Datacenter-Network-Aware Online Load Balancing in MapReduce

by

Yanfang Le

B.Eng., Zhejiang University, China, 2011.

Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Summer 2015

Approval

Name:	Yanfang Le
Degree:	Master of Science (Computing Science)
Title:	*Datacenter-Network-Aware Online Load Balancing in MapReduce*
Examining Committee:	**Dr. Petra Berenbrink** (chair) Associate Professor

Dr. Jiangchuan Liu
Co-Senior Supervisor
Professor

Dr. Funda Ergün
Co-Senior Supervisor
Professor

Dr. Jian Pei
Supervisor
Professor

Dr. Nick Sumner
Internal Examiner
Assistant Professor

Date Defended: 10 July 2015

Abstract

MapReduce has emerged as a powerful tool for distributed and scalable processing of voluminous data. Different from earlier heuristics in the very late reduce stage or after seeing all the data, we address the skew from the beginning of data input, and make no assumption about a priori knowledge of the data distribution nor require synchronized operations. We show that the optimal strategy is a constrained version of online minimum makespan and, in the MapReduce context where pairs with identical keys must be scheduled to the same machine, we propose an online algorithm with a provable 2-competitive ratio. We further suggest a sample-based enhancement, which, probabilistically, achieves a 3/2-competitive ratio with a bounded error. Examine the project again, we found that the datacenter network could be a bottleneck in the shuffle subphase of MapReduce. This could potentially lead to a poor overall performance even with a balanced workload and thus needed to be addressed. Earlier studies either assume the network inside a datacenter is of negligible delay and infinite capacity, or use a hop count as the network cost measurement. We consider the realistic bandwidth constraints in real world datacenter networks and propose an effective solution toward near optimal datacenter-network-aware load balancing.

Keywords: Load Balancing; MapReduce; Online; Datacenter Network

Dedication

"Hope for the best, prepare for the worst."

Acknowledgements

I would first like to thank my senior supervisors, Dr. Jiangchuan Liu and Dr. Funda Ergün, who have offered invaluable guidance and support to me through out my thesis from the beginning to the end with their knowledge and patience. Whenever I needed help, they made their time available and helped me with their valuable guidance and advice to tackle any of my problems. It was impossible for me to finish this journey without their warm encouragement and nice help.

I also want to thank Dr. Jian Pei and my thesis examiner Dr. Nick Sumner, for reviewing this thesis and providing helpful suggestions that helped me to improve the quality of the thesis. I also want to thank Dr. Petra Berenbrink for taking the time to chair my thesis defense.

My colleagues and friends not only provided help in my studies, but also in my everyday life. The time with them is unforgettable. I am deeply indebted to them.

Finally, I want to thank my family for their love and care. Nothing would happen without their supports. Thank you all!

Table of Contents

List of Figures

Chapter 1

Introduction

With the rapid growth of information in such applications as social networking and bioinformatics, there is an urgent need for large-scale data analysis and processing. With recent advances in Cloud Computing, MapReduce has recently emerged as a powerful tool for distributed and scalable processing of voluminous data [11]. The classical MapReduce framework has a master node as the centralized controller and a number of slave nodes as worker nodes. Two major steps, namely, *Map* phase and *Reduce* phase, comprise the standard MapReduce work flow. In the *Map* phase, a master node takes the input dataset, divides it into smaller sub-problems, and distributes them to worker nodes. The worker nodes process the smaller problem, and pass the intermediate answer, i.e., a number of (key, value) pairs back to its master node. The master node uses a partition function to assign these (key, value) pairs. The design of the partition function is bounded by a constraint that the pairs with the same key should go to the same machine. The worker nodes in the *Reduce* phase will pull these intermediate data from the mapper nodes through a datacenter network (referred to as the *shuffle* subphase), and combine them in some way to form the output for the original problem. This divide-and-conquer process enables MapReduce to work on huge datasets with distributed server clusters. Yet a user of MapReduce only needs to write the map and the reduce functions, which effectively hides the operation details of large server clusters, offering a highly flexible, scalable, and fault tolerant solution for general big data processing applications.

The generic and simple interface also implies that MapReduce can be a bottleneck in the overall processing with specific applications or specific data. Significant efforts have been demonstrated on relieving blocking operations [31], improving energy efficiency [27, 30], enhancing scheduling [34, 44], or relaxing the single fixed dataflow [5, 10, 33, 43]. There have also been recent works on efficient scheduling of massive MapReduce jobs running in parallel [6, 7]. Most of these studies have assumed that the input data are of uniform distribution, which, often hashing to reduce worker nodes, naturally leads to a desirable balanced load in the later stages.

The real world data, however, are not necessarily uniform and often exhibit remarkable skew. For example, in PageRank, the graph commonly includes nodes with much higher degrees of incoming edges than others [41], and in Inverted Index, certain content can appear in many more documents than others [25]. Such skewed distribution of the input or intermediate data,

i.e., (key, value) pairs generated by the map tasks, can make a small number of mappers or reducers take a significantly longer time to complete than others [32]. Recent experimental studies [41, 32] have shown that, in the CloudBurst application with a biology dataset of a bimodal distribution, the slowest map task takes five times longer to complete than the fastest; PageRank with the Cloud9 data is even worse, where the slowest map task takes twice longer to complete as the second slowest, and the latter remains five times slower than the average. Our experiments with the WordCount application also show a similar phenomenon. Given that the overall finishing time is bounded by the slowest task, it can be dramatically prolonged with such skewed data.

In distributed databases, data skew is a known common phenomenon and there have been such mature solutions as joining, grouping, aggregation, etc. [39], [42]. Unfortunately, they can hardly be applied in the MapReduce context. The map function transfers the input raw data into (key, value) pairs and the reduce function merges all intermediate values associated with the same intermediate key. In the database case, the pairs sharing the same key are not necessarily processed in a single machine; MapReduce, on the other hand, must guarantee that these pairs belong to the same partition, in other words, be distributed into the same reducer.

There have been pioneering works dealing with the data skew in MapReduce [17], [25], [36]. Most of them are offline heuristics, either waiting for all the mappers to finish so as to obtain the key frequencies or sampling before the map tasks to estimate the data distribution and then partition in advance, or repartition the reduce task to balance the load among servers. These solutions can be time-consuming with excessive I/O cost or network overhead. They also lack theoretical bounds for the solutions given that most of them are heuristics.

In this thesis, we for the first time examine the problem of accommodating data skew in MapReduce with online operations. Different from earlier solutions in the very late reduce stage [25] or after seeing all the data [17], we address the skew from the very beginning of data input, and we make no assumption to the *a priori* knowledge of the data distribution nor require synchronized operations. We examine the keys in a continuous fashion and adaptively assign the tasks with a load-balanced strategy. We show that the optimal strategy is a constrained version of the *online minimum makespan problem* [12], and we demonstrate that, in the MapReduce context where tasks with identical keys must be scheduled to the same machine, there is an online algorithm with provable 2-competitive ratio. We further suggest that the online solution can be enhanced by a sample-based algorithm, which identifies the most frequent keys and assign associated tasks in advance. We show that, probabilistically, it achieves a 3/2-competitive ratio with a bounded error.

We evaluate our algorithm on both synthetic data and real public datasets. Our simulation results show that, in practice, the maximum load of our online and sample-based algorithms are close to the offline solutions, and are significantly lower than that with the naive hash function in MapReduce. They enjoy comparable computation times as the hash function, which are much shorter than that of the offline solutions.

However, consider that the massive data movement through the network in the shuffle sub-phase can occur for load balancing when the application scale is large, the network inside the

datacenter can have a great impact on the overall performance of MapReduce. Earlier studies for load balancing [25, 36, 17, 28] have generally assumed that the network inside the datacenter has enormous bandwidth and thus incurs negligible delay for such movement. Recent studies suggest that the network inside the datacenter can be a potential bottleneck, too [16]. It has been shown that, in a real datacenter network, there is a great prevalence of highly utilized links (utility of 70% or higher), which are located throughout the network [22, 4]. As such, the point-to-point movement of intermediate data between the map tasks and reduce tasks in the shuffle subphase affects the overall performance of MapReduce [40]. Our measurement reveals that excessive network traffic for the sheer amount data movement can indeed contradict the benefit of load balancing if the network bottlenecks are not carefully avoided. Given the prevalence and the dynamics of the background traffic, the workloads of the machines, the data moving costs and the bottlenecks within datacenter networks must be jointly considered when minimizing the overall finishing time for MapReduce.

In this thesis, we for the first time examine this problem of datacenter-network-aware load balancing in the shuffle subphase in MapReduce. Different from earlier studies on load balancing that assume the network inside a datacenter is of negligible delay and infinite bandwidth, we consider the traffic and bottlenecks in real datacenter networks by introducing the constraints on available network bandwidth, and demonstrate that the corresponding problem can be decomposed into two subproblems for network flow and load balancing, respectively. We show effective solutions to both of them, which together yield a complete solution toward near optimal datacenter-network-aware load balancing. A much simpler yet performance-wise comparable greedy algorithm is also developed for fast implementation in practice.

We evaluate our algorithms both on synthetic and real public datasets. We compare our algorithm with the state-of-the-art load balancing algorithm LPT [17], locality-aware partition algorithm LEEN [20] and the MapReduce default solution. The results show that, in practice, our greedy algorithm can significantly reduce the shuffle finishing time for MapReduce. It also enjoys comparable maximum load as the LPT, which is much smaller than LEEN and the MapReduce default solution.

1.1 Background and Motivation

In this section, we present an overview of MapReduce and datacenter network. We will also discuss the skew issues and the data shuffling cost therein that motivates our study.

The MapReduce libraries have been written in different programming languages, taking Apache Hadoop (High-availability distributed object-oriented platform), one of the most popular free implementations, as an example. It runs on a cluster of machines (the *nodes*) with a *master-worker* architecture, where a master node makes scheduling decision and multiple worker nodes run tasks dispatched from the master. In the *Map* Phase, the master node divides the input, usually a large dataset, into small data blocks and distributes them to map workers. Each data block has three copies stored in different machines. The map workers will process the data blocks, generate a large amount of intermediate (key, value) pairs and report the locations of these pairs

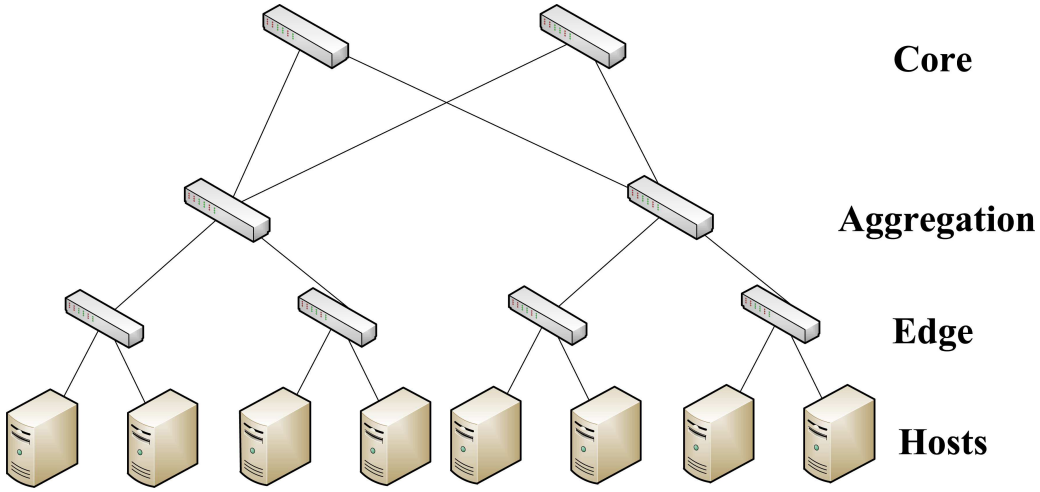


Figure 1.1: Typical 3-layer datacenter network topology.

on the local disk to the master, which is responsible for forwarding these locations to the reduce workers. The master node then uses a partition function to assign the intermediate data to different worker nodes to process in the *Reduce* Phase. In the *Reduce* Phase, the reduce workers will pull these intermediate data from mapper nodes in the server clusters to collect intermediate files, i.e., for m mappers and r reducers, up to mr distinct copy operations through the datacenter network are needed. A user defined reduce function will apply to these intermediate files and output the final results.

Existing datacenters generally adopt a multi-root tree topology to interconnect server nodes [22, 4]. In Figure 1.1, we present a generic datacenter network topology. The 3 layers of the datacenter are the *edge* layer, which contains the Top-of-Rack switches that connect the servers to the datacenter's network; the *aggregation* layer, which is used to interconnect the ToR switches in the edge layer; and the *core* layer, which consists of devices that connect the datacenter to the WAN. In general, for a datacenter, only part of its servers will serve as MapReduce workers. In MapReduce, the workers are located on the leaf nodes of the tree, and the intermediate data will be moved from one leaf node to another node during the shuffle subphase.

1.1.1 Data Skew

In Hadoop, the default partition function is a hash function and simply *Hash(HashCode(intermediate key) mod ReducerNumber)*, which is highly efficient and naturally achieves load balance if the keys are uniformly distributed. This however can fail with skewed inputs. For example, in WordCount, a classical MapReduce application, popular words like “the", “a" and “of" appear much more frequently, which, after hashing, imposes heavier workload to the corresponding reduce workers. Consider a toy example shown in Figure 1.2 with skewed input. The naive hash function will assign key a, d and g to the first machine, key b and e to the second machine, and key c and f to the third machine. As a result, the first machine achieves a maximum load with 21, 7 times more than that of the least load while the maximum load of optimal solution would

a	b	b	c	c	d	a	a	a	e	f	f	d	d	g	g	a	a	a	a	a	a	a	a	g	g	g	g	f	f

Hash	a: 12; d: 3; g: 6	b: 2; e: 1	c: 2; f: 4
Optimal	a: 12	b: 2; d: 3; f: 4	c: 2; e: 1; g: 6

Figure 1.2: An illustrative example for key assignment in MapReduce. There are three machines in this example. The “Hash", “Optimal" rows represent the result load distribution of each scheduling, respectively.

be 12 as shown in the “optimal" row. Since the overall finishing time is bounded by the slowest, such a simple hash-based scheduling is simply not satisfactory.

It is worth noting that Hadoop starts to execute the *Reduce* phase before every corresponding partition is available, i.e., the *Reduce* phase is activated when only part of mappers has been completed (5% by default) [25]. The rationale behind this synchronous operation is to overlap the map and reduce and consequently reduce maximum finishing time; yet it can prevent from making a partition in advance. In fact, the *Reduce* phase further consists of three subphases: *shuffle*, in which the task pulls the map outputs; *sort*, in which the map outputs are sorted by keys; and *reduce*, in which a user-defined function takes the map outputs with each key, and, after all the mappers finish working, starts to run and generates the final outputs. In Figure 1.3, we show a detailed measurement of processing times of all the phases for a WordCount application running on Amazon EC2. The *Map* phase starts first at initial time and then the *Reduce* phase, actually the *shuffle* phase, starts at about 200s. We can see that the shuffle finishing time is much longer than that of the reduce, for the reducer workers should wait the map workers generate intermediate pairs while use remote procedure calls to read the buffered data from the local disks of the map workers. Also note that the maximum map finishing time is quite close to that of shuffle. Therefore, if we wait till all the keys are generated, in other words, start the *shuffle* phase after all the map workers finish, the overall job finishing time will be doubled. This is unfortunately what the state-of-the-art offline algorithms do for balancing the load with skewed input data. It motivates our design of an online solution to start the *shuffle* phase as soon as possible while making the maximum load of the reduce workers as low as possible.

1.1.2 Why Datacenter-Network-Aware?

However, most of these load balancing strategies assume that the interconnected network for servers within a datacenter is of ultra-high speed, that is, ideally, of infinite bandwidth and these works literally ignore the original location of the (key, value) pairs as well as the cost to move the data to their destinations, simply sort the keys by decreasing order of the frequency of keys, and then assign to the least loaded machine [17]. Consider an example shown in Figure 1.4, where key z may be assigned to node A because initially the load of all the nodes is 0, thus making $\{z\}, \{w\}, \{x, y\}$ be assigned to node A, B, C with load 8, 7, 4, respectively. The resultant loads

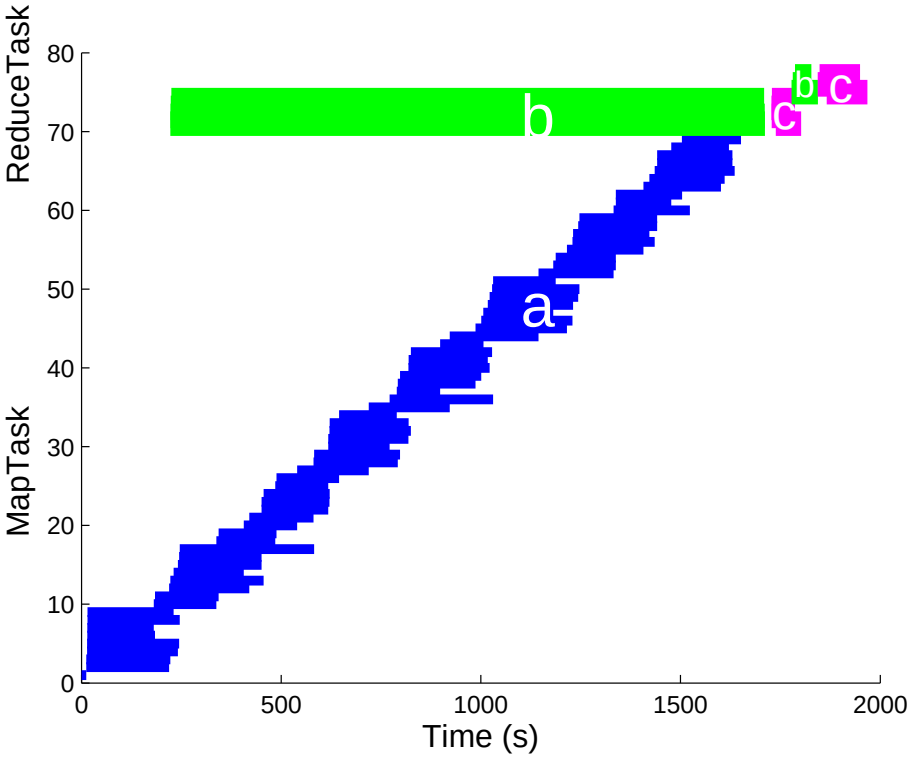


Figure 1.3: We ran WordCount application on Amazon EC2 with 4 instances and we set 70 map tasks and 7 reduce tasks. This figure describes the timing flow of each *Map* task and *Reduce* task. Region a represents the actual map function executing time. Region b represents shuffling time and region c represents the actual reduce function executing time. The regions between both b and c represent sorting time.

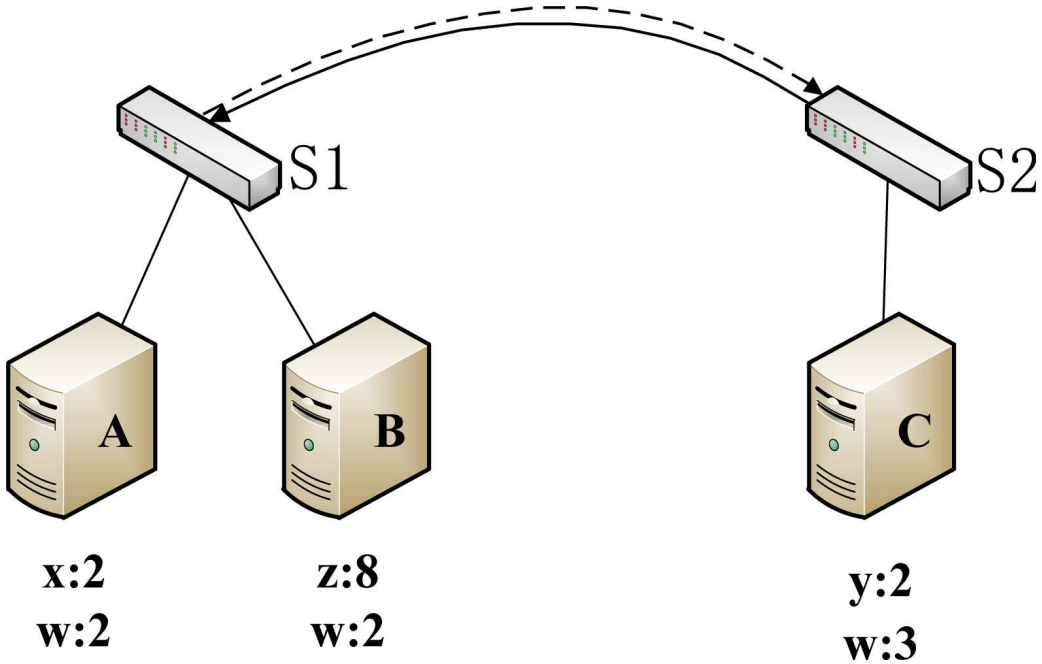


Figure 1.4: **x:2** under machine A indicates the key x with size 2 in the machine A. Note that the link from switch $S1$ to switch $S2$ is a bottleneck link.

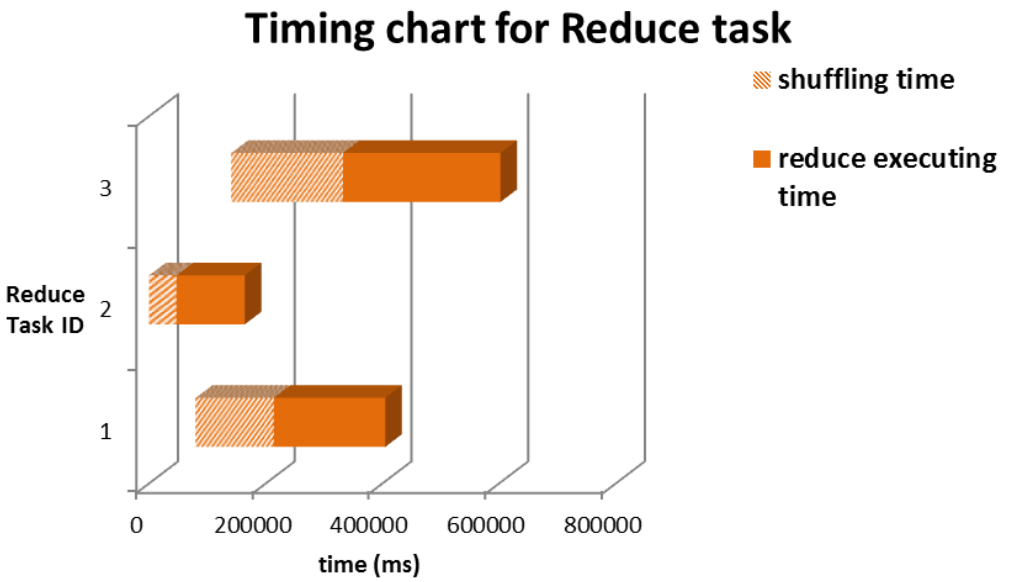


Figure 1.5: The timing flow of each *Reduce* task.

are well balanced; yet, almost all the data have incurred movement, causing significant network traffic.

To understand the impact of such traffic in real world, we have run a series of experiments on a cloud testbed that resembles the configuration of Amazon EC2. Figure 1.5 shows the detailed results of the processing time in the reduce phase for a *k-medoids* application, where we adopt 3 instances in 3 physical machines to have a clear view on the data movement. To accurately count the times, we have also enabled the *slow start running* mode, which means the reduce phase will start after all the mappers finish [20, 17]. In the figure, the left part of the processing time for each task gives the shuffling time and the processing time afterwards is roughly the computation time of the reduce function [1]. We can see that, even in this simplified setup, for each task, the latency for shuffling is comparable to that of the reduce function, although only the latter is responsible for computation.

There have been efforts toward minimizing the shuffled data for load balancing, so as to reduce the moving cost and consequently the associated latency [20, 19]. The latency however is not only affected by the amount of the data shuffled, but also by the traffic distribution within the datacenter network. It is known that, roughly 20% of the core links in a real datacenter network are 70% or higher utilized at least 50% of the time, and the link utilizations vary significantly over time [4]. For a datacenter network carrying over the traffic of 1500 monitored machines, 86% of the links observe congestion lasting at least 10 seconds, and 15% observe congestion lasting over 100 seconds [22]. As such, without avoiding potential network bottlenecks, simply minimizing the data to be shuffled does not necessarily achieve desirable performance. Consider Figure 1.4 again. To minimize the data shuffled, keys $\{x\}, \{z\}, \{y, w\}$ may be assigned to nodes

[1] It in fact includes both the sorting time and the computation time of the reduce function, while the former is small enough to be neglected.

A, B, C, respectively, if the strategy is blind to the locations of the network bottlenecks. This unfortunately hits the bottleneck link from switch $S1$ to switch $S2$. Instead, if we are aware of the bottleneck link by assigning key w to machine A, the overall finishing time can be reduced. In summary, the traffic and the bottlenecks of the underlying network play critical roles toward designing a practical load balancing strategy for MapReduce in real world datacenters.

Hence, in this thesis, we will first investigate the load balancing problem with skewed data input in MapReduce, then consider to reduce the high data movement cost that can occur for load balancing from the view of the datacenter network.

1.2 Thesis Organization

The organization of the rest of this thesis is as follows:

In chapter 2, We present an overview of related work on load balancing and minimizing data movement cost in MapReduce.

In chapter 3, we address the load balancing issue with skew input in MapReduce. We first present the problem formulation with the online model. We then illustrate our online algorithm and derives the 2-competitive ratio and further suggest a Sample-based semi-online algorithm, which, probabilistically, achieves a 3/2-competitive ratio with a bounded error.

In chapter 4, we analyze the problem with a generic network cost function and propose a 2-approximation algorithm. We then extend the model to consider the real network bandwidth and demonstrate that the corresponding problem can be decomposed into two subproblems for network flow and load balancing, respectively. We have shown effective solutions to both of them, which together yield a complete solution toward near optimal datacenter-network-aware load balancing. A much simpler and yet performance-wise comparable greedy algorithm was also developed for fast implementation in practice.

In chapter 5, we do the performance evaluation on both synthetic data and real public datasets with both set of algorithms: online load balancing algorithms and the network-aware algorithms. Our algorithms show significant improvements on load balancing and the shuffle finishing time.

In the last chapter, we discuss some points and ideas that need to improve or deserve continuing working on in the future work and conclude this thesis finally.

Chapter 2

Related Works

Recently the amount of data of various applications has increased beyond the processing capability of single machines. To cope with such data, scale out parallel processing is widely accepted. MapReduce [8], the de facto standard framework in parallel processing for big data applications, has become widely adopted. Nevertheless, MapReduce framework is also criticized for its inefficiency in performance and as “a major step backward" [9]. This is partially because that, performance-wise, the MapReduce framework has not been deeply studied and fine-tuned as compared to the conventional systems. As a consequence, there are many recent studies in improving MapReduce performance.

The MapReduce framework can be considered as a parallel structure. Though there are decades of studies in parallel processing scheduling [2, 23], whether these works can be directly applied to the MapReduce framework, which has a special map task - reduce task structure is not clear; especially, their theoretical bounds are unlikely to be directly transformed. As such, besides the system research in MapReduce [34, 44], there is also a flourish of works on understanding the theoretical performance and limitations of MapReduce systems. As an example, improved bounds on job scheduling are achieved in [7, 6].

There are recent studies on the *shuffle* phase which may introduce skewed loads towards the reduce tasks. The straggler problem is firstly described in [8], which proposed to use combiner that is used to reduce the output of the map phase, and backup task that is to alleviate the problem of straggler. It is shown that the straggler problem in MapReduce is caused by the Zipf distribution of the input or intermediate data [32]. In [41], it presents five types of skews that can arise in MapReduce applications and proposes five best practices to mitigate skew in MapReduce jobs, but does not specify the details of the practice.

Gufler [17] defined a new cost model that takes into account non-linear reducer tasks and provided an algorithm to estimate the cost in a distributed environment. According to the estimated cost, two load balancing approaches-fine partitioning and dynamic fragmentation are presented in this paper. However, there exists a reducer slow-start-synchronization barriers although the author argues that "for highly complex reduce algorithms, the time cost due to slow-start are negligible“. Ibrahim [21] also argues that it can fasten the map execution because the complete I/O disk resources will be reserved to the map tasks in the scenario where map tasks

and reduce tasks share the same nodes, which is the typical case in current MapReduce systems. For the challenge of gathering statistics from mappers, Gufler [18] presented TopCluster, a distributed monitoring system for capturing data skew in MapReduce systems. The TopCluster consists of two components: a monitoring component, which is executed on every mapper and an integration component. The monitoring component is responsible to capture the local data distribution and identify its most relevant subset for cost estimation. The integration component aggregates these subsets and approximates the global data distribution.

Another strategy to mitigate skew is to divide the workload into fine-grained partitions and re-assign these partitions to other nodes which have idle resources. Kwon [25] proposed and implemented SkewTune approach, which is an extension of Hadoop. The idea is to repartition the unprocessed input data of a task with the greatest expected remaining processing time to a idle node and fully utilize it. However, it will lead too much I/O cost and network overhead when transferring data, also it does not consider that some nodes are not suitable to be the destination of mitigators. For example, it is possible that the node that is reassigned will be judged as "straggler" again.

Progressive-sampling [36] takes a user-specified model of reducer performance, the load balancer uses a progressive objective-based cluster sampler to estimate the load associated with each reduce-key. It balances the workload by using Key Chopping to split keys with large loads into sub-keys that can be assigned to different distributive reducers, and Key Packing to assign keys with medium loads to reducers to minimize the maximum reducer load. Keys with small loads are hashed as they have little effect on the balance. This repeats until the user specified balancing objective and confidence level are achieved.

From a theoretical point of view, we present a study from an online balancing of the shuffle output point of view, and achieve a 2-competitive ratio. Our work is related to the online minimum makespan problem. In a classical minimum online makespan scheduling problem, jobs will come one by one with processing time and they need to be assigned to m identical parallel machines. Each job will be assigned irrevocably to a machine before the next job can be revealed. No preemption is allowed and the goal is to minimize the maximum finishing time. For this classical online problem, the well-known Greedy-Balance load balancing approach [24] has $(2-1/m)$-competitive ratio. However, we differ from online minimum makespan problem in that the same key should go to the same machine in MapReduce context. The offline minimum makespan problem is also NP-Complete. The best result for the offline version is a $(4/3-1/(3m))$ approximation [15]. Some other results on various versions can be found in [13, 37, 12].

There is a flourish of works on minimizing the high data moving cost during the shuffle subphase. These works mostly focus on how to put the computation nodes and the storage nodes as close as possible (e.g. Purlieus [35]), or move data as small as possible, e.g. improving data locality in reduce task [40], Leen [21] and LARTS [19]. These studies have made assumptions on the network traffic and performance such as the data movement is mostly affected by the hop count, there is no background network traffic, or there is abundant bandwidth in a datacenter network that inter-connects servers. However, the data transmission time in the real world is bounded by the bottleneck link, which has the maximum finishing time to transmit the data, and

could be significantly affected by the task placement, i.e, an inappropriate reduce task placement may lead to significant traffic on certain links and cause network bottlenecks.

Chapter 3

Online Load Balancing with Skewed Input

3.1 Problem Definition

We consider a general scenario where, during the *Map* phase, the mapper nodes generate many intermediate values (data) with associated keys. For each of these, they form a (key, location) pair, which we will typically denote (k_i, l_i), where location refers to where the (key, value) pairs is stored, and report the pair to the master node [1]. For the rest of this thesis, we will only be interested in processing the *key* attribute of such a pair, and sometimes, for simplicity, skip the location attribute altogether. The master node then assigns these pairs to different machines based on the key values. Each such pair must be assigned to one machine for processing, with the additional restriction that pairs with the same key must be assigned to the same machine. The number of pairs assigned to a machine makes up the *load* of the machine. Here, we assume each machine will have a finishing time directly proportional to its load, and the finishing time of the machine with the highest load (called the *makespan*) will be the overall finishing time. The objective then is to minimize the overall finishing time by minimizing the maximum load of all the machines [2].

From the perspective of the master node, the input is a stream $S = (b_1, b_2, \cdots, b_N)$ of length N, where each b_i denotes a (key, location) pair. Let N' denote the number of different keys; we denote $C = \{c_1, c_2, \cdots, c_{N'}\}$ as the universal set of the different keys, with $b_i \in C$ for every $i \in N$.

We assume that there are m identical machines numbered $1, \ldots, m$. We denote the load of machine i by M_i, i.e., the number of pairs assigned to machine i. Initially, all loads are 0. Our

[1] Note that the value is *not* being reported, thus, the information received by the master node for each item will require a small amount of space.

[2] Here we make an implicit assumption that each pair represents a workload of unit size, but our algorithm can easily work also for variable, integer workload weights.

goal is to assign each b_i in S to a machine, in order to obtain

$$min \underset{i \in \{1,2,\cdots,m\}}{Max} M_i$$

such that any two pairs (k_1, l_1) and (k_2, l_2) will be assigned to the same machine if $k_1 = k_2$.

We consider two input models, leading to two computational models. In our first model, we allow arbitrary (possibly adversarial) input, and stipulate that the master node will assign the pairs in a purely online fashion. In our second model, we assume that the input comes from a probability distribution, and, in order to exploit this fact, the master node is allowed to store and process a *sample* of its input before starting to assign the sample and the rest of the pairs in an online fashion.

3.2 A 2-Competitive Fully Online Algorithm

In order to minimize the overall finishing time, it makes sense to start the *shuffle* phase as soon as possible, with as much an overlap with the *Map* phase as possible. In this section, we give an *online* algorithm called List-based Online Scheduling for assigning the keys to the machines. That is, our algorithm decides, upon receiving a (key, location) pair, to which machine to assign that item without any knowledge of what other items may be received in the future. We assume that the stream of items can be arbitrary, that is, after our algorithm makes a particular assignment, it can possibly receive the "worst" stream of items for that assignment choice in terms of its optimization goals. Our algorithm will be analyzed for the worst case scenario: we will compare its effectiveness to that of the best offline algorithm, i.e., one that makes its decisions with knowledge of the entire input.

For assigning items to machines based on their keys, we adopt a Greedy-Balance load balancing approach [24] of assigning unassigned keys to the machine with the smallest load once they come in.

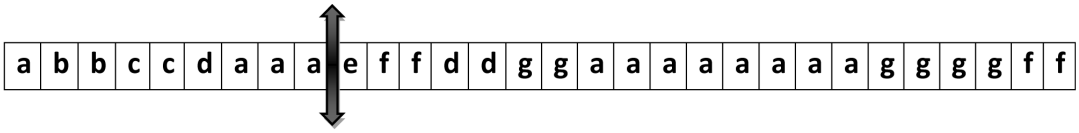
Algorithm 1 List-based Online Scheduling

```
Read pair (k_i, l_i) from S
if k_i has been assigned to machine j then
   Assign the pair to the machine j
else
   Assign the pair to the machine with the least load
end if
```

We now show that our algorithm yields a overall finishing time which at most twice that of the best offline algorithm for this problem. The proof follows the lines of proof for the Greedy-Balance load balancing algorithm.

Theorem 1. *List-based Online Scheduling has a competitive ratio of 2.*

Proof. Let OPT denote the offline optimum makespan, the maximum finishing time. Assume machine j is the machine with the longest finishing time in the optimal offline solution and



Online	a: 12; d: 3	b: 2; e: 1; g: 6	c: 2; f: 4
Sample-based	a: 12	b: 2; d: 3; f: 4	c: 2; e: 1; g: 6

Figure 3.1: An illustrative example showing benefits of sampling. The setting is the same as in Figure 1.2. The “Online" and “Sample-based" rows represent the respective result load of each machine in the online and sample-based schedules.

T' is the number of pairs read just before the last new key, say c_j, is assigned to machine j. Obviously, T' must be less than N, the total length of the input. Then, at that time c_j is assigned to machine j, j must have had the smallest load, say L_j. Thus we have:

$$L_j \leq \frac{T'}{m} < \frac{N}{m} \leq OPT$$

Let $|c_j|$ denote the number of pairs with key c_j in S. Then, the finishing time of machine j, denotedy by T, which is also the makespan, is

$$T = L_j + |c_j| \leq OPT + OPT = 2OPT$$

Thus, with our List-based Online algorithm, the makespan can achieve a 2-competitive ratio to the offline optimal makespan. □

3.3 A Sampling-based Semi-Online Algorithm

Our previous algorithm made no assumptions about our advance knowledge of the key frequencies, which could be adversarially chosen. Clearly, if we had some a priori knowledge about these frequencies, we could make the key assignments more efficiently.

In this section, we assume that the pairs are such that their keys are drawn randomly and independently from an unknown distribution. In order to exploit this, we compromise on the online nature of our algorithm and start by collecting a small number of input pairs into a *sample* before making any assignments. We then use this sample to estimate the frequencies of the K most frequent keys in this distribution, and use this information later to process our stream in an online fashion. In order to observe the advantages of such a scheme, consider another toy example shown in Figure 3.1. If we can wait for a short period before making any assignments, for instance, collect the first 9 keys in the example and assign the frequent keys to the machine with the least load in order of frequency, the maximum load is reduced to 12 from 15.

Our algorithm classifies keys into two distinct groups: the K most frequent, called *heavy* keys, and the remaining, less frequent keys. The intuition is that the heavy keys contribute

much more strongly to the finishing time than the other keys, and thus, need to be handled more carefully. As a result, our algorithm performs assignments of the keys to the machines differently for the two groups.

We first consider how to identify the heavy keys. Clearly, if one could collect all of the stream S, one could solve the problem exactly and easily. However, this would use up too much space and delay the assignment process. Instead, we would like to trade off the size of our sample (and the wait before we start making the assignments) with the accuracy of our estimate of the key frequencies. We explore the parameters of this trade-off in the following part of this section .

Our first goal is to show that we can identify the most frequent (heavy) K keys reliably. We will first analyze the sample size necessary for this task using techniques from probability and sampling theory. We will then move on to an algorithm for assigning the heavy keys as well as the remaining, less frequent ones.

3.3.1 Sample Size

In this section we analyze what our sample size needs to be in order to obtain a reliable estimate of the key frequencies. Estimating probabilities from a given sample is well understood in probability and statistics; our proof below follows standard lines.

Let S' denote our sample of size n and n' denote the number of distinct keys in S'. For simplicity, we will ignore the fact that S' consists of (key, location) pairs, instead considering it as a stream (or set) of keys.

Let p_i denote the proportion of key c_i in the stream S, and let X_i denote the number of occurrences of c_i in the sample S'. (It is possible to treat p_i as a probability as well without any changes to our algorithm.) Then X_i can be regarded as a binomial random variable with $E(X_i) = np_i$ and $\sigma_{X_i} = \sqrt{p_i(1-p_i)n}$. Provided that n is large (i.e., $X_i \geq 5$ and $n - X_i \geq 5$), the Central Limit Theorem (CLT) implies that X_i has approximately normal distribution regardless of the nature of the item distribution.

In order to select the heavy keys, we need an estimate of key probabilities. To this end, we estimate p_i as $\hat{p}_i = X_i/n$, which is the sample fraction of key i in S'. Since $\hat{p}_i$ is just X_i multiplied by the constant $1/n$, $\hat{p}_i$ also has approximately a normal distribution. Thus, $E(\hat{p}_i) = p_i$ and $\sigma_{\hat{p}_i} = \sqrt{p_i(1-p_i)/n}$, and according to the Central Limit Theorem (CLT), we have the following corollary bounding the size of the sample that we need in order to have a good estimate of the key frequencies.

Corollary 2. *Given a sample S' of size of $n = (z_{\alpha/2}/\epsilon)^2$, consider any key c_i with proportion p_i, satisfying $X_i \geq 5$ and $n - X_i \geq 5$, and let $\hat{p}_i = X_i/n$. Then, $|p_i - \hat{p}_i| \leq \epsilon\sqrt{(\hat{p}_i)(1-\hat{p}_i)}$ with probability $1 - \alpha$.*

Note that $z_{\alpha/2}$ is a parameter of the normal distribution whose numeric value depends on α and can be obtained from the normal distribution table

3.3.2 The Heavy Key

We first state our notion of the more frequent, called heavy, keys. The following guarantees that we will explore all keys with length at least $OPT/2$.

Definition 1. *(Heavy key) A key i is said to be heavy if $\hat{p}_i \geq 1/2m + \epsilon$.*

Note then that a key whose length (i.e., the number of times that it occurs in S) greater than $N/2m$ is then very likely to be heavy. Then, it is easy to see that there could be up to $2m$ heavy keys.

It is worth noting that one might need to see $O(m)$ samples to sample a particular heavy key. Thus we will need to increase our sample size by an $O(m \log m)$ factor to make sure that we sample the heavy keys and estimate the lengths of each of the heavy keys reliably, resulting in a sample size of $n = O((z_{\alpha/2}/\epsilon)^2 m \log m)$.

Intuitively, the assignments of the heavy keys will have a large impact on the finishing time, thus, we treat them with extra care.

3.3.3 A Sample-based Algorithm

We are now ready to present an algorithm for assigning the heavy keys, similar to the sorted-balance algorithm [24] for load balancing. Our algorithm, Sample-based algorithm, first collects it sample, then sorts the keys in the sample in non-increasing order of observed key frequencies, and selects the K most frequent keys. Then, going through this list, it assigns each type of key to the machine with the least current load. For assigning all other keys, we use algorithm 1.

Algorithm 2 Sample-based Algorithm

```
Wait until n = O((z_{α/2}/ε)^2 m log m) pairs are collected to form sample
Sort the K most frequent keys in the sample in non-increasing order, say p̂_1 ≥ p̂_2 ≥ ··· ≥ p̂_K

Going over the sorted list, assign each key i to the machine with the smallest load
while a new pair is received with key i do
  if i was previously assigned to machine j then
    Assign i to machine j
  else
    Assign i to the machine with the smallest load
  end if
end while
```

The following lemma bounds the size of the last key assigned to the machine which ends up with the longest finishing time.

Lemma 1. *If the makespan obtained by Sample-based algorithm is larger than $OPT + \epsilon N$, with probability at least $1-2\alpha$, the last key added to the machine has frequency at most $(OPT/2N+\epsilon)$.*

Proof. Let OPT be the optimal makespan of the given instance. Divide the keys into two groups: $C_L = \{j \in C : \hat{p}_j N > OPT/2 + \epsilon N\}$ and $C_S = C - C_L$, called large and small keys respectively.

With probability $1-\alpha$, we have that $p_j N > \hat{p}_j N - \epsilon N > OPT/2$ for all keys j. Note that, then there can be at most m large keys, otherwise one could not obtain a finishing time of OPT with two such keys scheduled on the same machine. Since the length of a large key is greater than $OPT/2$, this contradicts that OPT is the optimal makespan. It is also obvious that we cannot have any keys with length greater than OPT, i.e., no j exists such that $p_j N > OPT$. Thus, if the makespan obtained by the algorithm is greater than $OPT + \epsilon N$, with probability $1-\alpha$ the last new key that is assigned to the makespan machine must be a small key. Using the union bound, with probability $1-2\alpha$, the last type of key assigned to the machine with the longest processing time must have frequency at most $OPT/2N + \epsilon$. □

Theorem 3. *With probability at least* $1-2\alpha, 0 < \alpha < 1$*, Sample-based algorithm obtains a overall finishing time which is at most* $3/2OPT + N\epsilon$*.*

Proof. Assume machine j has the longest finishing time when Sample-based algorithm is used for the key assignments. Consider the last key k assigned to j. Before this assignment, the load of j is L_j, and it must be the least load at that point in time among all the machines. Thus,

$$L_j \leq \frac{N}{m} \leq OPT$$

Then, after adding the last key k, its finishing time becomes at most

$$L_j + Np_k \leq OPT + OPT/2 + \epsilon N \leq 3/2OPT + \epsilon N$$

Note that $L_j \leq OPT$ is deterministically true. Therefore, the probability of the above can be shown to be at least $1-2\alpha, 0 < \alpha < 1$. Thus, With at least $1-2\alpha, 0 < \alpha < 1$, our Sample-based algorithm can achieve the $3/2OPT + \epsilon N$. □

Chapter 4

On Datacenter-Network-Aware Load Balancing

4.1 Data Placement with Generic Network Cost

Define a *cluster* as a set of the (key, value) pairs that share the same key. The (key, value) pairs of a cluster may be generated by different mappers, and they are often scattered on different nodes. In the *Reduce* phase, each reducer should deal with several tasks, each of which will compute multiple clusters. Before *Reduce* starts, we should first partition the keys into several subsets with the consideration of load balancing among different nodes and move the partitions to the corresponding nodes through the network. Therefore, the overall finishing time of a whole job not only depends on the computation, but also on the network performance for moving partitions to their corresponding reducers.

To analyze and optimize the overall delay, we start from a generic network cost function $f_m(x, y)$, for moving a unit amount of data from node x to y. Note that $f_m(x, y)$ is equal to 0 if x and y are the same node. We denote $C = \{1, 2, \cdots, i, \cdots, K\}$ as the universal set of the different keys, where $K = |C|$ is the number of different clusters. For the network topology, we have $G = (V, E)$, where some of the nodes are the hosts that can run mappers and reducers, denoted by $M = \{1, 2, \cdots, m\}$, with $m = |M|$ as the number of different machines. We use function $f_c(s)$ to denote the computation time to process the data of size s.

We assume $X_{k,u}$ is the size of cluster k generated by the mapper on node u, where $k \in C$, $u \in M$. Then $\sum_{u=1}^{m} X_{k,u}$ gives the size of cluster k over all the machines. We also define

$$Y_{k,j} = \begin{cases} 1, & \text{if the cluster } k \text{ is assigned to node } j; \\ 0, & \text{otherwise.} \end{cases}$$

Thus, the size of data on node j is

$$\sum_{k=1}^{K} Y_{k,j} \sum_{u=1}^{m} X_{k,u}$$

and the corresponding time to process these data is

$$S_j = \sum_{k=1}^{K} Y_{k,j} f_c(\sum_{u=1}^{m} X_{k,u}) \ .$$

Also, the time to transfer these data is

$$C_j = \sum_{k=1}^{K} Y_{k,j} \sum_{u=1}^{m} X_{k,u} f_m(u,j) \ .$$

Let t indicate the maximum finishing time over all the machines. Thus,

$$t = \max_{j \in M} \ (S_j + C_j) \ .$$

and our objective is to minimize t.

This problem is equivalent to the following integer linear programming:
Minimize: t
Subject to:

$$\sum_{j \in M} Y_{k,j} = 1, k \in C \tag{4.1}$$

$$\sum_{k \in C} Y_{k,j} \Big(\sum_{u=1}^{m} X_{k,u} f_m(u,j) + f_c(\sum_{u=1}^{m} X_{k,u}) \Big) \leq t, j \in M \tag{4.2}$$

$$Y_{k,j} \in \{0,1\}, k \in C, j \in M \tag{4.3}$$

Our placement scheduling problem resembles the NP-complete scheduling problem on unrelated machine [29], which is to find an assignment of the jobs J so as to minimize the maximum finishing time, given a set of jobs J, a set of machines M and the processing time for job $k \in J$ on machine $j \in M$ is $p_{k,j} \in Z^+$. Even under the simple assumption that the delay per unit data size, $f_m(x,y)$, is equal for all node pairs, our problem is still NP-complete. Yet we can let

$$p_{k,j} = \sum_{u=1}^{m} X_{k,u} f_m(u,j) + f_c(\sum_{u=1}^{m} X_{k,u}).$$

Since the moving cost and the computation cost can become constant once we determine destination node, the inequality 4.2 can be converted to

$$\sum_{k \in C} Y_{k,j} p_{k,j} \leq t, j \in M \tag{4.4}$$

Then, we relax the integer constraint, and revise Eq. 4.3 to

$$Y_{k,j} \geq 0, k \in C, j \in M \tag{4.5}$$

Under this linear relaxation, we have the following polynomial-time algorithm with a 2-approximation to the optimal (Algorithm 3).

Algorithm 3 Network-Cost-Aware Placement Scheduling

1: Assign the key to the least load machine, having a maximum cost α
2: Apply a binary search on the interval $[\alpha/m, \alpha]$, find the smallest value T that is satisfying the constraints in the LP
3: Find an extreme point solution, say Y, to LP
4: For each $Y_{k,j}, k \in C, j \in M$, if $Y_{k,j}$ is 1, assign cluster k to machines j.
5: Construct a bipartite graph $H = \{C \cup M, E\}$, where$(k, j) \in E$ iff $Y_{k,j} \neq 0$ and find a perfect matching on it
6: Assign the clusters with fractional values in Y according to the matching in H

This algorithm is similar to the algorithm of the minimum makespan problem on unrelated parallel machines and our proof is motivated by that in [29]. We now show that this algorithm can yield an overall finishing time which at most twice that of the optimal algorithm.

Theorem 4. *Network-cost-aware Placement Scheduling algorithm has a 2-approximation to the optimal solution.*

Proof. The proof is similar to that of the scheduling algorithm on unrelated parallel machines [29], and thus, we only give the key steps. Let $LP(t)$ denote a feasible solution with maximum finishing time t and T^* is the optimal value for the integer linear programming. First, with the binary search, we can obtain the smallest value T satisfying the constraints in LP and note that the $T \leq T^*$ since $LP(T^*)$ is also a feasible solution. Second, we can prove that any extreme point solution (step 3 in Algorithm 3) to $LP(T)$ has at most $n + m$ non-zero variables for $X_{i,j}$ and at least $n - m$ jobs integrally, and thus, the number of the jobs fractionally is less than m. Third, we need to prove that there exists a perfect matching in the bipartite graph H, and thus, there is at most one job assigned to the machines at step 6 in Algorithm 3. Hence, the total finishing time is at most $2 \cdot T \leq 2 \cdot T^*$ as claimed.

□

4.2 Datacenter-Network-Aware Load Balancing

So far we have assumed a general network cost in the placement. While sharing certain similarities with the Internet, datacenter networks exhibit many unique characteristics that affect the instantiation of the cost. In particular, a network inside the datacenter network is often of a regular topology (say a fat-tree) and the hop count between a pair of nodes is generally short. The network-related cost therefore mainly comes from the bandwidth constraint of the bottleneck along a path. Although such bottleneck bandwidth can be much higher than that in the widearea Internet, as we have shown earlier, it can create non-negligible delays when moving big data for MapReduce operations [26]. Given there are multiple paths between node pairs, our focus is then to identify the best path for moving data partitions while still keeping the load on each node as balanced as possible.

Here, we assume that at the beginning of the *shuffle* subphase, all the intermediate data have been ready and that the bandwidth from a node to itself is unlimited. Similar to Section 4.1, we use $X_{k,u}$ to denote the size of cluster k generated by the mapper on node u and $Y_{k,j} = 1$ if cluster k is assigned to node j for $k \in C$, $u, j \in M$. The network topology is now defined as $G = (V, E)$, with any $e \in E$, $r_e \in R$, which indicates the link rate of e, i.e., the bandwidth. Some nodes of G are the hosts that can run reduce tasks, denoted by $M = \{1, 2, \cdots, m\}$.

Suppose paths $p_1, p_2, \cdots, p_w$ with amount of data $b_1, b_2, \cdots, b_w$ to be transmitted, respectively, share a link e with a bandwidth r_e. The time for the data to go through this link is thus $\sum_{i=1}^{w} b_i/r_e$. The total completion time for the *shuffle* subphase, then, is $max \sum_{i=1}^{w} b_i/r_e$ for all link e in the topology. Our goal is to find an assignment to place the clusters over all possible path selections and possible cluster placements so as to minimize the overall finishing time.

Obviously, the size of data on node j is

$$\sum_{k} Y_{k,j} \sum_{u=1}^{m} X_{k,u} \ .$$

Before analyzing the finishing time during the *shuffle* subphase, we introduce a variable p_{isd}, which is a set of flows $\{f_{isd}(u, v) \in R \mid u, v \in V, s, d \in M, i \in C\}$, indicating that $f_{isd}(u, v)$ size of the cluster i from s to d will put on the link (u, v). Thus, the amount of data to go through link (u, v) is

$$\sum_{s=1}^{m} \sum_{d=1}^{m} \sum_{i=1}^{K} f_{isd}(u, v),$$

$$\forall f_{isd}(u, v) \in p_{isd}, \forall i \in K, \forall s, d \in M$$

In this context, constraint (4.2) can then be replaced by

$$\max_{j \in M} \Big\{ \max_{(u,v)} \{ \frac{\sum_{s=1}^{m} \sum_{d=1}^{m} \sum_{i=1}^{K} f_{isd}(u, v)}{r_{(u,v)}} \} + f_c(\sum_{k} Y_{k,j} \sum_{u=1}^{m} X_{k,u}) \Big\} \tag{4.6}$$

where the (u, v) for the inner max should be chosen as

$$\forall (u, v) \in \Big\{ (u, v) | f_{isj}(u, v) \in p_{isj}, s \in M, u, v \in V, i \in C \Big\} \ .$$

$r_{(u,v)}$ is the rate, i.e., the bandwidth, of edge (u, v). We know that the overall finishing time is bounded by the maximum finishing time, which consists of the shuffle finishing time and the reduce function computation time in our context. For a specific machine j, the shuffle finishing time is determined by the time that the last packet goes through a certain link that is on the paths to the machine j. This link must have the maximum transmission time for the node j, which is the first part of the above formula. The second part is the computation time of the reduce function for the workload on node j. We apply f_c, a function of the size of data, to

get the computation time of the reduce function. This is reasonable because we can use the complexity of the reduce function to estimate the computation time of the reduce function for a certain cluster. The input of the f_c is the total size of data on a certain machine.

Our problem now is even more complicated than the one discussed in Section 4.1, we divide this problem into the network flow part and the load balancing part, and propose our solutions accordingly.

4.2.1 Optimizing Network Flow

For the networking part, given the destination of each cluster, our input of the networking part, T, is a list of tuples $T_i = (s_i, b_i, d_i)$, which means that b_i size of data at machine s_i will be delivered to machine d_i. Note that the size of $|T|$ is at most $m^2 - m$, since it is not necessary to transmit data when s_i is equal to d_i.

Let $f_i(u, v)$ denote the size of tuple T_i going through edge (u, v). The finishing time of data transmission is decided by the slowest edge. Thus we want to minimize

$$\max_{(u,v)\in V} \sum_{i=1}^{|T|} f_i(u, v)/r_{(u,v)} \ .$$

To be more clear of this problem, suppose we are given a time t to ship all the data to their destinations, we reconstruct the graph with the same nodes and edges with capacity $r_{(u,v)} * t$, for each edge (u, v). We are trying to find a feasible solution, such that all the data can reach their destinations, with the following constraints

$$\sum_{i=1}^{|T|} f_i(u, v) \leq r_{u,v} * t, \forall (u, v) \in E \ ;$$

$$\sum_{w\in V} f_i(u, w) = 0, \text{when } u \neq s_i, d_i \ ;$$

$$f_i(u, v) = -f_i(v, u), \forall u, v \ ;$$

$$\sum_{u\in V} f_i(s_i, u) = \sum_{u\in V} f_i(u, d_i) = b_i \ .$$

This problem produces an integer flow to satisfy all the demands. However, even its decision version is a NP-complete problem.

Fortunately, we can find the maximum ratio λ such that at least λ ratio of each tuple can be shipped without exceeding the capacity constraints, which is the maximum concurrent flow problem [38]. Here we need to maximize

$$\lambda = \min_{i\in C} \sum_{w\in V} f_i(s_i, w)/b_i \ .$$

This is equivalent to the problem of determining the minimum ratio by which the capacity must be increased to make all the data be shipped to their destinations. One solution to this problem is to use linear programming, which can get the λ for a given time t.

Here, λ is always less than or equal to 1. If $\lambda = 1$, we can decrease the guess value of t; otherwise, we can increase the guess value of t. Note that the solution is a feasible solution when $\lambda = 1$, although t may be too large. The network flow algorithm is shown as in Algorithm 4. In

Algorithm 4 Network-flow(T, σ)

Input: a list of $T_i = (s_i, b_i, d_i)$ tuples

1: $maxS = \sum_{i=1}^{|T|} b_i$
2: $maxT = maxS/min\ r_e$
3: $previousT = maxT$
4: $t = maxT/2$
5: $low = 0, high = maxT$
6: $previous\lambda = \lambda = 0$
7: **while** 1 **do**
8: $previous\lambda = \lambda$
9: Construct a graph with the same nodes and the same edge while the capacity of each edge e is $r_e * t$
10: Use linear programming to get the minimum percentage, λ, of each tuple that can be shipped without exceeding the capacity constraints
11: **if** $\lambda < 1$ **then**
12: $low = (low + high)/2$
13: **if** $|previousT - t| \leq \sigma$ && $previous\lambda = 1$ **then**
14: $FinalT = previousT$
15: **break**
16: **end if**
17: **else if** $\lambda = 1$ **then**
18: $high = (low + high)/2$
19: **if** $|previousT - t| \leq \sigma$ && $previous\lambda < 1$ **then**
20: $FinalT = t$
21: **break**
22: **end if**
23: **end if**
24: $previousT = t$
25: $t = (low + high)/2$
26: **end while**

Output: : $FinalT$

the algorithm, given a time t and by reconstructing the graph with the same nodes and same edges with the capacity $r_e * t$, for each edge e, the linear programming is applied to find the maximum minimum ratio λ. A binary search is then applied on the interval $[0, maxT]$ until the time t converges to the optimal value while λ is 1. $maxT$ is the upper bound of the maximum transmission time and can be calculated by the size of all the data over the minimum bandwidth link among all the links. We keep the value of time t and λ in the previous round of the loop in the variable $previousT$ and $previous\lambda$, respectively.

If $\lambda < 1$, there exists at least one tuple that only part of its data can be transmitted. Thus, we search the time on the higher part of the interval; otherwise, we search the time on the lower part. This process will continue until the difference on time t between the previous iteration and current iteration is less than a given small constant σ and $\lambda = 1$ in the previous iteration and $\lambda < 1$ in the current iteration (as shown in line 11 to line 16 in Algorithm 4) or $\lambda < 1$ in the previous iteration and $\lambda = 1$ in the current iteration (as shown in line 17 to line 23 in Algorithm 4). Then, we return the previous time in the first case and the current time in the second case when $\lambda = 1$, which guarantees all the data can be transmitted over the network.

Theorem 5. *Algorithm 4 can achieve near optimal for the network flow part with the given destinations for the clusters.*

Proof. Since λ indicates the least percent of each tuple that can be shipped without extending the capacity constraint, and λ can be obtained by linear programming, we can know that when $\lambda = 1$ by the given t, this is a feasible solution, even though t may be larger than the optimal solution. By the binary search for t, we terminate our algorithm on two cases. In the first case, when $\lambda < 1$ and $|previousT - t| \leq \sigma$ and $previous\lambda = 1$. Thus, we can know that the optimal solution must be between t and $previousT$ and $previousT$ deviates with optimal solution at most σ. This is because $previousT < \sigma + t \leq \sigma + OPT_{network}$, when $OPT_{network} >> \sigma$, this error can be omitted. The proof for the second case that when the difference of time in two sequential iteration is smaller than σ and $\lambda = 1$ is in the current iteration and $\lambda < 1$ in the previous iteration, is similar as above. In summary, our algorithm can be near optimal by the given destinations for the clusters.

□

4.2.2 Integrated into Load Balancing

For the load balancing part, our problem is similar to the well-known NP-complete problem to schedule jobs on the identical machines [24], where we can deem each cluster as a job, and the processing time of each job is thus the size of each cluster. The state-of-the-art solution for this problem is to sort the clusters in non-increasing order by the size, then put them into the least load machine one by one, which can achieve $4/3-$ approximation to the optimal solution [24].

Analyzing the structure of our problem can help us to design a good heuristic algorithm based on the theoretical guaranteed classical solution, while considering the interactions of these 2 parts. We adopt the slow start in MapReduce, which waits until all the mappers finish, i.e., all the intermediate data are generated and ready to be transmitted. Notice that the same key could be on different machines before the placement. We propose a network flow-based algorithm outlined in Algorithm 5. The input of the algorithm is a list of tuples $(i, u, x_{i,u})$, indicating key i of size $x_{i,u}$ at machine u. The algorithm first sorts the key i by non-increasing order of size of tuples with same key. For each key j, it forms the input set of $T_i = (m_i, s_i, n_i)$ for Algorithm 4. It then tries all the machines and assigns the key to the machine with minimal cost, which is the sum of the shuffling time and the computation time.

Algorithm 5 Network Flow-based Partition Placement

Input: A list of $(i, u, x_{i,u})$ tuples

1: Sort the key i by non-increasing order of size $\sum_{u=1}^{m} x_{i,u}$
2: **for** each key j **do**
3: $maxT = \sum_{i=1}^{j} x_{i,u}/min\ r_e$
4: **for** each machine n with load less than or equal to the average load **do**
5: $Y_{j,n} = 1$ and $Y_{j,r} = 0$, $\forall r \neq n$
6: Form a set $T = \{T_i = (s_i, b_i, d_i) | b_i = \sum_{i=1}^{j} Y_{i,d_i} x_{i,s_i}, s_i, d_i \in M\}$
7: t = Network-flow(T, σ)
8: **end for**
9: Reassign $Y_{j,n} = 1$ and $Y_{j,r} = 0$, $\forall r \neq n$, such that the $t + f_c(\sum_{i=1}^{j} Y_{i,n} x_{i,u})$ is minimized when assign to machine n
10: **end for**

Theorem 6. *The load balancing part of the algorithm 5 can achieve 2-approximation.*

Proof. Since the average load, ave, is less than or equal to the maximum load of the optimal solution, OPT, and the maximum size of any key is also less than or equal to that of OPT. Assume the maximum load is on machine j, and the tuple with size b is the last new one assigned to the machine j with load L at that time. Because the algorithm always chooses the node with load less than or equal to the average, thus $L \leq ave \leq OPT$ and $b \leq OPT$, as a result, the maximum load is $L + b \leq 2 * OPT$, in other words, the maximum load in worst case still can achieve 2-approximation to the optimal solution. □

Since the networking part of the algorithm can achieve $(1+\sigma/OPT)$-approximation and when $\sigma << OPT$, it can achieve near optimal solution. The load balancing part of the algorithm 5 can achieve 2-approximation. Thus, this algorithm can achieve a near 2-approximation solution. Although Algorithm 5 can achieve a well-bounded performance, in practice, we find that linear programming may take relatively longer time with larger number of machines and data size. To this end, we further design a greedy heuristic outlined in Algorithm 6. This algorithm works by trying to place the data flow on the links with the least finishing time while choosing the destination machine with the load less than or equal to the average.

In algorithm 6, the load that adds to machine is instead the sum of all the size with the same key over all the machines. For example, if a tuple $(i, u, x_{i,u})$, indicating key i with size $x_{i,u}$ located on machine u, is assigned to the machine n, then the load adding to the machine n should not just $x_{i,u}$, but the sum of all the size with key i, say, $\sum_{u=1}^{m} x_{i,u}$. Let $load_j$ indicate the amount of data that would go through the link j. and $rate_j$ denotes the bandwidth of link j.

Algorithm 6 Greedy Network-aware Partition Placement

Input: A list of $(i, u, x_{i,u})$ tuples, ordered in non-increasing order of size, $x_{i,u}$.

1: **for** each tuple $(i, u, x_{i,u})$ **do**
2: **if** key i has already has destination n **then**
3: P= set of shortest paths from u to n
4: **else**
5: P=set of shortest paths from u to n $\forall$ node n if n's load is less than or equal to average load of all the machines
6: **end if**
7: $min = MAX\_VALUE$
8: $minP = NULL$
9: **for** each path p in P **do**
10: $maxLink = MIN\_VALUE$
11: **for** each link j in p **do**
12: **if** $load_j/rate_j > maxLink$ **then**
13: $maxLink = load_j/rate_j$
14: **end if**
15: **end for**
16: **if** $maxLink < min$ **then**
17: $min = maxLink$
18: $minP = p$
19: **end if**
20: **end for**
21: Select the path $minP$
22: Update the load on $load_j$ for each link j on $minP$
23: Update the load on machines
24: **end for**

Chapter 5

Performance Evaluation

In this chapter, the performance evaluation includes two sets of algorithms: online load balancing algorithms and datacenter network-aware load balancing algorithms. We will show the performance evaluation results of comparing our online load balancing algorithms with the state-of-the-art algorithms in section 5.1 and the results of our datacenter network-aware load balancing algorithms in section 5.2.

5.1 Online Load Balancing Algorithms

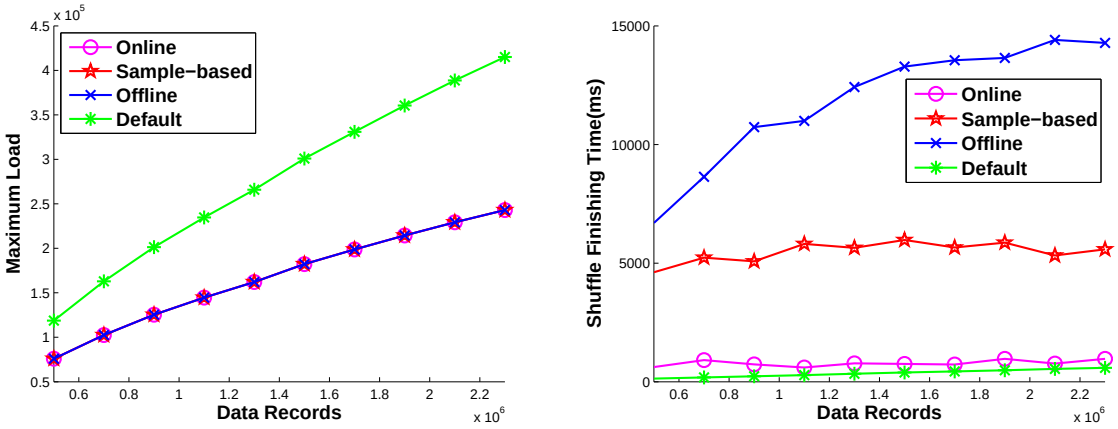
5.1.1 Simulation Setup

We evaluate our algorithms on both a real data trace and synthetic data. The real trace is a public data set [1], which contains the Wikipedia page-to-page link for each terms. This trace has a data size of 1 Gigabytes. We generate the synthetic data according to Zipf distribution with varying parameter s, by which we can control the skew of the data distribution.

In our performance evaluation, we not only simulate the data assignment process, but also the procedures that the reduce workers pull data from the specific place.

We evaluate both of our two algorithms: the List-based online scheduling algorithm (Online) and the Sample-based algorithm (Sample-based). Recall that the former is faster and the latter has better accuracy. We compare our algorithms with the current MapReduce algorithm with the default hash function (Default). To set a benchmark for our Online algorithm, we also compare it with the offline version (Offline), which sorts the keys by their frequencies and then assigns them to the machine with the least load so far. The primary evaluation criteria for these algorithms are the maximum load and the shuffle finishing time.

The default values in our evaluation are $z_{\frac{\alpha}{2}} = 1.96$, $\epsilon = 0.005$, the number of records is 1,000,000 and the number of identical machine is 20. The parameter s is set to 1 by default and we also vary it to examine its impact. Note that we scale the y axis to make the figure visually clean.



(a) Maximum load

(b) Shuffle finishing time

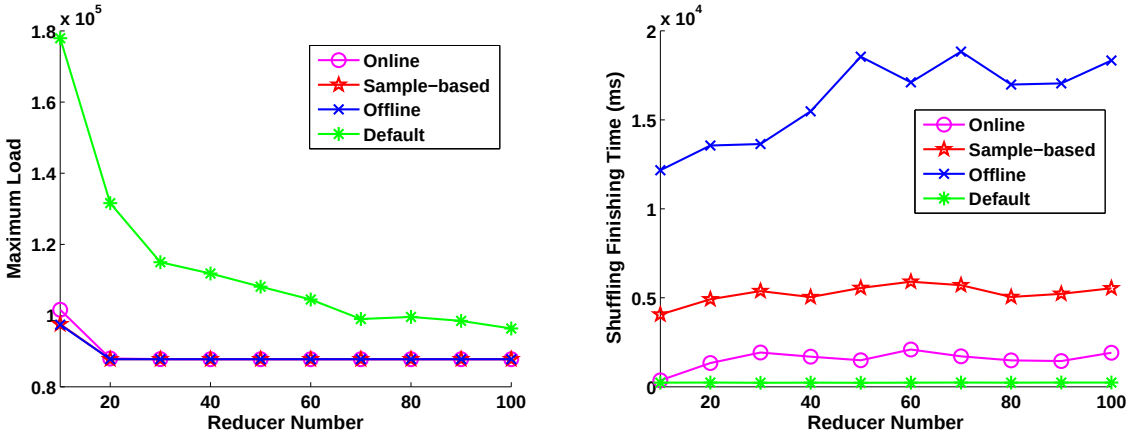
Figure 5.1: Different data record number on synthetic data

5.1.2 Results on Synthetic Data

Figure 5.1(a) shows the maximum load as a function of the number of data records on the synthetic data. Our data record is the key as mentioned earlier. We increase the number of data records from 0.5×10^6 to 2.3×10^6. We compare all the four algorithms. We can see that the Default algorithm performs much worse than all other three. When the number of data records is 2.1×10^6, the maximum load of the Default algorithm is 3.78×10^5 and our Online algorithm has a maximum load of only 2.3×10^5, an improvement of 39.15%. In addition, we also can see when the number of data records increases, the maximum loads of all the algorithms increase. This is not surprising as we need to process more data. However, the loads in our algorithms increase in a much slower pace as compared to the Baseline algorithm. Further, we can see that the performance of our Online algorithm is almost identical to the Offline algorithm. This indicates our algorithm not only bounds the worse case scenario theoretically, but also in practice performs much better than the theoretical bound.

Figure 5.1(b) shows the shuffle finishing time as a function of the number of data records on the synthetic data of all the four algorithms. We still vary the number of the records from 0.5×10^6 to 2.3×10^6. We can see that the Offline algorithm behaves much worse than all the other three algorithms. Especially, when the number of data records is 2.1×10^6, the shuffle finishing time of the Offline algorithm is 14000 ms and our Online algorithm has a shuffle finishing time of 1000 ms, an improvement of 14 times, while the shuffle finishing time of Sample-based algorithm is 5000 ms, improving almost 3 times. It is not surprising to see that as the number of data records grows, the shuffle finishing time increases. However, the loads in our algorithms increase in a much slower pace as compared to the Offline algorithm. Moreover, it shows that the shuffle finishing time of our Online algorithm is almost identical to the Default algorithm, which takes the least time to finish the *shuffle* subphase.

Figure 5.2(a) shows the maximum load as a function of reducer number on the synthetic data of all the four algorithms. We increase the reducer number from 10 to 100. We can see the



(a) Maximum load

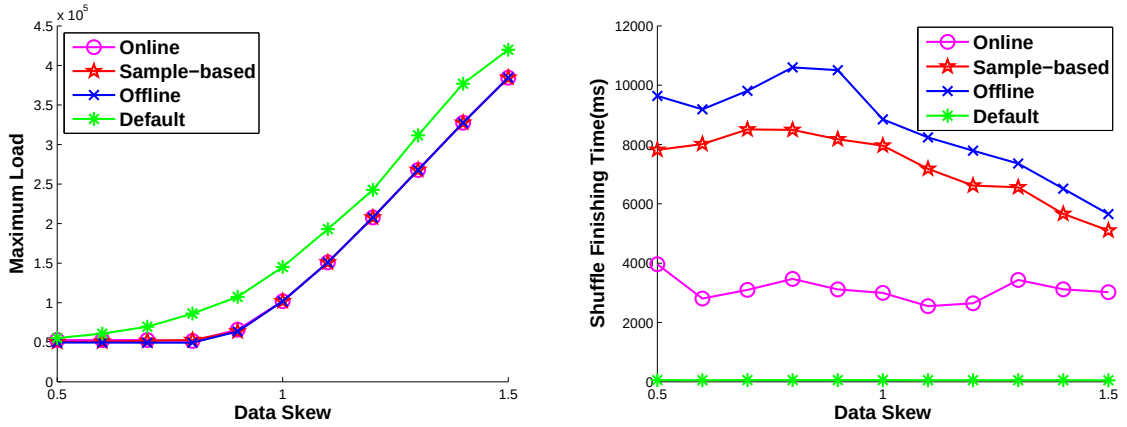
(b) Shuffle finishing time

Figure 5.2: Different reducer number on synthetic data

Default algorithm performs much worse than all other three. In particular, when the reducer number is 20, the maximum load of the Default algorithm is 1.779×10^5 and the maximum load of our Online algorithm is only 1.016×10^5, a reduction of 75.05%. It is natural that the maximum load decreases as the reducer number increases as the Default algorithm shows. However, it is interesting that the other three algorithms do not change much as the reducer number increases. We have checked the data distribution and found that there is one key that is extremely frequent. Our algorithms indeed have identified this key so that the performance of our Online algorithm is almost identical to that of the Offline algorithm.

Figure 5.2(b) shows the shuffle finishing time as a function of reducer number on the synthetic data with all the four algorithms. We have tested the performance by increasing the reducer number from 10 to 100. We have found that the shuffle finishing time of the Online algorithm is good as expected since the decision to assign a newly incoming key to a specific machine is made earlier in the Online algorithm than in the Sample-based algorithm, which is earlier than in the Offline algorithm. We should notice that as the reducer number increases, the shuffle finishing time also increases for all the algorithms expect for the Default algorithm. This is because we need to check whether the reducer machine contains the incoming keys or get the least load machine in these algorithms, which however does not cost so much time as compared with waiting in Offline algorithm.

Figure 5.3(a) compares the maximum load of all the four algorithms as a function of the skew on the synthetic data. We set the skew by adjusting parameter s from 0.5 to 1.5 in the Zipf distribution function. The larger s is, the more skew the data has. We could see that when the data are even, the parameter s is from 0.5 to 0.7, the maximum loads of all the four algorithms are almost the same. When the data distribution becomes more and more skewed, it is easy to recognize that the Default algorithm behaves much worse than all the other three. Not surprisingly, when the data are highly skewed, the balancing strategies are always better than the Default algorithm. This is because the maximum load is the frequency of the most

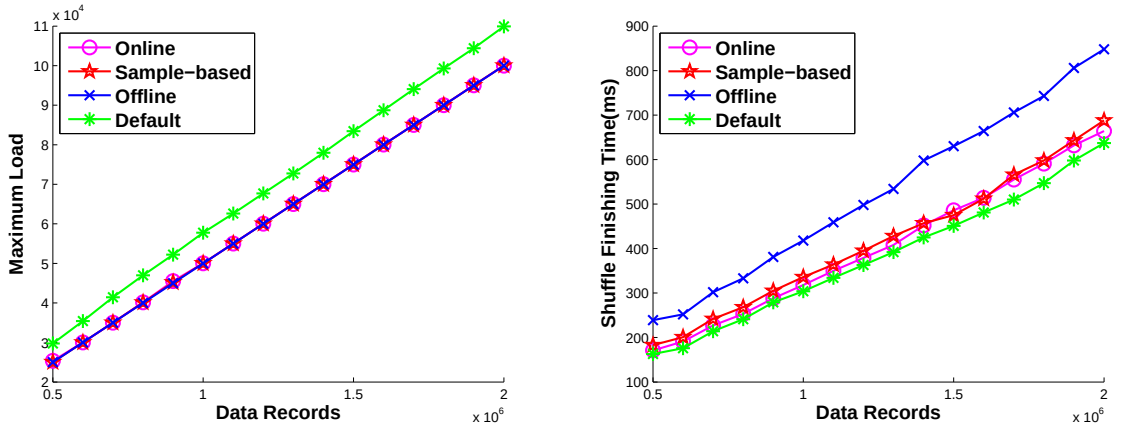


(a) Maximum load (b) Shuffle finishing time

Figure 5.3: Different data skew parameter on synthetic data

frequent key. For the Online algorithm and the Sample-based algorithm, it is easy to identify the most frequent key while it is not necessary to see all the keys.

Figure 5.3(b) shows the shuffle finishing time of all the four algorithms as a function of data skew on synthetic data. We still set the skew parameter from 0.5 to 1.5. Note that the data sets are generated independently for the different skew parameters, which leads the trend of the shuffle finishing time in each algorithm not to be as monotonic as expected.

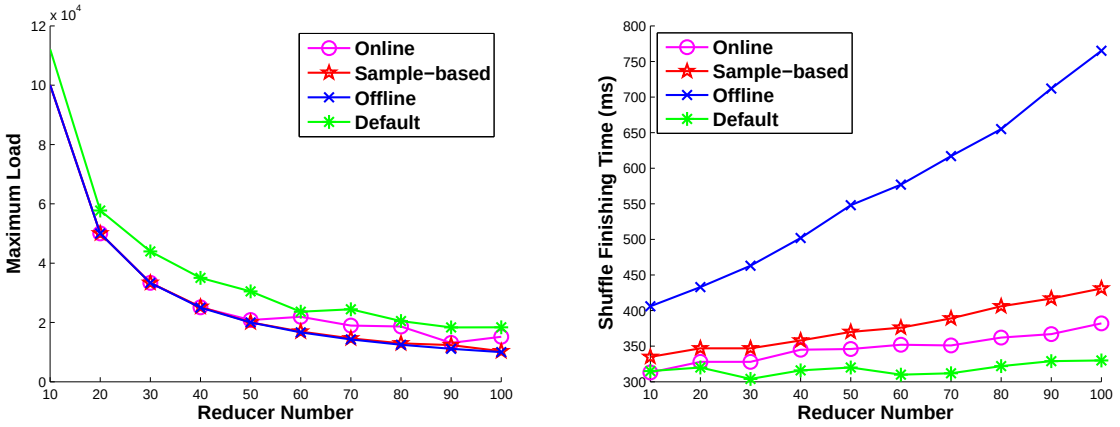


(a) Maximum load (b) Shuffle finishing time

Figure 5.4: Different data record number on real data trace

5.1.3 Results on Real Data

Figure 5.4(a) shows the result of the maximum load as a function of the number of data records in real public dataset. We tested the performance by setting the number of data records from 0.5×10^6 to 2.0×10^6. Again, our Online algorithm and Sample-based algorithm outperform



(a) Maximum load

(b) Shuffle finishing time

Figure 5.5: Different reducer number on real data trace

the Default algorithm while perform almost the same as the Offline algorithm, which further confirms the effectiveness of our algorithms.

Figure 5.4(b) compares the shuffle finishing time of all the four algorithms as a function of the number of data records in real public dataset. Our Online algorithm, Sample-based algorithm, and the Default algorithm achieve a reasonable better result than that of the Offline algorithm. In addition, we found that when the number of data records increases, the shuffling time of all the algorithms increase. This is because more data processed needs more time. However, our algorithms increase in a much slower pace as compared to the Offline algorithm. This is because when data records become larger, it will lead to the longer processing time of *Map* phase. As a result, it makes the Offline algorithm wait much longer.

Figure 5.5(a) shows the result of the maximum load as a function of reducer number of all the four algorithms in real public dataset. We evaluated the performances by increasing the reducer number from 10 to 100. We can see the Default algorithm performs worse than all the other three algorithms. It is natural that the maximum loads of all the algorithms decrease when the reducer number increases, which is as expected. Further, we can see that the performance of our Sample-based algorithm is almost identical to that of the Offline algorithm.

Figure 5.5(b) shows the result of the shuffle finishing time as a function of reducer number of all the four algorithms. We also found that the shuffle finishing time grows as the number of reduce workers increases. This is illustrative since it will cost much more time to check whether the coming key is assigned or not and find the least load machine. Interestingly, the shuffle finishing time of Offline algorithm increases so quickly than the other three. This gap appears probably because the sum of the waiting time and the increasing cost brought by the increasing number of reducers is very large.

In summary, our Online and Sample-based algorithm perform almost the same as the Offline algorithm. The two algorithms always perform better than the MapReduce Default algorithm

from maximum load point of view. Our algorithms also show comparable shuffle finishing time to that of the Default algorithm, and is better than that of the Offline algorithm.

5.2 Datacenter Network-Aware Load Balancing Algorithms

5.2.1 Methodology

We evaluate our solution by both synthesized data sets and a real data trace. The real trace is adopted from [1], which describes the Wikipedia page-to-page link for each term and can generate $130,160,392$ (key, value) pairs [20]. For the synthesized data sets, we adopt a typical setting as used in [28, 17], which generally follows the Zipf distribution. By default, we set the skewness parameter $s = 0.7$ and the size of a data set is $5,000,000$ (key, value) pairs with each pair of the size $1MB$. We use a typical datacenter network topology, namely, the fat-tree topology [3] [1] and set the pod number to 12, which leads to 432 hosts and 180 switches in total. The machines used for the MapReduce workers are randomly chosen from the 432 hosts and the default number of the reducers is 100. All the links in the network are set to the bandwidth of $1GB/s$. To emulate the real world datacenter network traffic, we also generate the background traffic according to the characteristics of the datacenter network reported in [4].

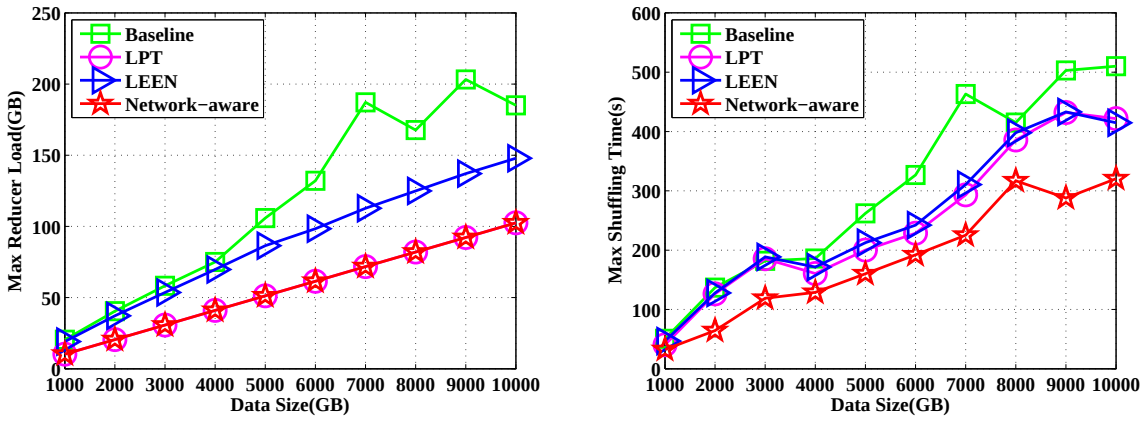
For comparison, we implement another three algorithms: the Baseline algorithm that uses the default routing algorithm and hash function in MapReduce to determine the cluster placement; a state-of-art load balancing algorithm named LPT [17], which sorts the keys by the decreasing order of their size and each time assigns one key to the machine currently with the least load; and LEEN [20], which is the state-of-art locality- and fairness- aware key partitioning algorithm proposed for MapReduce. Due to the linear programming procedure in the algorithm 5 might take long time to solve, especially when the number of the reducers and the number of the (key, value) pairs grow large, we use our greedy network-aware partition placement algorithm (denoted as Network-aware) for the evaluation, where the observed performance difference when compared to our linear programming approach on small scale datasets, is generally within 20%. Also, to better understand how our solution can improve the overall performance, instead of using an arbitrary complexity function on the load of each machine, we separate the final results into the maximum load on a reducer (Max Reducer Load) and the maximum shuffling time (Max Shuffling Time) during the shuffle subphase. We then investigate how our solution performs with different data size, reducer number and data skewness, respectively.

5.2.2 Results on Synthetic Datasets

A. Scale with Different Data Size

Figure 5.6 shows how the four algorithms perform with different size of the synthesized datasets from $1,000$ GB to $10,000$ GB. It is easy to see that in Figure 5.6(a), the max reducer loads of both the Baseline and LEEN algorithms become much worse when the size of the datasets

[1] It is worth noting that our algorithms can apply to the general datacenter network topology, because we did not assume any network topology in our algorithm design.



(a) Max reducer load

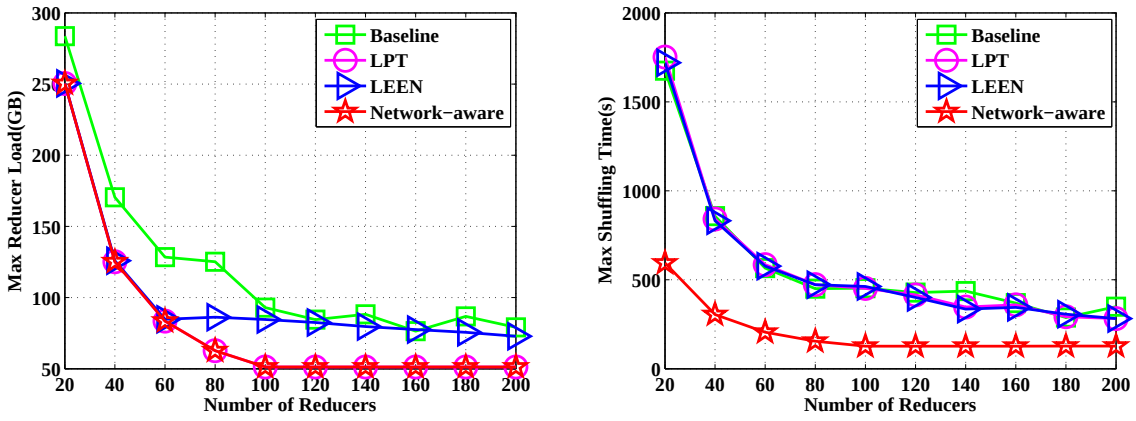
(b) Max shuffling time

Figure 5.6: Scalability with Different Data Size on Synthetic Datasets.

grows. Especially when the data size is 10,000 GB, the performance degradation can be up to 60% as compared to the LPT algorithm, which is a state-of-the-art design dedicated to optimize the load balance. On the other hand, our Network-aware solution performs almost identical to the LPT algorithm, further demonstrating that our solution can effectively balance the load among the reducers.

Figure 5.6(b) shows how the maximum shuffling time of the four algorithms changes with different data size. We can see that our solution always outperforms the other three algorithms with the gain up to 40.69%. The Baseline algorithm generally performs the worst as it uses the default hash function in MapReduce which is oblivious to the costs of moving data across the network. The LEEN algorithm performs slightly better than the LPT algorithm as it considers the data locality and tries to move the minimum amount of data over the network, which can help reduce the max shuffling time if the data happen to go through the routing paths with similar amount of background traffics. However, as neither of the two algorithms is aware of the traffic bottlenecks in the network, their max shuffling time can significantly increase if any link along the routing path is heavily loaded by the background traffic, while our Network-aware solution can successfully avoid such situations and constantly achieve much lower max shuffling time than the other three algorithms, especially when the data size becomes large.

Comparing the results in both Figures 5.6(a) and 5.6(b), it is not surprising to see that as the data size grows, both the max reducer load and the max shuffling time will increase. However, our solution increases in a much slower pace than the other three algorithms, indicating a good scalability for big data processing. Moveover, our solution can successfully reduce the max shuffling time while still performing almost the same to the LPT algorithm on optimizing the load balance, which further confirms the superiority of our algorithm design.



(a) Max reducer load

(b) Max shuffling time

Figure 5.7: Adaptability to Various Reducer Number on Synthetic Datasets.

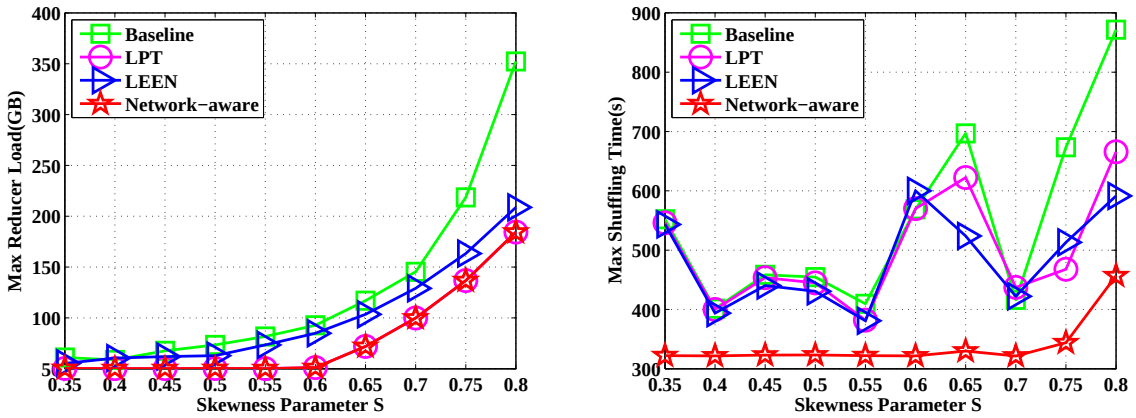
B. Adapting to Various Reducer Number

We next examine how the performance changes for all the four algorithms by varying the number of reducers from 20 to 200. The results are illustrated in Figure 5.7. In term of the max reducer load as shown in Figure 5.7(a), our Network-aware solution stays almost the same as the LPT algorithm and always outperforms the LEEN and Baseline algorithms with the reduction up to 44.3% on the max reducer load. One interesting observation is that our solution becomes stable when the number of reducers is greater than 100. A close look at the data distribution reveals that there is one key that is extremely frequent and our solution successfully assigns this key solely on a reducer while still keeping the loads on the other reducers less than this, which further confirms that our solution can reach a near optimal load balance.

For the maximum shuffling time shown in Figure 5.7(b), our Network-aware algorithm always outperforms the other three algorithms, with the performance gain up to 55.22%. In both Figures 5.7(a) and 5.7(b), we can see that our solution can well adapt to the various number of available reducer resources, which can be an important feature in practice, especially given the dynamic resource availability due to the interferences from other applications in the datacenter, this feature can allow the available resources to be fully exploited.

C. Resist to Diverse Data Skewness

Figure 5.8 shows the results of the max reducer load and max shuffling time as a function of the skewness on different synthesized datasets. We vary the skewness parameter s in the Zipf distribution from 0.35 to 0.8, where the larger s is, the more skew the dataset is. As shown in Figure 5.8(a), we can see that when the data is less skew (i.e., $s = 0.35$), although the gaps of the max reducer loads among all the algorithms are relatively small, our solution performs almost the same to the LPT algorithm, and always outperforms the LEEN and Baseline algorithms, which becomes more remarkable as the skewness grows large, with a gain as much as 47.73%. The result of the maximum shuffling time shown in Figure 5.8(b) also shows that our algorithm

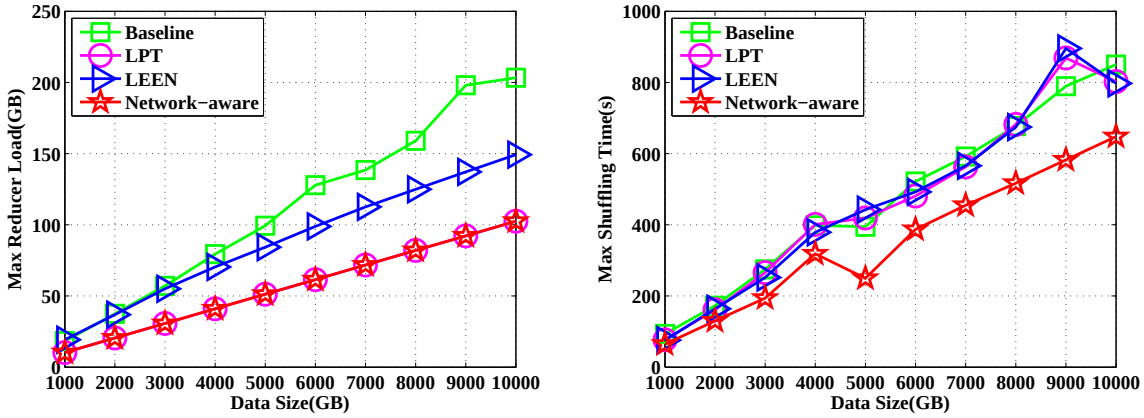


(a) Max reducer load (b) Max shuffling time

Figure 5.8: Resistance to Diverse Data Skewness on Synthetic Datasets.

always outperform the other three algorithms with a reduction up to 49.01%, which further verifies the effectiveness of our algorithm. It is interesting to see that the other three algorithms are experiencing notable fluctuations in Figure 5.8(b). This is because these three algorithms are blind to the network status when assigning the (key, value) pairs to the machines, which can lead to unexpected heavy traffics on certain links and cause network bottlenecks.

5.2.3 Results on Real Data Trace

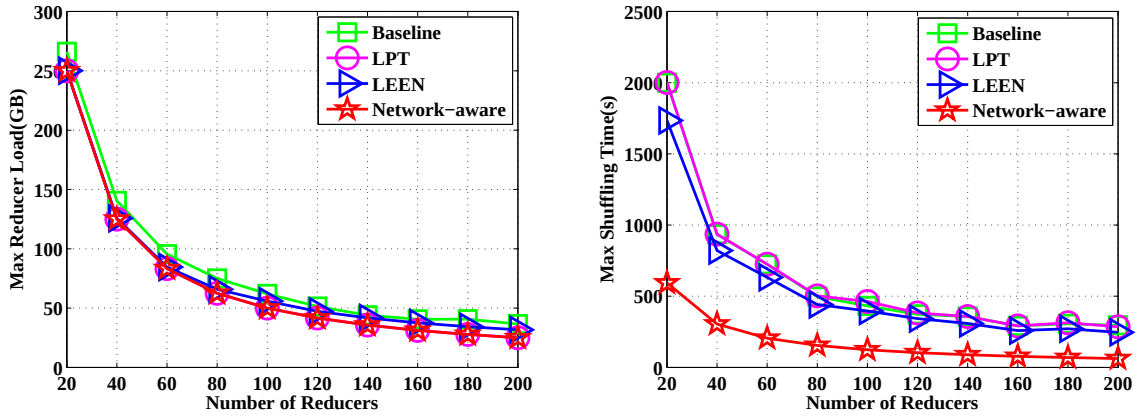


(a) Max reducer load (b) Max shuffling time

Figure 5.9: Performance as a Function of Data Size on Real Data Trace.

Besides synthesized datasets, we also conduct extensive evaluations with a real trace from [1], where we emulate the map phase with the original tuples from the trace randomly allocated on different machines. We vary the number of (key, value) pairs from $1,000,000$ to $10,000,000$. Given that the size of one (key, value) pair is 1 MB, the size of the data is thus from $1,000$ GB to $10,000$ GB. Figure 5.9 shows the results of the max reducer load and the max shuffling

time, where the trend matches well to that in Figure 5.6, further confirming the validity of the evaluation results under the synthesized datasets.



(a) Max reducer load

(b) Max shuffling time

Figure 5.10: Performance as a Function of Reducer Number on Real Data Trace.

We also examine how the four algorithms perform with different reducer number. The results are shown in Figure 5.10. Again, our solution has the best performance in terms of both max reducer load and max shuffling time. Figure 5.10(b) shows a similar trend and gain to Figure 5.7(b), while the gain on the max reducer load in Figure 5.10(a) is relatively small compared to that shown in Figure 5.7(a). We examine the key id and distribution in the dataset and find that the key ids happen to be well aligned with the hash function mapping mechanism, which results in a relatively good load balancing even for the Baseline algorithm. Nevertheless, the gain in Figure 5.10(a) is still up to 32.5% when the reducers number is 180.

In summary, our Network-aware solution always outperforms the other three algorithms by achieving the lowest max shuffling time. It also achieves almost the same max reducer load to the LPT algorithm that focuses on optimizing the load balancing. Moreover, our solution can adapt to various amount of available reducer resources and resist well to diverse data skewness with excellent scalability on the data size.

Chapter 6

Further Discussion and Conclusion

6.1 Further Discussion on Online Load Balancing

Note that the original sorted-balance algorithm can achieve $4/3OPT$ [24]. Our semi-online algorithm achieves $3/2OPT$ plus some additive error for the K most frequent keys. The gap could be further shrunk through advanced algorithm design, or the error could be reduced with a higher tolerance to its probabilistic nature. The keys can also be finely classified into different groups according to certain weights, so as to refine the results. In addition, if the data distribution follows a known distribution, e.g., Zipf distribution, the parameters can be better estimated, making the identification of the K initial keys much easier and more accurate. The additive error can expectedly be made smaller, too.

We are also highly interested in implementing the online algorithms in real world MapReduce packages, e.g, the Hadoop system. We have currently set up a server cluster for this purpose, which is a fully controlled and configurable environment with homogeneous machines. In the long run, we expect to move the implementation to the public cloud environment. The real world network overhead and I/O cost will have to be accommodated, together with the heterogeneous and instable machine resources. These new constraints would all affect the overall job finishing time, since intermediate pairs need to be passed from the map workers to the reducer workers. The load balancing problem and evaluation criteria will have to be refined to reflect these changes.

Finally, it is worth noting that we focus on the skewed distribution of the input data in our work. Another kind of skew is caused by certain portions of the input data inherently taking longer time to process than others. If the reduce function is non-linear, load balance cannot be ensured only by scheduling the same number of pairs into each reducer. Earlier studies have shown that, in a case that the number of the records in the fastest reduce task is 2 times of that of the slowest, there is a factor of 6 difference between their running times [41]. In this case, it is necessary to know the runtime of the user-defined reduce function in advance and incorporate it into algorithm design, which however remain quite difficult in practice.

6.2 Further Discussion on Datacenter-Network-Aware Load Balancing

It is also worth noting that the shuffle subphase of MapReduce can overlap with the map phase, i.e., shuffle can start before all the map tasks finish, which may reduce the finishing time of the reduce phase. However, the early start of the shuffle subphase will occupy the task slots and do nothing but pull data, which may lead to performance degradation of map tasks. In addition, once the shuffle subphase starts earlier than the end of the map phase, for each new key in the generated (key, value) pairs, we have to dynamically decide which machine it should be assigned to, which can become more challenging for optimization as we do not have the actual size of each cluster yet. It is thus interesting to further investigate such issues as how early the shuffle subphase can start so as to have an overall performance improvement.

Also, the algorithms proposed in this paper are centralized solutions. Given that the current MapReduce often has a centralized entity for management, our solution can be implemented there, thus being practically useful for real-world big data processing. It will also work well in a software defined networking environment [14] that allows centralized management on network services and performance. On the other hand, in certain situations, e.g., for better scalability or avoiding the single point of failure, a solution that can route the data through the network in a distributed manner (as the current fat-tree routing mechanism does) may be preferred. Yet, this may cause performance degradation as we now need to make distributed decisions with partial or imprecise information about the network. Therefore, it is worth further exploring on how to deal with these issues and integrate our idea into a distributed mechanism.

6.3 Conclusion

In this thesis, we first investigated the data skew in MapReduce application with real world data, which motivates us to balance the load of all the reduce workers. Then, by conducting a detailed timing flow through the *Map* phase and *Reduce* phase, including *shuffle*, *sort*, *reduce* subphases, we found it is really necessary to consider online operations of load balancing in MapReduce context.

Motivated by these reasons, we first proposed an online model which assigns the key once it comes in and provided a List-based Online algorithm with provable 2-competitive ratio. We further suggested a sample-based model, which has an initial offline phase, where the keys are collected to form a sample, and then the sample, as well as the remaining keys, are assigned as they come in based on estimates of key frequencies obtained from the initial sample and developed a Sample-based algorithm that can achieve a finishing time of $3/2OPT + \epsilon N$, with probability at least $1-2\alpha, 0 < \alpha < 1$. Both algorithms work well in our performance evaluation.

Besides the load balancing issue with the skewed data input problem, we also examined the datacenter-network-aware load balancing problem in the shuffle subphase in MapReduce. Different from earlier studies on load balancing that assume the network inside a datacenter is of negligible delay and infinite bandwidth, we considered the more realistic constraints imposed by

the real world datacenter networks. We started from the data placement with generic network cost models, and present a 2-approximation solution. We further extended our model to consider the bandwidth constraints in real datacenter networks, and demonstrated that the corresponding problem can be decomposed into two subproblems for network flow and load balancing, respectively. We have shown effective solutions to both of them, which together yield a complete solution toward near optimal datacenter-network-aware load balancing. A much simpler and yet performance-wise comparable greedy algorithm was also developed for fast implementation in practice. The performance evaluation results confirmed the advantage of our algorithms.

Bibliography

[1] Wikipedia page-to-page link. *http://haselgrove.id.au/wikipedia.htm.*, 2013.

[2] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, December 1974.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM*, Seattle, WA, USA, 2008.

[4] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM Internet Measurement Conference*, Melbourne, Australia, 2010.

[5] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proceedings of the Very Large Data Bases Endowment*, September 2010.

[6] Hyunseok Chang, M. Kodialam, R.R. Kompella, T. V. Lakshman, Myungjin Lee, and S. Mukherjee. Scheduling in mapreduce-like systems for fast completion time. In *Proceedings of IEEE INFOCOM*, Shanghai, China, April 2011.

[7] Fangfei Chen, M. Kodialam, and T. V. Lakshman. Joint scheduling of processing and shuffle phases in mapreduce systems. In *Proceedings of IEEE INFOCOM*, Orlando, Florida, USA, March 2012.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, December 2004.

[9] D. DeWitt and M. Stonebraker. Mapreduce: A major step backwards. *The Database Column*, 2008.

[10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High-Performance Parallel and Distributed Computing*, Chicago, Illinois, USA, June 2010.

[11] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. Mapreduce for data intensive scientific analyses. In *Proceedings of the 4th IEEE eScience*, Indianapolis, Indiana, USA, December 2008.

[12] M. Englert, D. Ozmen, and M. Westermann. The power of reordering for online minimum makespan scheduling. In *Proceedings of the 49th IEEE Annual Symposium on Foundations of Computer Science*, Philadelphia, Pennsylvania, USA, October 2008.

[13] Rudolf Fleischer and Michaela Wahl. Online scheduling revisited. In *Proceedings of the 8th Annual European Symposium on Algorithms*, ücken, Germany, September 2000.

[14] Open Networking Fundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.

[15] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 1969.

[16] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM*, Barcelona, Spain, 2009.

[17] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Handling data skew in mapreduce. In *Closer*, 2011.

[18] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *Proceedings of 28th IEEE International Conference on Data Engineering*, Washington, DC, USA, April 2012.

[19] M. Hammoud and M.F. Sakr. Locality-aware reduce task scheduling for mapreduce. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science*, Athens, Greece, 2011.

[20] S. Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, Indianapolis, Indiana, 2010.

[21] S. Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, 2010.

[22] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: Measurements & analysis. In *Proceedings of the 9th ACM Internet Measurement Conference*, Chicago, Illinois, USA, 2009.

[23] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, November 1984.

[24] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[25] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD*, Scottsdale, Arizona, USA, May 2012.

[26] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. Choreo: Network-aware task placement for cloud applications. In *Proceedings of the 13th ACM Internet Measurement Conference*, Barcelona, Spain, 2013.

[27] Willis Lang and Jignesh M. Patel. Energy management for mapreduce clusters. *Proceedings of the Very Large Data Bases Endowment*, September 2010.

[28] Yanfang Le, Jiangchuan Liu, Funda Ergun, and Dan Wang. Online load balancing for mapreduce with skewed data input. In *Proceedings of IEEE INFOCOM*, Toronto, Canada, 2014.

[29] JanKarel Lenstra, DavidB. Shmoys, and ÃĽva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1-3):259–271, 1990.

[30] Jacob Leverich and Christos Kozyrakis. On the energy (in) efficiency of hadoop clusters. *ACM SIGOPS Operating Systems Review*, January 2010.

[31] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proceedings of the 2011 ACM SIGMOD*, Athens, Greece, June 2011.

[32] Jimmy Lin. The curse of zipf and limits to parallelization:a look at the stragglers problem in mapreduce. In *The 7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, July 2009.

[33] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD*, Indianapolis, Indiana, USA, June 2010.

[34] Kristi Morton, Magdalena Balazinska, and Dan Grossman. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD*, Indianapolis, Indiana, USA, June 2010.

[35] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus: locality-aware resource allocation for mapreduce in a cloud. In *Proceedings of Super Computing*, Seattle, Washington, 2011.

[36] Smriti R. Ramakrishnan, Garret Swart, and Aleksey Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, San Jose, California, USA, 2012.

[37] III Rudin, F. John, and R. Chandrasekaran. Improved bounds for the online scheduling problem. *SIAM Journal on Computing*, March 2003.

[38] Farhad Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *J. ACM*, 37(2):318–334, April 1990.

[39] J.W. Stamos and H.C. Young. A symmetric fragment and replicate algorithm for distributed joins. *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[40] Jian Tan, Xiaoqiao Meng, and Li Zhang. Coupling task progress for mapreduce resource-aware scheduling. In *Proceedings of IEEE INFOCOM*, Turin, Italy, April 2013.

[41] Magdalena Balazinska Y Kwon and Jerome Rolia Bill Howe. A study of skew in mapreduce applications. In *The 5th Open Cirrus Summit*, 2011.

[42] Weipeng Yan and Per-Ake Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21th Very Large Data Bases Conference*, Zurich, Switzerland, September 1995.

[43] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD*, Beijing, China, June 2007.

[44] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, December 2008.