbf{Pr}\left\{ W_1 < \frac{k(n-lm)\rho}{t-1} \right\} + \mathbf{Pr}\left\{ W_2 > k(n-lm)\rho \right\} \end{aligned}$$

By Chernoff bounds, $k > 1$ and $(t-1)l > (t-l)k$,

$$\mathbf{Pr}\{\, W_2 > k(n-lm)\rho \,\} \leq \exp\left\{ -\frac{(k-1)^2}{k+1}(t-l)m\rho \right\} \tag{4.3}$$

$$\mathbf{Pr}\left\{ W_1 < \frac{k(n-lm)\rho}{t-1} \right\} \leq \exp\left\{ -\frac{1}{2}(1-\frac{(t-l)k}{(t-1)l})^2(t-1)lm\rho \right\} \tag{4.4}$$

Let (4.3) and (4.4) be under $\frac{1}{nt}$. Equation system (4.1) is simply a rearrangement of terms plus sanity conditions. □

We remark that although Theorem 3 focuses on Claim 2, condition (4.1) also ensures that Claim 1 holds, because $(k-1)^2(t-l) \geq (k+1)$ is stronger than $(k-1)^2 t \geq (k+1)$.

Theorem 3 describes a family of bounds $(l+1)m$ by admissible l and t. The choice of l and t also interprets the trade-off between the evenness and number of partitions. As t increases, lower l is accessible; larger number of partitions implies greater evenness. Corollary 1 is a recipe of bounds, $\left\{ (2+2t^{\epsilon-\frac{1}{2}})m \right\}_t$, $0 < \epsilon < \frac{1}{2}$.

Corollary 1. *Following the denotation in Lemma 3 and assume $\rho \geq \frac{1}{m}\ln nt$. Let $0 < \epsilon < \frac{1}{2}$ be a parameter and set $l := 1 + 2t^{\epsilon-\frac{1}{2}}$. If t satisfies*

$$t - t^{1-2\epsilon} - 2t^{\epsilon-\frac{1}{2}} - 1 > 0$$

then it holds that,

$$\mathbf{Pr}\left\{ |B(lm) \cap S| < \frac{|S|}{t} \right\} \leq \frac{2}{nt}$$

Note that the condition for t in the corollary always holds for large enough t regardless of the choice of ϵ. Therefore, we obtain arbitrarily strong evenness as long as t is allowed to be sufficiently large. To see this clearer, let $b := t^{\epsilon-\frac{1}{2}}$ and the bound becomes $(2+2b$. Dropping b into the corollary produces an equivalent condition $2b^3 + (1-t)b^2 + 1 < 0$, which illustrates the relationship between the tightness of the bound and t. For any given t, let b be root in $(\frac{1}{\sqrt{t}}, 1)$ of x's equation $2x^3 + (1-t)x^2 + 1 = 0$, and this represents the bound $(2+2b)m$. We conclude the discussion as 2.

Corollary 2. *With probability at least $1 - O(\frac{1}{m})$, the size of any subgroup is $(2+2b)m$ where $\frac{1}{\sqrt{t}} < b < 1$ is determined by $2b^3 + (1-t)b^2 + 1 = 0$. Moreover, such b always exists.*

As t increases, b nears $\frac{1}{\sqrt{t}}$ downwards and always stays above; but b quickly get close enough that the difference is negligible. In fact, it can be shown via calculus that when $t \geq 5$, we have $b < \frac{1}{\sqrt{t}} + \frac{1}{t}$. By combining all of above, we arrive Corollary 3.

Corollary 3. *Following the denotation in Lemma 4. With* $t \geq 5$*,*

$$\mathbf{Pr}\left\{ |A \cap [b_i, b_{i+1}]| > (2 + \frac{2}{\sqrt{t}} + \frac{2}{t})m, \text{ for } 0 \leq i \leq t-1 \right\} \leq O\left(\frac{1}{n}\right)$$

where $b_0 := -\infty$ *and* $b_t := +\infty$*.*

4.2 More tolerance to failures

In cases where $O(\frac{1}{m})$ is already negligible[1], we are able to further restrict the size of the subgroups. The trick is that in constructing the intervals $\{ I_j \}_j$, instead of placing the left endpoint of the next interval m away from that of the current one, we put it $\frac{m}{t}$ apart (see Figure 4.1). As a result,

- Lemma 2 has a stronger form: no two adjacent marked elements are more than $(l + \frac{1}{t})$ away from each other.
- when we apply union bound with the at most t^2 intervals, we obtain a probability of failure no larger than $O(\frac{1}{nt}) \times t^2 = O(\frac{1}{m})$.

This notion augments every bound shown in the previous section. We phrase the counterpart of Corollary 3 as an example.

Theorem 4. *Following the denotation in Lemma 4. With* $t \geq 5$*,*

$$\mathbf{Pr}\left\{ |A \cap [b_i, b_{i+1}]| > (1 + \frac{2}{\sqrt{t}} + \frac{3}{t})m, \text{ for } 0 \leq i \leq t-1 \right\} \leq O\left(\frac{1}{m}\right)$$

where $b_0 := -\infty$ *and* $b_t := +\infty$*.*

[1]This is reasonable because nowadays the main memories of computation nodes are on the order of gigabytes. Main memory computation is very fast but if a reducer is too large, then computation involving secondary storage must be used which usually suffers from serious I/O bottlenecks. If we consider m (the average workload) to be safely within the capacity of main memory (e.g. m is on the order of 2^{20} while the size of a single object is less than 2^{10} bytes), then $\frac{1}{m}$ is very small, on the order of 2^{-20}.

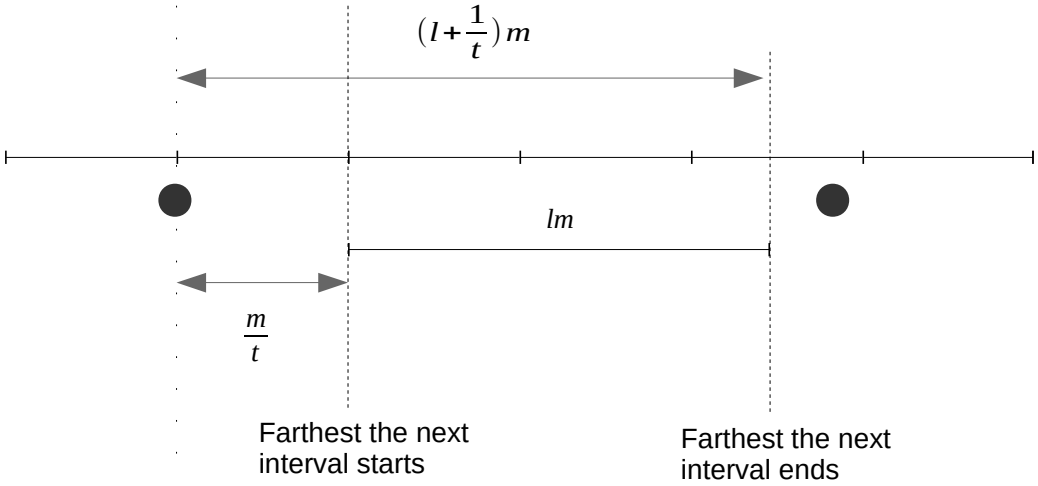

Figure 4.1: With $\frac{m}{t}$ spacing, Lemma 2 is strengthened to binding adjacent marked elements within $(l + \frac{1}{t})$. Obviously, it works with any spacing.

Chapter 5

Top-k and k-Selection Problems

The uniformity of the partition scheme imply an efficient solution to k-selection problem, in which given an index k, the k-th smallest element of the data-set stored across $\mathcal{M}_1$ to $\mathcal{M}_t$ should be output. With the knowledge of the number of elements in every partition, every machine could tell the location of the k-th smallest item and the carrier would select it by computing a local offset index. We describe the algorithm in detail below.

Map 1 Each element is selected into the sample S with probability ρ.

Reduce 1 S is sent to $\mathcal{M}_1$. On this machine, b_j, $j = 1, ..., t-1$ form an ordered even t-partition of S. On all other machines, no operation is performed.

Map 2 Assume that S has been broadcast to all machines. Element x is sent to $\mathcal{M}_j$ if $b_{j-1} \leq x < b_j$, where $b_0 := -\infty$ and $b_n := +\infty$. Denote the number of local elements of $\mathcal{M}_i$ sent to $\mathcal{M}_j$ as $n(i, j)$; that is, $n(i, j) := |\{ x \in \mathcal{M}_j \,|\, b_{j-1} \leq x < b_j \}|$. Define

$$n(i, < j) := \sum_{k<j} n(i, k)$$
$$n(i, > j) := \sum_{k>j} n(i, k)$$

To each $\mathcal{M}_j$, three numbers $n(i, j), n(i, < j), n(i, > j)$ are also sent.

Reduce 2 On every $\mathcal{M}_i$, let

$$n(< i) := \sum_{j=1}^{t} n(j, < i)$$
$$n(> i) := \sum_{j=1}^{t} n(j, > i)$$
$$n(i) := \sum_{j=1}^{t} n(j, i)$$

which respectively represent the number of elements in the data-set that are smaller, larger and within the partition residing in the machine. The sum of three numbers is the size of the data set n. In this way $\mathcal{M}_i$ is able to decide whether the k-th smallest element is local by judging $n(< i) < k \leq n(i)+n(< i)$.

Only one machine has positive decision. In this case, it outputs the $(k-n(< i))$-th smallest local element via any local selection algorithm.

The first round is identical to that of TeraSort. In the second round, each machine sends and receives $O(t)$ size messages. Hence the algorithm shares the same efficiency characteristics with TeraSort. It can also be easily extended to compute multiple queries at the same time.

The above k-selection algorithm is transformed into a top-k algorithm by letting the machines with $n(< i) < k$ output all the local storage and the single machine with $n(< i) < k \leq n(i) + n(< i)$ output the top $(k - n(< i))$ local elements.

Chapter 6

Conclusions

In the report, we have analyzed the probabilistic properties of the self-sampling strategy used in TeraSort and conclude that with high probability it yields nearly even partitions. Specifically, we have shown,

- *smaller constant.* Our new method leads to smaller constant in $O(m)$ given
- *asymptotic optimality.* As the size of input data set and the number of partitions grow, the unevenness tends to vanish.
- *top-k and k-selection.* The sample-partition technique in TeraSort leads to efficient solutions to the top-k and k-selection problem in MapReduce framework.

The asymptotic optimality indicates the significance of the partitioning technique, in that it could be used as a reliable building block in many scenarios where partitioning help solve the problem. As an example, we state an efficient MapReduce algorithm for the k-selection problem.

Bibliography

[1] Foto N Afrati and Jeffrey D Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1282–1298, 2011. optimize against lowest communication cost.

[2] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*, 2011.

[3] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM. equi-join, logs.

[4] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[5] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[8] David J DeWitt, Jeffrey F Naughton, and Donovan A Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Parallel and distributed information systems, 1991., proceedings of the first international conference on*, pages 280–291. IEEE, 1991. no proof.

[9] Jon Feldman, S Muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Transactions on Algorithms (TALG)*, 6(4):66, 2010.

[10] W Donald Frazer and AC McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)*, 17(3):496–507, 1970. 13.

[11] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. *Database system implementation*, volume 654. Prentice Hall Upper Saddle River, NJ:, 2000.

[12] T Graves. Graysort and minutesort at yahoo on hadoop 0.23. *http://sortbenchmark.org/Yahoo2013Sort.pdf*, 2013. 2013 sort benchmark, terasort report.

[13] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961. quickselect.

[14] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. Society for Industrial and Applied Mathematics, 2010.

[15] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 85–94. ACM, 2011.

[16] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[17] Ahmed Metwally and Christos Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proceedings of the VLDB Endowment*, 5(8):704–715, 2012.

[18] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 949–960. ACM, 2011.

[19] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[20] Owen OMalley. Terabyte sort on apache hadoop. *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)*, pages 1–3, 2008. original terasort report.

[21] Owen OMalley and Arun C Murthy. Winning a 60 second dash with a yellow elephant, 2009. 2009 sort benchmark, terasort report.

[22] Steven J Plimpton and Karen D Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[23] John H. Reif and Leslie G. Valiant. A logarithmic time sort for linear size networks. *J. ACM*, 34(1):60–76, January 1987. random routing, complicated splitting strategy, N nodes of logN memory.

[24] Steven R Seidel and William L George. Binsorting on hypercubes with d-port communication. In *Proceedings of the third conference on Hypercube concurrent computers and applications-Volume 2*, pages 1455–1461. ACM, 1989. 25.

[25] Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *J. Parallel Distrib. Comput.*, 14(4):361–372, April 1992. deterministic, 2 worst case, t(t-1) samples, n¿t3.

[26] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010. HDFS.

[27] Xueyuan Su and Garret Swart. Oracle in-database hadoop: When mapreduce meets rdbms. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 779–790, New York, NY, USA, 2012. ACM.

[28] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011.

[29] Yufei Tao, Wenqing Lin, and Xiaokui Xiao. Minimal mapreduce algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 529–540, New York, NY, USA, 2013. ACM.

[30] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[31] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846. ACM, 2009.

[32] Rares Vernica, Michael J Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506. ACM, 2010.

[33] Milan Vojnovic, Fei Xu, and Jingren Zhou. Sampling based range partition methods for big data analytics. Technical report, Technical report, Microsoft Research, 2012. not self-sampling.

[34] Jiamang Wang, Yongjun Wu, Hua Cai, Zhipeng Tang, Zhiqiang Lv, Bin Lu, Yangyu Tao, Chao Li, Jingren Zhou, and Hong Tang. Fuxisort. *http://sortbenchmark.org/FuxiSort2015.pdf*, 2015. 2015 sort benchmark, fuxisort report.

[35] Youngju Won and Sartaj Sahni. A balanced bin sort for hypercube multicomputers. *The Journal of Supercomputing*, 2(4):435–448, 1988. 29.

[36] Xiaofei Zhang, Lei Chen, and Min Wang. Efficient multi-way theta-join processing using mapreduce. *Proceedings of the VLDB Endowment*, 5(11):1184–1195, 2012.