A DATA-DRIVEN APPROACH TO OPTIMAL RESOURCE MANAGEMENT FOR LARGE-SCALE DATA PROCESSING PLATFORMS

By

Wei Yan

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

Computer Science

August 2015

Nashville, Tennessee

Approved:

Professor Yuan Xue

Professor Aniruddha S. Gokhale

Professor Bradley A. Malin

Professor Douglas C. Schmidt

Dr. Amr A. Awadallah

ACKNOWLEDGEMENTS

During my Ph. D journey at Vanderbilt, I had the great pleasure of working with an amazing group of talented people.

Foremost, I would like to express my deepest appreciation and thanks to my advisor, Professor Yuan Xue, who has guided me as a passionate, inspirational, and supportive mentor throughput my graduate study. This work would not have been possible without her support and patience. I want to express sincere gratitude to my dissertation committee: Professor Aniruddha S. Gokhale, Professor Bradley A. Malin, Professor Douglas C. Schmidt and Dr. Amr A. Awadallah. I appreciate their advice and guidance.

I would also like to thank my colleagues at VANETS group and the Institute for Software Integrated Systems. They are like families sharing frustration, joy and hope. I will never forget their help and encouragement. I am very thankful to my friends here, with whom I have had a great time at Vandy.

Most importantly, I want to express my deepest gratitude to my parents and wife Li Li. I would not have come this far without their love and support. This work is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Managing and analyzing data at a large scale has become a key skill driving business and science. The total big data management and analysis market reached $11.59 billion in 2012 and is predicted to be $47 billion by 2017 [68]. Companies like Google, Facebook, and Yahoo! maintain and process petabytes of data, including web/software logs, click streams, customer interactions, and other types of information. Advanced analysis techniques (such as data mining, statistical modeling and machine learning) are now applied routinely to big data to drive automated processes for applications like spam and fraud detection, advertisement placement, web documents analysis and customer relational management. They can lead to cost savings and higher revenue. However, success in the big data era is about more than the ability to process large amount of data. It is about retrieving revenue from these huge datasets in a timely and cost-effective manner.

To perform large-scale big data processing in a cost-effective manner, several companies have developed distributed data storage and processing systems on large clusters of shared-nothing commodity servers, including Google's File Systems [31], Bigtable [14], MapReduce [25], Yahoo!'s Pig system [61], Facebook's Hive system [65], and Microsoft's Dryad [39]. The MapReduce [25] framework and its open source implementation Hadoop [64] are becoming increasingly popular in the enterprise setting as well as in scientific and academic settings. The development of cloud computing especially lets people easily rent computing resources on-demand, bill on a pay-as-you-go basis. As an example, the New York Times used 100 Amazon Elastic Computing Cloud (EC2) [75] instances and a Hadoop application to process 4 TB of raw image TIFF data into 11 million finished PDFs in the space of 24 hours at a computation cost of about $240 [74].

However, MapReduce [25] is not a silver bullet, and there has been much work probing its limitations [52], both from a theoretical [3, 41] and empirical perspective, mainly by exploring classes of algorithms that cannot be efficiently implemented with it [9, 13, 28, 92] (e.g., iterative computing, interactive analytics). Fortunately, more large-scale data processing systems have been proposed, developed and deployed. For instance, Dremel [56] and its open source implementations (Apache Drill [6], Cloudera's Impala [19] and Facebook's Presto [63]) support interactive analytics by introducing massive parallel processing (MPP) mechanisms. Spark [89] provides more efficient iterative/streaming/interactive analytics by introducing in-memory computation. Pregel [54] and its open source implementation Giraph [7] provide programming model for iterative graph processing, and Storm [66, 78] can support stream processing.

Problem Statement

In such a shared cluster, computing resources are allocated to various data processing jobs. Each machine runs (or may run) a CPU-intensive, one or more MapReduce jobs, and more random applications. This has the advantages of statistical multiplexing of physical resources and centralized asset management, as well as workload-specific benefits, such as sharing of common datasets and intermediate computational results. Nevertheless, such sharing environments present new resource management challenges. First, the workload of data processing jobs depends on the input data – not only the data size, but more importantly, the internal data structure and semantics, which is usually unknown a priori. Second, unlike traditional dedicated clusters, data processing jobs in sharing clusters are highly diverse in terms of their resource and performance requirements. Optimal resource management is needed to ensure high resource utilization and

optimize the performance for each job with minimum expenditure. We call this the "optimal resource management problem in large-scale data processing platforms".

In this dissertation, we solve this problem from two perspectives: *within a data processing job*, and *across data processing jobs*. For a single job, we want to optimize its performance (e.g., job completion time) under its allocated resources. For multiple jobs, we want the computing resources to be allocated and utilized efficiently and meet each job's performance requirements with minimum expenditure.

Optimal Resource Management within a Data Processing Job

The objective of optimal resource management within a data processing job is to optimize the job performance using its allocated resources, such as minimize the job completion time.

Large-scale data processing systems provide a good abstraction of distributed operations over a cluster of machines. For example, in MapReduce, users only need to implement *map* and *reduce* functions. The underlying run-time system achieves parallelism by partitioning the data and processing different partitions concurrently using multiple machines. It is clear from the success of large-scale data processing systems that parallelism is an effective mean to achieve dramatic speedup and scaleup.

However, the basic techniques that large-scale data processing systems use for exploiting parallelism is vulnerable to the presence of **skew** in the underlying data. Simply put, if the underlying data is sufficiently skewed, load imbalance in the resulting parallel task execution will swamp any of the gains due to the parallelism and unacceptable performance will result.

In addressing the skew problem for a data processing job, we use MapReduce as our target application. Arguably, MapReduce is one of the most important classes of

applications. Thus, the problem is narrowed down to optimizing a MapReduce job's performance with the presence of skew. We measure a MapReduce job's performance as its completion time.

Fundamentally, a MapReduce job is executed through two primary phases. In the *map* phase, a function is applied in parallel to data from various input datasets. This function yields intermediate results in the form of a list of key-value pairs. Pairs with the same key are subsequently grouped together and allocated to a reduce task based on a *partition* function. In the *reduce* phase, the reduce tasks run in parallel over each key group to produce the final results. When the intermediate key groups are not uniformly distributed, load skew may occur at the reducer phase.

Reduce-phase skew may arise from two sources. First, the hash-based key-group-to-reducer assignment mechanism, as adopted by the default partition function, may not pack the key groups for even load. We refer to this factor as *partition skew*. Second, the workload of certain key groups may be significantly larger than others and exceed the balancing capacity of the partition function. These are called *expensive key groups* and even if one such group is assigned to a particular reduce task, that task will still be a straggler in the reduce phase.

In response to reduce-phase skew, a general skew-handling solution is to do a significant amount of preprocessing (called *profiling*) in order to profile the data distribution and compute a new workload assignment plan designed to minimize load imbalance. To solve partition skew, a packing operation can be adopted that packs key groups into several sets, each of which has even workload. For expensive key groups, key group division is required, which divides expensive key groups into subgroups first and then performs the packing operation. However, there still are two major challenges that make it hard to implement such a skew-handling solution.

- First, *the input data is huge*. The profiling process needs to build a profile that captures the data distribution information. For example, in a database join, this profile contains the number of records for each join key. However, it is impossible to maintain an accurate profile for MapReduce applications as they always have to process millions or billions of records. Maintaining such "big data" would introduce much overhead. A scalable data structure is needed to capture the data distribution information through data profiling.

  Additionally, as the input data is huge, it may take a substantial amount of time to build the profile if the profiling process operates on entire data. This additional overhead may diminish the benefits coming from achieving reduce-phase load balancing. To overcome this problem, we need to build an efficient sampling strategy to fast the profiling process.

- Second, *the reduce function is a black box*. For the relational operators such as join and aggregate, the semantics are well understood, and many specialized techniques to handle the skew problem are available [26]. In contrast, in MapReduce, the *reduce* function is implemented by users and the system has no idea of the semantics. This issue brings two challenges: (1) key group workload estimation: the system cannot calculate the workload for each key group; (2) key group division: the system does not know how to perform the key group division without losing the original semantics.

Optimal Resource Management across Data Processing Jobs

The objective of optimal resource management across data processing jobs is to schedule jobs/tasks to optimize their performance and achieve high resource utilization.

In this dissertation, we choose large-scale interactive ad hoc queries as our target application.

Interactive ad hoc data query over massive datasets has recently gained significant traction. Massively parallel data query and analysis frameworks (e.g., Dremel [56], Impala [19]) are built and deployed to support SQL-like queries over distributed and partitioned data in a clustering environment. In these systems, each query is first compiled into a plan tree, which is then decomposed into several query fragments. Each fragment is dispatched to the machines where its data blocks locate, and each machine gets one or more fragments, Depend on the query semantics (i.e., SQL operation), the execution of each query is then converted into a set of coordinated tasks including data retrieval, intermediate result computation and transfer, and result aggregation. As a result, each query consumes different amount of resource (e.g., CPU, memory, bandwidth) at each machine.

Since significant benefits can often be realized by sharing the cluster among multiple clients, a principal challenge here is the development of efficient resource management mechanism to support concurrent multiple interactive queries. Coordinated management of multiple resources in clustering environment is critical to provide a guarantee on service-level agreement (SLA) for each client. Without any resource coordination, query tasks may create system bottleneck, leading to long query's response time, low resource utilization, and unfairness among different clients.

To alleviate the resource collision between different queries and maximize the cluster utilization, we need a coordinated resource management solution. Three major challenges need to be solved when designing such a resource management framework.

- First, this framework needs to capture the various resource consumption for each query at different machines. As we discussed above, each query is converted into a

set of tasks, and each task performs different SQL operations on different datasets. This makes each query consumes various resources at different machines. Without considering this characteristic, the resource management framework cannot management the cluster resources efficiently.

- Second, the framework should maximize the cluster resource utilization. This includes two-fold requirements: given the per-query resource consumption profile, (1) minimize the available resource fragments (utilize the resource as much as possible), and (2) alleviate the resource collision between different queries (avoid the some machines overloaded).

- Thirdly, the framework should consider the performance requirements coming from the client-side. Modern production clusters normally require to implement certain type of client-side performance requirements. For example, certain fairness objectives(proportional, max-min, etc) are always required, in which the cluster resources are shared among various queries in a fair sharing way. To be practical, the resource management solution needs to provide an interface for taking inputs of client-side performance requirements.

Contributions of this Dissertation

The high level contribution of the dissertation is twofold. First, we have demonstrated our optimal resource management approach for a single MapReduce job. We first deploy a scalable profiling mechanism, utilizing a scalable data structure called *sketch* to capture the data distribution information. And then we utilize the built sketches to direct the process of assigning key groups to reducers in a load balancing manner. Second, we have built an utility-based optimization framework for coordinating resource allocation

for large-scale interactive data query systems. We first profile the resource consumption for each query at different machines, and then put the "profile" into a price-based algorithm. This algorithm can find a unique "maximum utility" allocation point, at which point the cluster resource utilization is Pareto-optimal. Meanwhile, certain client-side performance requirements can be achieved when we choose appropriate utility functions for queries.

The detailed main contributions of the dissertation are:

- In Chapter III, we study the reduce-phase skew problem in MapReduce and propose a sketch-based profiling approach to capture the key group size statistics. In particular, we compress the key group sizes into a two-dimensional array called sketch. An *optimal sketch packing* algorithm is developed that operates bin packing operation on top of the sketch to provide a load balancing solution. This approach can solve the partition skew for applications whose key group workload is proportional to its key group size. Details are illustrated in Chapter III.

- In Chapter IV, we study the reduce-phase skew in record linkage application [58, 43]. A record linkage application involves two types of datasets (dataset R and data S) and performs join-like operation. Its reduce-phase skew is always caused by expensive key groups. The *optimal sketch packing* algorithm cannot be deployed here directly as it can only solve partition skew, while expensive key groups requires key group division.

  To mitigate such reduce-phase skew, we first profile the data distribution of each type of dataset using sketch structure, and perform sketch multiplication to estimate key group workload. Then we perform sketch cell division on top of the built sketches to mitigate skew from expensive key groups and achieve reduce-phase load balancing in record linkage application. Details are illustrated in Chapter IV.

- In Chapter V, we study the coordinated resource management problem in a multi-tenant cluster that supports interactive ad hoc queries over massive datasets. We adopt a utility-based optimization framework where the objective is to optimize the resource utilization, coordinate among multiple resources from different machines, and maintain certain fairness among different clients.

  Concretely, each client is associated with a utility, which corresponds to the query rate it is able to issue. The objective of the optimal resource allocation is to maximize the aggregate utility of all clients, subject to the cluster resource constraints. We solve this utility-based resource allocation problem via a price-based approach. Here, a "price" signal is associated with each type of resource (e.g., CPU, memory) for each machine. For each query, we: (1) collect resource prices from the machines where the query runs its fragments; (2) adjust a new query rate based on the updated prices such that the query's "net benefit", the utility minus the resource cost, is maximized. For each machine, we: (1) collect the new rates for queries that run fragments on current machine; (2) update the price for each type of resource based on the availability. The resource prices and query rates are updated iteratively, until reaching a "maximum utility" point. Details are illustrated in Chapter V.

Outline of Dissertation

The dissertation is organized as follows. Chapter II introduces the large-scale data processing ecosystem and the existing literatures related with the work in this dissertation. Chapter III and IV present two approaches to achieve reduce-phase load balancing

for various MapReduce applications. Chapter V studies the coordinated resource management for large-scale interactive query systems. Finally, we conclude and discuss possible areas for future work in Chapter VI.

CHAPTER II

BACKGROUND AND RELATED WORK

In this chapter, we take a closer look at our target environment, the large-scale data processing ecosystem. We also investigate related work regarding load balancing in MapReduce, and resource management for large-scale interactive query systems.

The Large-Scale Data Processing Ecosystem

The MapReduce Framework

MapReduce [25] was proposed to simplify large-scale data processing on distributed and parallel architectures, particularly clusters of commodity hardware. The main idea of this programming model is to hide details of the data distribution and the load balancing and let users focus on data processing. A MapReduce program consists of two primitives, *map* and *reduce*, as shown below:

map:: $(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$

reduce:: $(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3)$

Users can implement their processing logic by specifying customized *map* and *reduce* functions written in a general-purpose language Java or Python. The *map* function is invoked for every key-value pair (k_1, v_1) in the input data to output key-value pairs of the form (k_2, v_2). The *reduce* function is invoked for every unique key k_2 and corresponding values list(v_2) in the map output. The *reduce* outputs key-value pairs of the form list(v_3).

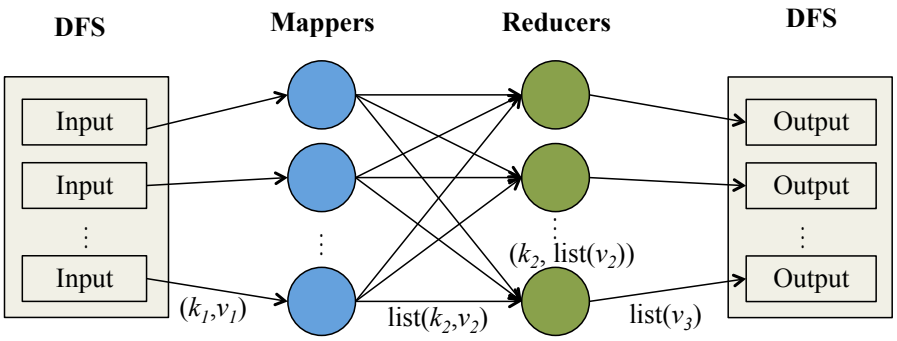


Figure II.1: The MapReduce workflow.

Figure II.1 illustrates the execution flow of a MapReduce job. The MapReduce job execution can be decomposed further into phases with map and reduce tasks. In the map phase, input data is divided into equal-size chunks (64 MB by default) and each data chunk is processed by a map task; the intermediate outputs of the map tasks are collected locally and grouped based on their key values. Based on a (default hashing or user-defined) partition function, these key groups are allocated to the appropriate reducers depending on their keys. Once the map phase is completed and the intermediate results have been transferred to the reducers, the reduce phase begins. In this phase, the reduce function is applied in parallel to each key group and produces the final results.

Hadoop [64] is the most popular open-source implementation of a MapReduce framework that follows the design laid out in the original paper. A number of companies use Hadoop in production deployments for applications such as web indexing, data mining, report generation, log analysis, machine learning, and financial analysis. Infrastructure-as-a-Service cloud platforms like Amazon EC2 [75] have made it easier than ever to run Hadoop workloads by allowing users to instantly provision clusters and pay only for the time and resources used. A combination of features contributes to Hadoop's increasing popularity, including fault tolerance, data-local scheduling, ability to operate

in a heterogeneous environment, handling of straggler tasks[1], as well as a modular and customizable architecture.

The Interactive Ad Hoc Query Systems

The MapReduce [25] framework plays an important role in large-scale data processing. However, MapReduce is not a silver bullet and still has its own limitations. Due to latency, most MapReduce applications are developed for batch workloads, which take minutes to hours to finish. To provide a tool for large-scale interactive data analysis over massive datasets, Google developed Dremel [56]. Dremel is a system that supports the interactive analytics of very large datasets over shared clusters of commodity machines. Dremel can execute many queries over large datasets that would ordinarily require a sequence of MapReduce [25] jobs, but at a fraction of the execution time. Dremel's architecture borrows the concept of a serving tree used in distributed search engines [24]. A Dremel query gets pushed down the tree and is rewritten at each step. The result of the query is assembled by aggregating the replies received from lower levels of the tree. Dremel provides a high-level, SQL-like language to express ad hoc queries. In contrast to layers such as Pig [61] and Hive [65], it executes queries natively without translating them into MapReduce jobs. Another important feature is that Dremel adopts a novel columnar storage format, which enables it to read less data from secondary storage and reduce CPU cost due to cheaper compression.

Cloudera Impala [19] is an open-source implementation of Dremel by bringing real-time, ad hoc query capability to Hadoop, complementing traditional MapReduce batch processing. Impala is an open-source full-integrated, state-of-the-art MPP SQL query

[1] A straggler is a task that performs poorly typically due to faulty hardware or misconfigurations.

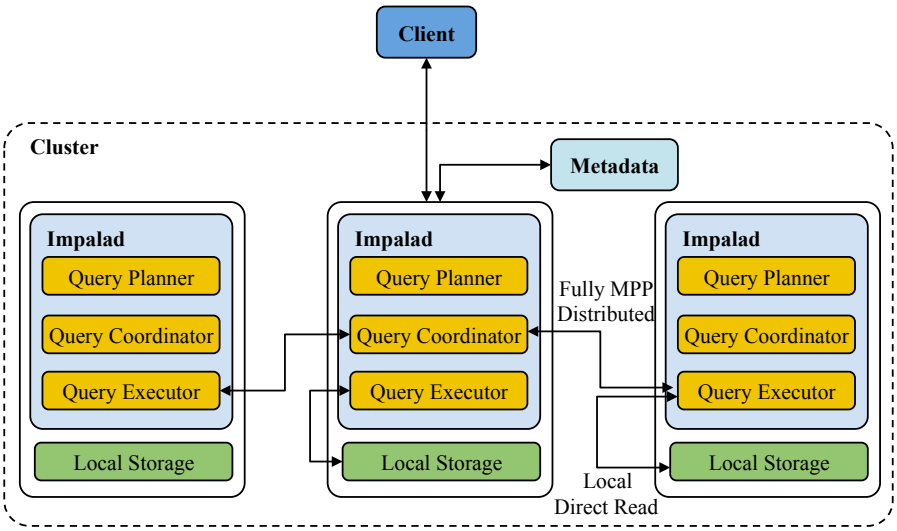


Figure II.2: Impala architecture.

engine designed specifically to leverage the flexibility and scalability of Hadoop. Impala's goal is to combine the familiar SQL support and multi-user performance of a traditional analytic database with the scalability and flexibility of Apache Hadoop.

This dissertation uses Impala as the target interactive query system. As illustrated in Figure II.2, the system has two main tiers. The topic tier is the user interface. To support ad hoc queries, Impala provides a programming interface that is consistent with the standard SQL model. The core is the query processing tier. It is implemented as a long running daemon on each machine and submits its own queries. This daemon process (i.e., *Impalad* in Figure II.2) comprises the query planner, query coordinator, and the query execution engine. After fetching the data location information from the metastore, the local query planner compiles the query into a pipeline execution plan, consisting of several query fragments. The query coordinator dispatches these query fragments to other machines, where each fragment is executed by the query execution engine to process over the local data. All local processing results are assembled together at the query coordinator and returned to the client.

Other Large-Scale Data Processing Systems

Pregel [54] provides a programming model for iterative graph processing. In Pregel, programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology. The vertex-centric approach is reminiscent of MapReduce in that users focus on a local action, processing each item independently, and the system composes these actions to lift computation to a large dataset. The high-level organization of Pregel programs is inspired by Valiant's Bulk Synchronous Parallel model [80]. Giraph [7] originated as the open-source counterpart to Pregel.

Storm [66, 78] is a real-time fault-tolerant and distributed stream data processing system. The basic Storm data processing architecture consists of *streams of tuples* flowing through *topologies*. A topology is a directed graph where the vertices represent computation and the edges represent the data flow between the computation components. Vertices are further divided into two disjoint sets – spouts and bolts. Spouts are tuple sources for the topology, pulling data from upstream (like Kafka [46]). On the other hand, bolts process the incoming tuples and pass them to the next set of bolts downstream.

Existing Resource Management within a Data Processing Job

With given resources, the completion time of a data processing job depends on its last finished task. To fully utilize the benefits from parallelism, a general approach is to balance the workload assigned to concurrent tasks. However, load imbalance is common in large-scale data processing, and may diminish the benefits from parallelism.

In this section, we will review this problem and existing load balancing solutions in MapReduce.

In MapReduce, a task is identified as an outlier if its time to finish is longer than 1.5x the median task duration in its phase [5]. In production clusters, 25% of phases have more than 15% of their tasks as outliers. 80% of the runtime outliers last less than 2.5 times the phase's median task duration, with a uniform probability of being delayed by between 1.5x to 2.5x. The tail is heavy and long – 10% of the tasks take up more than 10x the median duration. By carefully measuring a large MapReduce cluster, Ananthanarayanan *et al.* [5] identify three main causes of outliers: machine characteristics, network characteristics and data skew. We will mainly focus on data skew in this proposal.

Data Skew Problem in MapReduce

In MapReduce, data skew refers to the problem that some map/reduce tasks cost longer time because of their input data. Kwon *et al.* [48] present a detailed analysis of the data skew problem, which can be grouped into two categories (map-phase skew and reduce-phase skew).

Map-Phase Skew

Map-phase skew refers to the problem that some map tasks take longer time than other map tasks in the same phase. The map-phase skew has three different causes: *expensive record*, *heterogeneous maps* and *non-homomorphic maps*.

Expensive record: Map tasks typically process a collection of records , one-by-one. Ideally, the processing time does not vary significantly from record to record. However, depending on the application, some records may require more CPU and memory to

process than others. These expensive records may simply be larger than other records, or the map algorithm's running time may depend on the record's value.

Heterogeneous maps: MapReduce is a unary operator, but can be used to emulate an n-ary operation by logically concatenating multiple datasets into a single input file. Each dataset may need to be processed differently, leading to a multi-modal distribution of task running times. For example, SkewedJoin [38] is one of the join implementations in the Pig system [61]. Each map task in SkewedJoin distributes frequent join keys from one of the input datasets in a round-robin fashion to reduce tasks, but broadcasts joining records from the other dataset to all reduce tasks. These two algorithms exhibit different running times because the map tasks that perform the broadcasts do more I/O than the other map tasks.

Non-homomorphic map: One of the key features of the MapReduce framework is that users can run arbitrary code as long as it conforms to the MapReduce's map and reduce interfaces, and typical initialization and cleanup. Such flexibility enables users to push, when necessary, the boundaries of what map and reduce phases have been designed to do: each map output can depend on a group of input records, i.e., the map task is non-homomorphic. For example, although the conventional join algorithm in MapReduce requires both map and reduce phases, if the data are sorted on the join attribute, the join can be implemented directly in the map phase using a sort-merge algorithm. In this scenario, a map task may run what is normally reduce logic such as aggregation or join, consuming a group of records as a unit rather than a single record as in a typical MapReduce application. Thus, the map tasks may experience reduce-phase skew discussed in the following section.

Reduce-Phase Skew

Reduce-phase skew refers to the problem that some reduce tasks take longer time than other reducer tasks in the same phase. There are two types of reduce-phase skews: *partition skew* which is unique to reducer, and *expensive key groups* which is analogous to the expensive record in map-phase skew.

Partition skew: In MapReduce, the outputs of map tasks are distributed among reduce tasks via hash partitioning (by default) or some user-defined partitioning logic. The default hash partitioning is usually adequate to evenly distribute the data. However, reduce-phase skew can still arise in practice. Consider the inverted indexing application, each map task processes several documents and outputs terms as intermediate data. If the hash function partitions the intermediate data based on the first letter of a term, reducers processing more popular letters are assigned a disproportional amounts of data, which induces reduce-phase skew.

Expensive key groups: In MapReduce, each reduce task processes a sequence of (key, set of values) pairs. As in the case of expensive records processed by map, expensive (key, set of values) pairs can skew the runtime of reduce tasks. Since reduce operates on key groups instead of individual records, the expensive input problem can be more pronounced, especially when the reduce is a holistic operation that requires memory proportional to the size of the input data. A holistic reduce may load the entire associated values with a reduce key in memory and run complex algorithms (e.g., find clusters in a multi-dimensional input data using a spatial index, perform complex joins, and analyze the activities of a user given a subgraph of social network).

The running time of a holistic reduce that runs a complex algorithm can significantly vary per reduce key. For example, the reduce function of MapReduce-based record linkage (Chapter IV) performs a similarity calculation for records within key groups.

Although all reduce tasks are extremely well balanced in terms of the reduce keys, there is a factor of 44 difference between the maximum and the average task running time. This suggests that some reduce keys are more expensive to process than others.

Existing Skew-Avoidance Solutions in MapReduce

In this section, we survey skew-avoidance solutions to mitigate reduce-phase skew in MapReduce. Skew-avoidance solutions always involve a data profiling phase, which collects the data distribution information (in terms of key group size) and then assigns workload to reducers in a load balancing manner.

We organize existing solutions from two perspectives: profiling mechanism (accurate or approximate), and supported applications (application-transparent or application-specific). An accurate profiling mechanism collects accurate size for each key group, while approximate profiling only provides an approximate estimation for key group size. Application-transparent solutions are designed to achieve reduce-phase load balancing for most MapReduce applications, while application-specific solutions are built for some particular applications (e.g., join operation, record linkage application). We review each part in the following sections.

Different from skew-avoidance solutions, SkewTune [49] is a type of skew-handling mechanism that adopts a work-stealing mechanism to mitigate reduce-phase skew at runtime. SkewTune dynamically monitors the task execution and estimates the remaining time for each task. Whenever a machine becomes idea, SkewTune mitigates workload from other overloaded machines to the idle ones. SkewTune always involves with several rounds of moving workload across different machines, which can take a non-trivial amount time and consume network bandwidth, especially with a large dataset.

Additionally, SkewTune can only work with key group granularity and cannot handle skew caused by expensive key groups.

Application-Transparent Solutions

We further divide application-transparent solutions into supporting *simple* applications and supporting *complex* applications. Most application-transparent transparent solutions are designed for very simple applications (e.g., PageRank, inverted indexing), which can use key group size to represent the key group workload. That is, the workload of a key group is proportional to its size.

Ibrahim *et al.* [37] propose a reduce-phase load balancing solution called LEEN to support simple MapReduce applications. LEEN first profiles the size for each key group, and then performs a bin packing operation on these key groups. Key groups are grouped into several partitions, and each reducer will be assigned to one partition. [27, 34] follow the same approach. However, there remains several hurdles to translating such an approach into practice. (1) The first problem corresponds to scalability. Specifically, when the number of key-value pairs is large, significant overhead will be incurred during the profiling phase. (2) A second, more substantial problem, arises when the number of key groups is huge. In this case, the data structure that maintains the profile of the key group sizes can impose extremely high memory requirements. (3) A third problem is that the algorithm which operates on this data structure (and utilizes the key group size profile) for the design of an optimal partition function incurs non-trivial computational overhead. (4) A fourth problem is that this approach works on key group granularity, and cannot mitigate skew caused by expensive key groups.

To build a scalable solution, Gufler *et al.* [33] present TopCluster, which builds a key group histogram with k buckets devoted to the top-k frequent keys and 1 bucket devoted to all remaining keys. TopCluster provides a way to estimate key group size with

acceptable memory requirement. However, such a top-k histogram cannot be deployed to mitigate load imbalance as it only has size information for the top-k key groups. The remaining key groups still may skew the reducer loads. We further analyze TopCluster's limitations in mitigating reduce-phase skew in Chapter III, and propose a new sketch-based profiling mechanism which compresses all key group sizes in a sketch structure instead of only top-k. Through such an approach, our approach can deliver stable load balancing performance.

Ramakrishnan *et al.* [67] provide another scalable load balancing solution by introducing progressive sampling. This approach works in an efficient way as it only builds the data profiles use a small subset of the key-value pairs. Additionally, this approach also supports key group division which can mitigate skew caused by expensive key groups. However, its key group division is very simple as it directly divides key-value pairs with an expensive key group into several subgroups and assumes this operation does not hurt the original reduce function semantics. However, this assumption cannot be satisfied by most MapReduce applications. For example, in a join operation, the direct division of records with the same key into two parts may cause some record pairs to be dropped.

The progressive sampling mechanism introduced in [67] provides an efficient way to fast the profiling phase.

Application-Specific Solutions

There are also several solutions that work for special types of applications. The join is one of the most common operations in MapReduce, and there are many different ways to implement a join operation. Pig [61], a declarative layer of Hadoop, implements an algorithm proposed in the parallel database literature [26] to handle data skew in a join algorithm. Blanas *et al.* [11] surveyed and compared different join implementations in

MapReduce. Okcan *et al.* [60] studied a theta-join in MapReduce. In general, these algorithms adopts a simple sampling mechanism with a sampling rate (such as, 5%) to profile the data distribution.

Record linkage is another join-like MapReduce application. In record linkage, records are divided into several key groups and records within each key group are compared with each other to generate record pair similarities. Kolb *et al.* [43] implement a load balancing solution for record linkage application. This approach profiles the number of records within each key group in an accurate way, and then assigns each reducer with same amount of record pairs. However, the solution proposed in [43] is not scalable because it adopts accurate profiling. We propose a scalable solution for record linkage in Chapter IV.

Speculative Execution in MapReduce

Besides data skew, other factors cause some tasks to be slower, such as heterogeneous environment (e.g., machine, network, etc.). Even when each task is assigned with even load, stragglers may still exist. MapReduce itself has a general mechanism called speculative execution to alleviate the problem of stragglers. When a MapReduce job is close to completion, the job master schedules backup executions of the remaining in-progress tasks. The task is marked as completed whenever either the primary or the backup execution completes.

MapReduce's speculative execution identifies the straggler according to workload completed and assumes that tasks make progress linearly. However, this assumption does not always hold, especially in heterogeneous environment. Zaharia *et al.* [90] propose a speculative task scheduling algorithm called Longest Approximate Time to End (LATE). LATE predicts the remaining time for each running task and identifies the

task with the maximum remaining time as the straggler. The completion time of a task is predicted by tracking task progress instead of work completed.

Both of the above solutions duplicate stragglers at the end of each phase when free slots are available. Mantri [5] introduces a probability-based restart algorithm which attempts to identify straggler tasks as early as possible. Mantri uses two variants of restart, the first kills a running task and restarts it elsewhere, the second schedules a duplicate copy. Mantri restarts only when the probability of restarting a new task costing less time is very high. Mantri kills and restarts a task if its remaining time is so large that there is a more than even chance that a restart would finish sooner. The "kill and restart" scheme drastically improves the job completion time without requiring extra slots. However, the current job scheduler incurs a queuing delay before restarting a task, that can be large and high variant. Hence, Mantri considers scheduling duplicates. Scheduling a duplicate results in the minimum completion time of the two copies and provides a safety net when estimates are noisy or the queuing delay is large. However, it requires an extra slot and if allowed to run to finish, consumes extra computation resource that will increase the job completion time if outstanding tasks are prevented from starting. Hence, when there are outstanding tasks and no spare slots, Mantri schedules a duplicate only if the total amounting of computation resource consumed decreases. By scheduling duplicates conservatively and pruning aggressively, Mantri has a high success rate of its restarts.

In general, compared with skew mitigation, these approaches work from another perspective and focus on task scheduling solution. However, as they doesn't balance the load assigned to each task, a reduce task may still take longer time compared to others even we restart it in another place. Thus, balance the task load is an essential step to achieve reduce-phase load balancing.

Existing Resource Management across Data Processing Jobs

With the development of the big data applications, various resource management frameworks have been proposed and deployed in production clusters, including Hadoop YARN [81], Omega [73] and Mesos [36]. A popular resource allocation mechanism adopted by these systems is fair sharing [88, 32, 10].

Originally, MapReduce was aimed at large (generally periodic) batch jobs. As such, the natural goal would be to decrease the completion time required for a batch window. For such scenarios, a simple job scheduling scheme as *First In, First Out* (FIFO) works very well. However, the use of MapReduce has evolved (in the natural and standard manner) towards more user interaction. There are now many more ad-hoc query MapReduce jobs, and they share cluster resource with the batch work. For users who submit these queries, expecting quick results, schemes like FIFO does not work well. This is because a large job can starve a small, user-submitted job which arrives even a little later. To avoid such a starvation, a fair sharing [87, 88] mechanism was introduced by Hadoop to achieve fairness among jobs. Quincy [40] is a flow-based fair-share scheduler for Dryad [39], which is a more generalized variant of the MapReduce framework. It maps the scheduling problem to a graph in which edge weights and capacities represent data locality and fairness, and then it uses standard optimization solvers to find a schedule. Sandholm and Lai [70] use user-assigned and regulate priorities to optimize the MapReduce schedule by adjusting resource share dynamically and eliminating bottlenecks.

YARN inherits the scheduling mechanism directly from MapReduce, and its RM runs the scheduler, supporting FIFO and fair scheduling (implemented as fair scheduler [72] and capacity scheduler [71]). As discussed above, fair sharing is designed to run heterogeneous applications as a shared, multi-tenant cluster in an operator-friendly

manner while maximizing the throughput and the utilization of the cluster. The YARN fair scheduler maintains multiple queues, and shares resources fairly among these queues. Each job is submitted to one queue. Queues can be arranged in a hierarchy to divide resources and configured with weights to share the cluster in specific proportions. YARN is capable of scheduling multiple resource types (e.g., memory, CPU). By default, the fair scheduler achieves fairness only on memory. It can be configured to schedule with both memory and CPU, using the notion of dominant resource fairness [32].

In fair sharing, when one queue does not need its full guaranteed share, the excess is split between other queues having running applications. This let the scheduler guarantee capacity for queues while utilizing resources efficiently when these queues don't contain applications.

Sometimes a queue may need to take its sharing resource back from other queues due to resource shortage of its own jobs. The scheduler typically replies on preemption to coordinate such resource fair sharing. Specifically, preempting a task means terminating the task and using the resources to schedule a different task. By default YARN does not deploy any work-preserving preemption, and leaves this work to the AM. Several works [4, 15] have discussed possible work-preserving preemption for particular workload.

Curino *et al*. [23] introduce a reservation-based scheduling algorithm for YARN. In this approach, a resource description language is proposed that provides a more powerful way for each job to specify its resource requirements. Besides the amount of resources needed, this language supports more features including time window and dependency. This new scheduling algorithm gives the system flexibility in allocating resource across several jobs, while also allowing it to plan ahead and determine whether it can satisfy any given job's resource request.

Different from the central design of the YARN scheduler, Google's Omega system [73] introduces a shared-state scheduling approach. Omega consists of several individual schedulers, each of which is implemented with different policies. There is no central resource allocator in Omega, and all of the resource-allocation decisions take place in the schedulers. Omega maintains a master copy of all resource allocations in the cluster called a *cell state*. Each scheduler is given a private, local frequently-updated copy of cell state that is uses for making scheduling decisions. Each scheduler updates its local resource allocation with the master copy, and which also takes care of the conflict if multiple schedulers allocate the same resource.

Sparrow [62] is another newly developed scheduling algorithm for large-scale data processing systems. Sparrow provides a decentralized, randomized sampling approach that provides near-optimal performance while avoiding the throughput and availability limitations of a centralized design. Sparrow consists several distributed running schedulers, and each task is submitted to a randomly selected scheduler. Each machine is treated as an indivisible resource, and maintains a waiting queue containing its allocated tasks. For a new incoming task, the scheduler checks the queue size for randomly selected machines, and assigns the task to the machine with the least queue size.

[82, 83, 84] has mainly focused on global optimization for MapReduce jobs with respect to system-centric performance metrics that are ignored by the fair sharing. However, for systems with significant sharing (especially multi-tenant clusters), though, these approaches also present limitation. Global optimizations are not targeted to be consistent with each job's individual resource valuations. Optimizations are performed as though all jobs are equally important while ignoring individual job value of the resources which vary based on the immediacy, importance, the resource demands of the job's computing needs. Thus, in allocating resources to competing jobs, these solutions

are unlikely to deliver the greatest value to the jobs for a given set of resources. Different from the above solutions, our approach jointly consider the per-job performance requirement and the entire cluster utilization.

Another set of work is market-based resource allocation mechanisms [86]. Researchers have proposed using economic approaches to resource allocation in computer systems. Tools offered by microeconomics for addressing competition and pricing are thought useful in handling computing resource allocation problem. And the pricing-based methods can reveal the true needs of clients who compete for the shared resources and allocate resource more efficiently. The application of market-based resource allocation ranges from computer networking [86], distributed file systems [47], distributed database [76] to computational job scheduling problems [59, 18].

CHAPTER III

SCALABLE AND ROBUST KEY GROUP SIZE ESTIMATION FOR REDUCER LOAD BALANCING IN MAPREDUCE

In this chapter, we first look at the reduce-phase skew problem in MapReduce, where reduce tasks are often assigned imbalanced load (in terms of key groups). Even though several approaches [37, 43, 60] have been proposed to solve the reduce-phase skew problem, most of the solutions are not scalable and cannot be deployed with "big data". To mitigate these limitations, we introduce a sketch-based data structure for capturing MapReduce key group size statistics and present an optimal packing algorithm which assigns the key groups to the reducers in a load balancing manner. We perform an empirical evaluation with several real and synthetic datasets over two distinct types of applications. The results show that our load balancing algorithm can strongly mitigate the reduce-phase skew. It can decrease the overall job completion time by 45.5% of the default settings in Hadoop and by 38.3% in comparison to the state-of-the-art solution.

We begin by motivating the need for reduce-phase load balancing and discuss the limitations of existing approaches in Section III.1. Our system and approach are discussed and analyzed in detail in Section III.2 and III.3. Then, we discuss the implementation with Hadoop in Section III.4 and demonstrate the evaluation results with various applications in Section III.5. We finally summarize this chapter in Section III.6.

Motivation

A MapReduce job is executed through two primary phases. In the *map* phase, a function is applied in parallel to data from various input datasets. This function yields intermediate results in the form of a list of key-value pairs. Pairs with the same key

are subsequently grouped together and allocated to a reduce task based on a *partition function*. In the *reduce* phase, the reduce task runs in parallel over each key group to produce the final result.

Despite its merits, MapReduce suffers from certain limitations. One of the most significant issues is referred to as the *reduce-phase skew problem* [48]. This occurs when a varying number of intermediate key-value pairs are assigned to reducers, thus skewing the load in the reduce phase. It has been shown that this problem can lead to suboptimal performance of many applications executed over the MapReduce framework [43, 60, 91].

In the MapReduce framework, the reducer workload is computed as the sum of workload of all key groups assigned to it. The key group workload, in turn, is a function of its size. As a result, the load at the reducers depends on two factors: (1) the key group size (i.e., the number of records within each key group), and (2) the partition function, which assigns a key group to a reducer. The default partition function adopted by MapReduce leverages a hash function to perform the key-group-to-reducer assignment. In this function, the hash value of a key is directly mapped to the index of a reducer through a simple modulo operation. Depending on the distribution of key group sizes, this hashing-based partition function can lead to highly skewed workload distribution at the reducers, which deteriorates MapReduce performance.

Figure III.1 presents an example that demonstrates the current blind hashing partition. In this example, we have 7 key groups represented by k_1 through k_7, and each key group has different number of records (i.e., key-value pairs). According to the hashing partition (here we assume the hashing function is (i mod 3) for key group k_i), the 3 reducers have imbalanced workload. The reducer 1 needs to process 96 key-value pairs, while the reducer 3 only has 42 key-value pairs. This imbalanced workload distribution

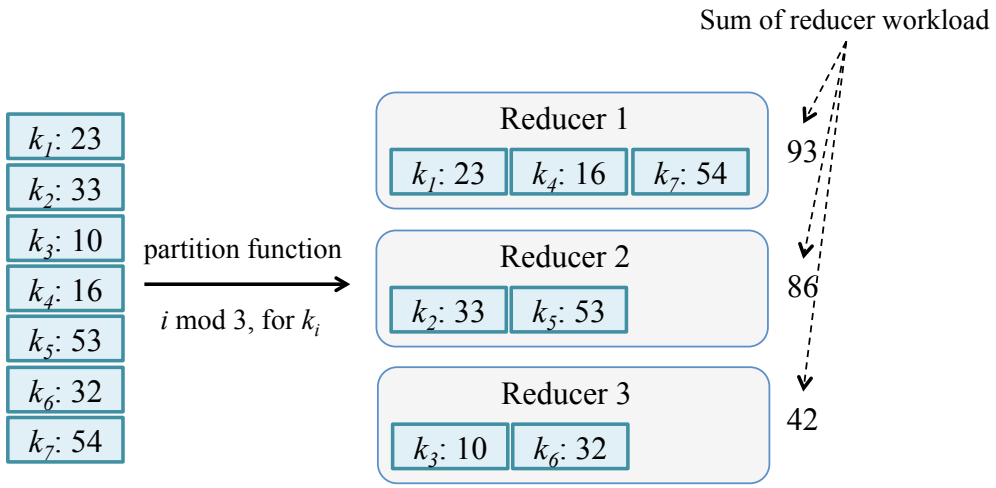


Figure III.1: An example of reduce-phase skew caused by hashing-based partition in MapReduce framework.

prolongs reduce execution on some reducers (reducer 1 here), and degrades the system performance.

To mitigate reduce-phase skew, recent methodologies [33, 37] have augmented the MapReduce framework by profiling key group sizes and designing a new partition function based on the profiling statistics. However, there remain several hurdles to translating such an approach into practice. The first problem corresponds to scalability. Specifically, when the number of key-value pairs is large, significant overhead will be incurred during the profiling phase. A second, more substantial problem, arises when the number of key groups is large. In this case, the data structure that maintains the profile of the key group sizes can impose extremely high memory requirements. A third problem is that the algorithm which operates on this data structure (and utilizes the key group size profile for the design of an optimal partition function) incurs non-trivial computational overhead.

To summarize, in order to recognize and mitigate the reducer-phase skew, we believe the MapReduce framework can be significantly enhanced through: (1) a data representation (or summary) and estimation method for the distribution of key group sizes; (2) a load balancing key-group-to-reducer assignment method that is specialized to the data representation.

Additionally, the following properties are desirable for such a representation and load balancing strategy:

- *Scalable*: As discussed above, when the number of key groups is very large, it is unrealistic to maintain the sizes of all key groups in memory. Thus, a scalable representation is needed, where the memory cost and the computation cost are independent of the number of key groups. In addition, this scalable representation should introduce a bounded approximation error regardless of the skew in the data and yield highly accurate load balancing strategy.

- *Efficient*: The process for building the representation should be accomplished in a reasonable amount of time, without incurring additional overhead. Specifically, the construction should only make one pass over the intermediate key-value pairs.

- *Robust*: The key-value pairs are generated in a streaming manner at the mappers, such that the system cannot anticipate the order of their arrival. Thus, the representation should be robust to any arrival order (i.e., changing the arrival order of the key-value pairs should not change the final representation).

- *Mergeable*: The MapReduce framework contains multiple mappers, each of which may emit a large volume of key-value pairs. It is inefficient to send all local key-value pairs to a central point to build the representation. A desirable approach is to have each mapper build its local representation, which can later be merged for

Algorithm 1 Update operation for local Count-Min sketch C^L

Require: A new coming key-value pair with key k
Require: A local sketch C^L with size $d \times w$
1: **for** $i = 1 \rightarrow d$ **do**
2: $\quad C^L[i, h_i(k)] \leftarrow C^L[i, h_i(k)] + 1$
3: **end for**

a global representation. This merged global representation should be equal to the one that is built directly on all key-value pairs.

Sketch-based Key Group Size Profiling

In this section, we introduce the notion of *sketch* [22, 21, 69] into the MapReduce framework as a data structure for summarizing key group sizes and present a distributed method for its construction. Here we use Count-Min sketch [22] in our implementation, as it provides the most efficient performance to do load balance. We also discuss and evaluate other popular sketches (i.e., FastAGMS sketch [21]) in Section III.

Local Sketch

To leverage the sketch structure for key group sizes in the MapReduce framework, we first build a local sketch using map tasks. Specifically, a local sketch C^L is a two-dimensional array of counters with d rows of length w, which are indexed by a set of pairwise independent hash functions $\mathcal{H} = \{h_i, i = 1, 2, \ldots, d\}$. Each hash function h_i maps an intermediate key k into a hashing space of size w; i.e., $h_i(k) \in \{1, 2, \ldots, w\}$.

Initially, all of the counters in the array are set to zero.

$$C^L[i, j] = 0, \text{for all } i \in \{1, 2, ..., d\}, j \in \{1, 2, ..., w\}.$$

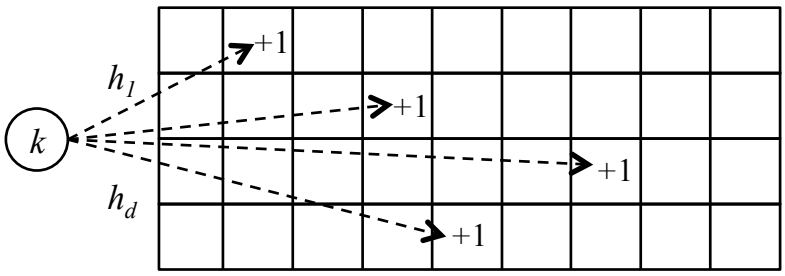


Figure III.2: An example of local Count-Min sketch update ($w = 9$ and $d = 4$).

When a new key-value pair with key k is emitted at a mapper, the local sketch at this mapper is updated according to Algorithm 1. Essentially, the arrival of this new key-value pair increments the value of d counters by 1 in the local sketch. The positions of these counters are specified by the hash functions that are associated with this sketch, where the index of the hash function i specifies the row and the hash value $h_i(k)$ of key k under function h_i specifies the column.

Figure III.2 illustrates this update process. Here, a key-value pair with key k is mapped to a counter in each row i ($i \in \{1, 2, ..., d\}$) by the hash function h_i and increments the counter by 1. A local sketch can be established in an online fashion where the mapper progressively updates the sketch because it processes the input records and emits intermediate key-value pairs.

Let S_k represent the number of key-value pairs with key k emitted at a mapper. Then, its constructed local sketch C^L can be represented as follows.

$$C^L[i, j] = \sum_{\forall k: h_i(k)=j} S_k.$$

In this representation, counter $C^L[i, j]$ holds the sum of the sizes from all key groups whose keys k are mapped by hash function h_i to value j.

Since each hash function can be evaluated in constant time, an update to the sketch requires time $O(d)$ for each intermediate key-value pair emitted at the mapper.

Global Sketch

After the map phase is complete and the local sketches are constructed, the global sketch can be constructed from each mapper's local sketch to summarize the global key group sizes. This task can be performed at a central central point, such as a reducer in our implementation. Specifically, to build a global sketch, the following steps need to be performed:

- *Local sketch preparation at mappers*. Each mapper maintains a local sketch C^L as discussed in Section III. To be mergeable, each local sketch agrees to the same configuration, including the values of d and w and the definition of hash functions $\mathcal{H}$.

- *Communication*. Each mapper sends C^L to the single reducer.

- *Global sketch construction*. The reducer aggregates all of the local sketches to compose a global sketch. Let M be the number of mappers. Given local sketches C_1^L, ..., C_M^L of size $d \times w$, the aggregated global sketch C^G is also a two-dimensional array of size $d \times w$, where each entry $C^G[i, j]$ sums the corresponding local sketch entries.

$$C^G[i, j] = \sum_{l=1}^{M} C_l^L[i, j], \text{for all } i \in \{1, ..., d\}, j \in \{1, ..., w\}.$$

  We denote the aggregation operation as $C^G = \sum_{l=1}^{M} C_l^L$.

Properties of Sketch-based Profiling

This sketch-based key group size summary fits well in the MapReduce framework and offers the following advantages.

- *Scalable* and *efficient*. The memory overhead is $O(dw)$ at both the mappers and the reducer. Additionally, the communication overhead is bounded by $O(dwM)$. The *update* operation in Algorithm 1 requires $O(d)$ time for each data element (i.e., an intermediate key-value pair) at each mapper. The *aggregate* operation overhead at the reducer is bounded by $O(dwM)$. All overhead are independent of the number of key groups.

- *Robust* to the data arrival sequence. Given a set of key-value pairs, the sketch is invariant to the order of the data. This property directly stems from the facts that i) each sketch is a linear projection of the original data and ii) the *update* operation of one data item is independent from others.

- *Mergeable*. Given a set of key-value pairs, the global sketch can be established by the aggregation of local sketches that are constructed from mutually exclusive and collectively exhaustive subsets. This property is formally specified in the following proposition.

  Proposition 1 *Let $\mathcal{P}$ be the set of key-value pairs and $C(\mathcal{P})$ be its sketch. Consider any partition of set $\{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_M\}$ where $\bigcup_{l=1}^{M} \mathcal{P}_l = \mathcal{P}$ and $\forall l \neq h, \mathcal{P}_l \bigcap \mathcal{P}_h = \emptyset$. Let $C(\mathcal{P}_l)$ be the sketch of $\mathcal{P}_l$. Then $C(\mathcal{P}) = \sum_{l=1}^{M} C(\mathcal{P}_l)$.*

The latter two properties are a significant contrast from the best previously known approach for estimating key group sizes; i.e., TopCluster [33]. In TopCluster, the final histogram estimation of the key group sizes is influenced by both (1) the order and (2) the number of local histograms at the mappers.

The probability property of the sketch structure provides a mathematical guarantee for accurate estimation on key group sizes and has been widely adopted for data stream

processing. In Count-Min sketch [22], an estimation for the group size of key k is given by

$$\hat{S}_k = \min_{1 \le i \le d} C^G[i, h_i(k)].$$

The key group size estimation error bound is given in the following lemma. The proof of this lemma follows directly from the results in [22].

Lemma 1 *The size estimate $\hat{S}_k$ of key group k has the following guarantees: $S_k \le \hat{S}_k$, and with probability at least $1 - \delta$, $\hat{S}_k \le S_k + \varepsilon S$. Here $\delta = 1/e^d$, $\varepsilon = e/w$, and S is the total number of key-value pairs.*

However, this key group size estimation method cannot be directly applied to design the key-group-to-reducer assignment algorithm. In Count-Min sketch, the key must be supplied to obtain the group size estimation. Since the key information is lost in the aggregation of the sketch, we either need to record it in a separate data structure (which introduces additional storage overhead) or make another pass of the input data, which eliminates the benefit of one-pass profiling. In the next section, we investigate a load balancing mechanism that directly operates on the sketch structure.

Sketch-based Load Balancing Algorithm

In this section, we investigate the design of the partition function, which maps the intermediate key to a reducer index, based on the key group size information as summarized in the global sketch. Formally, let $\mathcal{K}$ be the key space and R be the number of reducers. A partition function $\Phi : \mathcal{K} \to \{1, 2, ..., R\}$ maps key k to the index of the desired reducer $r = \Phi(k) \in \{1, 2, ..., R\}$. We assume the reducer load is proportional

to its input key group sizes. As such, the load at reducer r can easily be derived as $S(r) = \sum_{k:\Phi(k)=r} S_k$, where S_k is the size of key group k.

By designing the partition function (Φ) in this manner, we aim to to balance the load at different reducers (i.e., to minimize the maximum reducer load). Formally, the objective can be stated as follows.

$$\min_{\Phi} \max_{1 \leq r \leq R} S(r).$$

The performance of a load balancing algorithm can be evaluated using the reduce-phase load imbalance ratio σ, which is the ratio between the maximum reducer load and the average reducer load. Formally, σ is defined as follows.

$$\sigma = \frac{\max\limits_{1 \leq r \leq R} S(r)}{\sum\limits_{r=1}^{R} S(r)/R}.$$

Optimal Sketch Packing Algorithm

To address this objective, we introduce an *optimal sketch packing* algorithm. This algorithm works directly on the global sketch without the need for knowledge of the intermediate keys. As a result, the algorithm is scalable to a large number of key groups.

As shown in Figure III.3, the basic operation in the *optimal sketch packing* algorithm is very similar to the key group packing algorithm. Instead of working over the key groups, the *optimal sketch packing* algorithm works over the counters in each row in the sketch. As discussed before, the sketch contains d rows, while each row contains w counters. Each row can be treated as a linear projection of the original key groups. Since w is very small compared to the data space $|\mathcal{D}|$, the *optimal sketch packing* algorithm can utilize the key group packing algorithm (Algorithm 2) directly. The *optimal sketch*

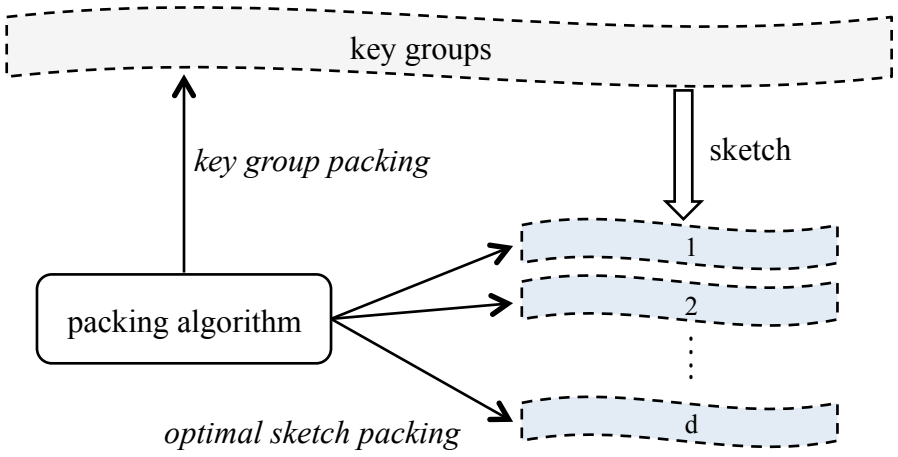


Figure III.3: The *optimal sketch packing* algorithm.

Algorithm 2 Key Group Packing Algorithm

Require: Key group sizes $S_k, k \in \mathcal{K}$
Require: Number of reducers R
1: sort($S_k, k \in \mathcal{K}$)
2: **for all** $k \in \mathcal{K}$ **do**
3: $r \leftarrow$ selectLeastLoadedReducer()
4: $S(r) \leftarrow S(r) + S_k$
5: $\Phi(k) \leftarrow r$
6: **end for**
7: **return** Φ

packing repeats this process for each row and chooses the one with the minimum reduce-phase imbalance ratio.

Algorithm 3 presents the details of the *optimal sketch packing* algorithm. In this algorithm, σ_{min} records the current minimum reduce-phase imbalance ratio. It is initialized to R, which is the maximum reduce-phase imbalance ratio. The Φ_{min} variable records the sketch-cell-to-reducer mappings in as the current best solution.

As shown in Figure III.3, the basic operation in the *optimal sketch packing* algorithm is very similar to the key group packing algorithm. As illustrated in Algorithm 2, the key group packing algorithm adopts bin packing mechanism [20] and works on key groups

Algorithm 3 Optimal Sketch Packing Algorithm

Require: Global sketch C with size $d \times w$
Require: Number of reducers R
1: $\sigma_{min} \leftarrow R$
2: $\Phi_{min} \leftarrow$ NULL
3: **for** $i = 1 \rightarrow d$ **do**
4: `// Use key group packing algorithm to pack each row`
5: $\Phi \leftarrow$ keyGroupPacking($C[i][1...w]$, R)
6: `// Calculate imbalance ratio and update min`
7: $\sigma \leftarrow$calculateImbalanceRatio(Φ, C)
8: **if** $\sigma < \sigma_{min}$ **then**
9: $\sigma_{min} \leftarrow \sigma$
10: $\Phi_{min} \leftarrow \Phi$
11: **end if**
12: **end for**

directly. As discussed above, maintaining all key group sizes is impractical in reality, which makes key group packing algorithm difficult to deploy.

The sketch contains d rows, while each row contains w counters. Each row can be treated as a linear projection of the original key groups. Since w is very small compared to the data space $|\mathcal{D}|$, we can deploy packing algorithm on each row in the sketch. Thus, we propose an *optimal sketch packing* algorithm, which repeats the packing operation for each row and chooses the one with the best load balancing performance (the one with the minimum reduce-phase imbalance ratio σ).

Algorithm 3 presents the details of the *optimal sketch packing* algorithm. In this algorithm, σ_{min} records the current minimum reduce-phase imbalance ratio. It is initialized to R, which is the maximum reduce-phase imbalance ratio. The Φ_{min} variable records the sketch-cell-to-reducer mappings in as the current best solution.

Performance Analysis

Here we analyze the memory, communication and computational complexities of our *optimal sketch packing* algorithm, and the load balancing performance in terms of the reduce-phase imbalance ratio.

Proposition 2 *The memory complexity of the* optimal sketch packing *algorithm is* $O(dw)$*, the communication cost is* $O(mdw)$*, and the time complexity is* $O(dw\ log(w) + dRw)$*. Here the* d *and* w *represent the sketch depth and width,* m *is the number of mappers and* R *is the number of reducers.*

Proof 1 *Each map task maintains a local sketch, which takes* $O(dw)$ *memory. The global sketch is also a* $d \times w$ *array, and the memory cost is also* $O(dw)$*. Thus, the memory complexity of the* optimal sketch packing *algorithm is* $O(dw)$*.*

For communication, each map task sends its local sketch to a central controller. The sketch size is $d \times w$*, thus the communication cost is* $O(mdw)$ *for* m *map tasks.*

For the local sketch update, each update process involves with d *counters, the time complexity for update is* $O(d)$*.*

Every time the optimal sketch packing *algorithm sends one row in the sketch to the* key group packing *algorithm to perform pack operation. The time complexity for packing one row is* $O(w\ log(w) + Rw)$*. This process is repeated a total of* d *times; thus, the total time complexity for the* optimal sketch packing *algorithm is* $O(dw\ log(w) + dRw)$*.*

Theorem 1 *The optimal sketch packing algorithm is a* $(2 + \frac{eR}{w})$*-approximation algorithm, with probability at least* $1 - \delta$*.*

Proof 2 *Once the* optimal sketch packing *algorithm assigns the counter for an arbitrary row* i *in the sketch, each reducer will receive several counters. Let us say that reducer* s

received the maximum workload and counter $C[i,j]$ is the last one it was assigned. Let L_{st} be the workload of reducer s before it received $C[i,j]$. When counter $C[i,j]$ was assigned, the workload of its reducer was no larger than other reducers, so every reducer at that time had a workload larger than L_{st}. Thus, the maximum reducer workload is:

$$\begin{aligned}\mathsf{MAX}_{app} &= L_{st} + C[i,j] \\ &\leq \frac{N - C[i,j]}{R} + C[i,j] \\ &= \frac{S}{R} + (1 - \frac{1}{R})C[i,j].\end{aligned}$$

Now, if we let L_{max} be the workload of the maximum key group, it can be seen that $C[i,j] \leq L_{max} + \varepsilon S$, with probability at least $1 - \delta$. As such, it follows that:

$$\begin{aligned}\mathsf{MAX}_{app} &\leq \frac{S}{R} + (1 - \frac{1}{R})(L_{max} + \varepsilon S) \\ &\leq \mathsf{MAX}_{opt} + (1 - \frac{1}{R})(\mathsf{MAX}_{opt} + \varepsilon R \mathsf{MAX}_{opt}).\end{aligned}$$

From which, it can then be deduced that:

$$\mathsf{MAX}_{app} \leq (1 + (1 - \frac{1}{R})(1 + \varepsilon R))\mathsf{MAX}_{opt}.$$

Therefore, the load balancing performance ratio is:

$$\rho = 1 + (1 - \frac{1}{R})(1 + \varepsilon R) \leq 2 + \varepsilon R = 2 + \frac{e \times R}{w}.$$

Implementation with Hadoop

We implement our profiling and load balancing solutions in Apache Hadoop [64]. The overall implementation involves two MapReduce jobs, as shown in Figure III.4. An additional *profiling* MapReduce job needs to be executed before the *real* job, to sample the data, build the sketch and compute the sketch-cell-to-reducer assignment.

A block-level sampling technique is introduced to reduce the extra running overhead brought by the *profiling* job. In Hadoop, input data is stored in Hadoop File System (HDFS) as equal-sized data blocks (64 MB in default). The block-level sampling technique randomly selects a small percent of data blocks to build the sketch. In comparison to the record-level sampling technique, the block-level sampling has a better running efficiency with similar accuracy [50]. As shown in our experimental analysis, a 5% block-level sampling rate delivers good load balancing performance for all applications and datasets evaluated in this study.

In the *profiling* job, each mapper processes the sampled input data blocks and builds its local sketch. All local sketches are sent to one reducer, where the global sketch is generated. Our *optimal sketch packing* algorithm is also deployed at this reducer. The output of the *profiling* job is a sketch-cell-to-reducer mapping. Along with the set of hash functions that are used to build the sketch, this mapping will be used as the new partition function for the *real* job to direct the shuffle (i.e., reducer assignment) phase.

Experimental Evaluation

Experiment Setup

Applications and Dataset. We investigate two real world applications to assess the performance of MapReduce jobs using our load balancing mechanism: *PageRank* [12]

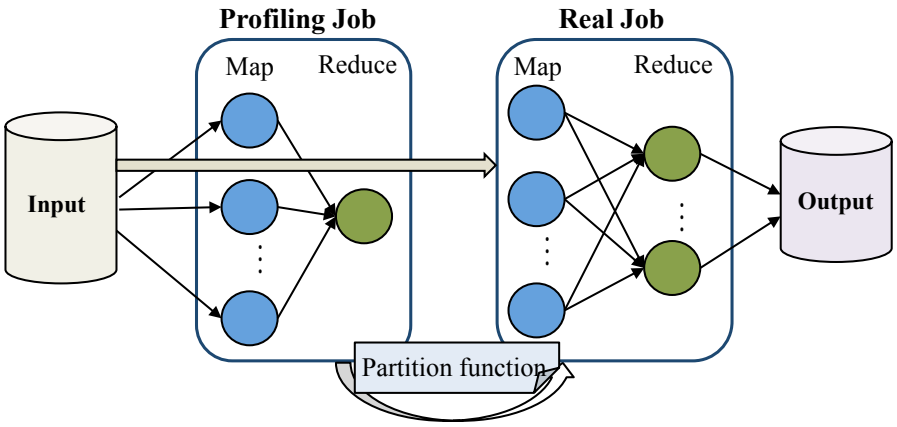


Figure III.4: Implementation of the *optimal sketch packing* algorithm with Hadoop.

and *Inverted Indexing* [51]. We evaluate *PageRank* using three real datasets: Flickr[1], YouTube[2] and Twitter. Each dataset represents a directed or undirected graph, where the rank of each vertex is computed via *PageRank* algorithm. We also use three synthetic datasets, whose data size distribution follows a Zipf distribution[3] to evaluate the impact of skew on the algorithm performance. *Inverted Indexing* is evaluated using two additional datasets: DBLP[4] and Wikipedia[5]. Each dataset contains several publications/documents, where an inverted index is built for each word. Table III.1 provides summary statistics of the real datasets.

Running Environments. We conduct experiments in both simulated and real MapReduce cluster environments. In the simulated environment, we evaluate the accuracy and the performance of the sketch-based solution in estimating the key group size and achieving reducer-phase load balancing. The three datasets with relatively small sizes (i.e., DBLP, Flickr, and YouTube) are evaluated in this environment.

[1]http://snap.stanford.edu/data/web-flickr.html
[2]http://snap.stanford.edu/data/com-Youtube.html
[3]For a given Zipf distribution with parameter z, the number of records in the k^{th} key group is proportional to k^{-z}.
[4]http://dblp.uni-trier.de/xml
[5]http://dumps.wikimedia.org/enwiki/

Table III.1: Summary for experimental datasets used in Chapter III

Application	Dataset	# of key groups	# of key-value pairs
PageRank	Flickr	105,938	4,633,896
	Youtube	1,134,890	5,975,248
	Twitter	40,104,238	1,539,743,478
Inverted Indexing	DBLP	482.810	11.127.479
	Wikipedia	14,180,286	1,487,606,462

We use the Amazon Elastic MapReduce service[6] for the experiments over the two massive datasets, Twitter and Wikipedia. In this environment, 20 m1-medium instances are used with a separate master instance. Each instance has 1 vCPU and 2 ECU with 4 GB of memory and 410 GB disk. The HDFS block size is set to 64 MB and each instance is configured to run at most two map tasks and two reduce tasks concurrently. We disable the speculative task execution feature to better analyze the running time of each task. By default, each MapReduce job is configured with 40 reducers.

Baseline Algorithms. We compare our *optimal sketch packing* (SP) algorithm with the following approaches.

- The Hadoop default (HD) algorithm, which uses the hashing partition function for key group assignment.
- The key group packing (KP) algorithm (Algorithm 2), which performs packing algorithm on all key groups with accurate sizes.
- The state-of-the-art *TopCluster* (TC) algorithm, which builds a histogram to track the k most frequent key groups.
- The sampling-based (Sample) solution. This sampling approach is different from the block-level sampling we discussed in Section III. In this approach, we sample

[6]http://aws.amazon.com/elasticmapreduce/.

the intermediate key-value pairs generated by map tasks, and maintains a sampled key group sizes in memory. After that, we deploy key group packing algorithm on the sampled key groups. For the key groups not contained in the samples, we use the default hash function as the partition function. As already known, a simple sample rate cannot ensure the memory cost. To be fair, for each group of experiment, we try several different sample rates to discover the one with the same memory cost as TC and SP.

Note that we only evaluate the key group packing algorithm in our simulated environment, where the input datasets are small. For the two massive datasets, it is not pragmatic to hold the accurate sizes for all key groups in memory, and the program will throw *OutOfMemory* exception.

Sketch and Profile Settings. In most experiments, the sketch structure is set with size w = 1000 and d = 5, which requires less than 1 MB memory. For *TopCluster* algorithm, we configure it with k = 5000, which leads to a similar memory usage for fair comparison. For both algorithms, we adopt a block-level sampling strategy that samples the HDFS blocks at a 5% sampling rate.

Simulated Environment

Key Group Size Estimation

Though key group sizes are not directly used for load balancing, they serve as a basis of our load balancing algorithm. This allows us to derive important insight into the performance of the load balancing algorithm. Thus, we first evaluate the performance of sketch in terms of its key group size estimation.

These experiments use four real datasets to assess the extent to which the techniques can estimate key group sizes. In Figure III.5, we rank key groups according to their real

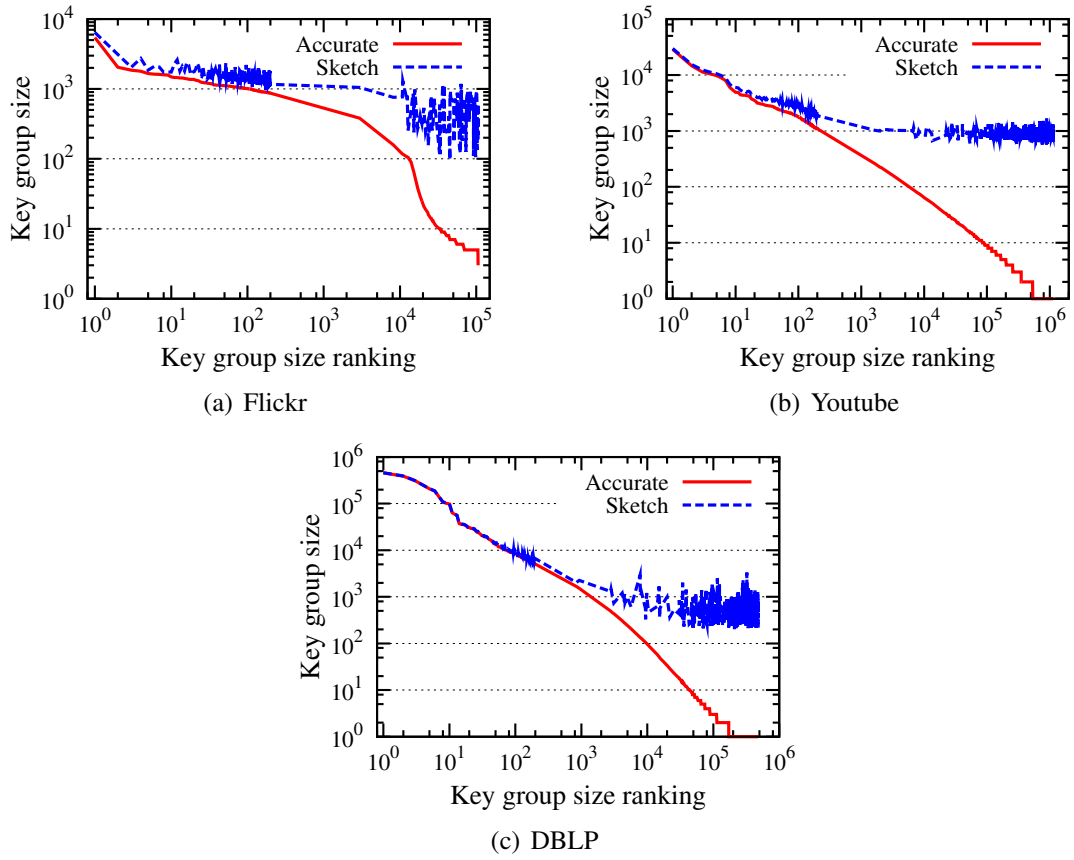
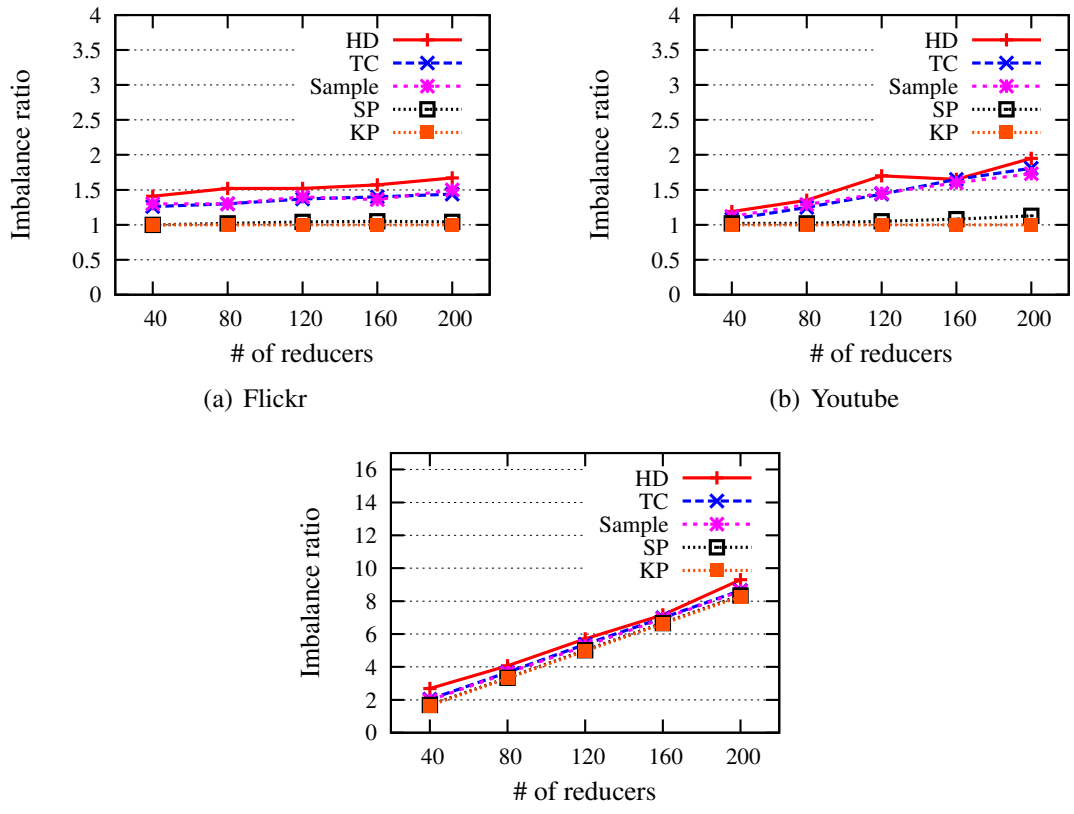


Figure III.5: Key group size estimation for various datasets.

size (*Real*) and mark the estimated size using Count-Min sketch as *Sketch*. As shown in Figure III.5, sketch provides an accurate estimation for highly ranked key groups (which have large sizes). And, the performance of the sketch-based method is directly correlated with the degree of data skew [69]. Of all the datasets, sketch achieves the best performance on the DBLP dataset (shown in Figure III.5(c)). This is in line with the observation that DBLP has the most skewed distribution of key group sizes.

Reduce-Phase Load Balancing

We compare the performance of our SP algorithm with HD, TC, Sample and KP. This set of experiments is deployed with three real datasets and three synthetic datasets.



(a) Flickr (b) Youtube

(c) DBLP

Figure III.6: Reduce-phase imbalance ratio for three small datasets.

Each synthetic dataset is composed of 50 million key-value pairs, organized in 5 million groups. We vary the Zipf parameter $z \in \{0.6, 0.8, 1.0\}$. To evaluate the algorithm performance with different scenarios, we vary the number of reducers from 40 to 200. For each group of experiments, we record the number of intermediate key-value pairs assigned to each reducer as a representation of the reducer workload. Then, we calculate the reduce-phase imbalance ratio.

Figure III.6 shows the results with the three real datasets. Since HD utilizes a hash function as its default setting, its performance largely depends on the input data and cannot provide any assurance over the reducer load. The TC algorithm can provide some assurance, but it only tracks the top-k key groups, as such, its performance is also

limited. The Sample approach only has limited information about the key group sizes, and most key groups cannot be sampled by given a limited space. Thus its performance is also limited. Our SP algorithm requires much less memory than KP while delivering competitive load balancing. In all cases, SP significantly outperforms HD, TC and Sample.

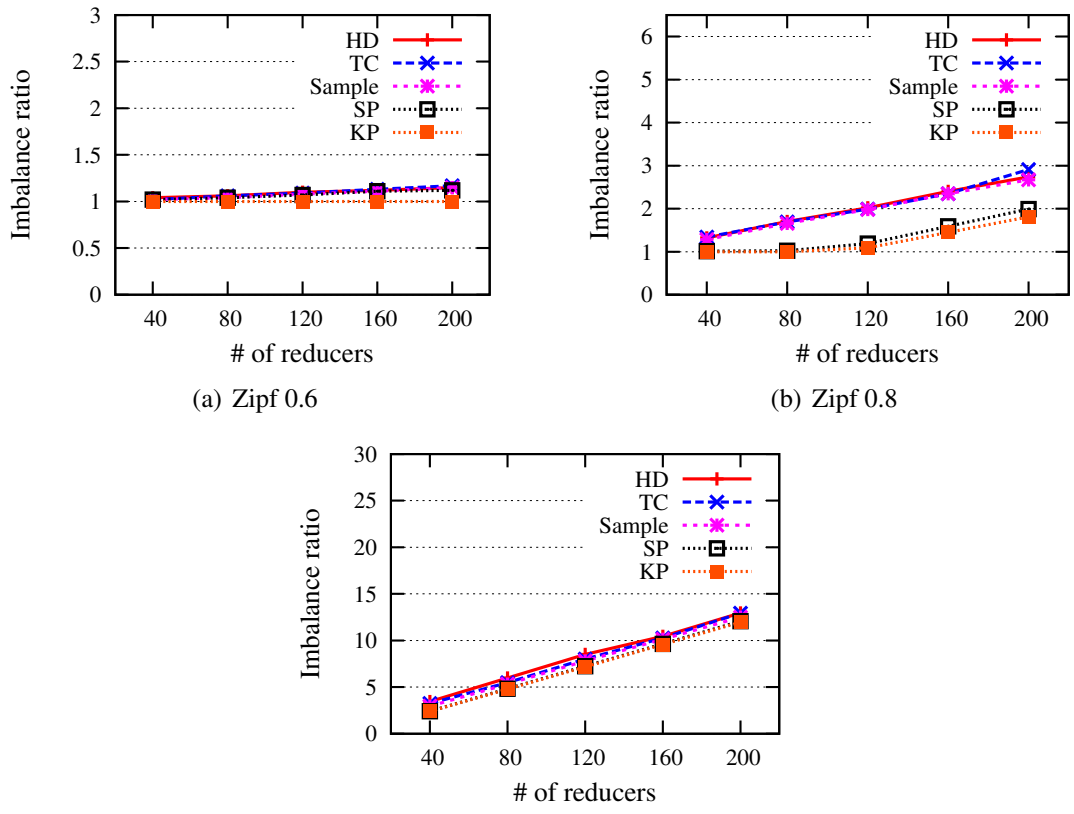
In Figures III.6(a), III.6(b), our SP algorithm maintains an imbalance ratio close to 1.0. In Figure III.6(c), the imbalance ratio is always much larger than 1. This is because the key group size distribution is highly skewed and there exists an *expensive key group* whose workload exceeds the average. Whereas we assign this *expensive key group*, the assigned reducer would become a straggler.

Figure III.7 presents the results of synthetic datasets, which further validate the superiority of SP. In Figures III.7(a), where the input dataset has low skew, all four algorithms exhibit similar performance. In Figure III.7(b), where the dataset has medium skew, SP performs much better than HD, TC and Sample. In Figure III.7(c), where the input dataset has high skew and a particular key group dominates most of the workload, none of the five algorithms can mitigate the reduce-phase skew.

Generally, our SP algorithm provides much better load balancing compared to the default approach of Hadoop, the state-of-the-art TC algorithm and the Sample approach. Furthermore, the SP algorithm can achieve a load balancing performance that is close to the KP algorithm in most situations.

Robust to Data Sequence

As discussed earlier, the distributed streaming data model dictates that robustness is a core requirement for composing an appropriate MapReduce data representation. In this section, we perform several experiments to evaluate our SP algorithm's performance under different data arriving sequences and compare them to the TC algorithm.



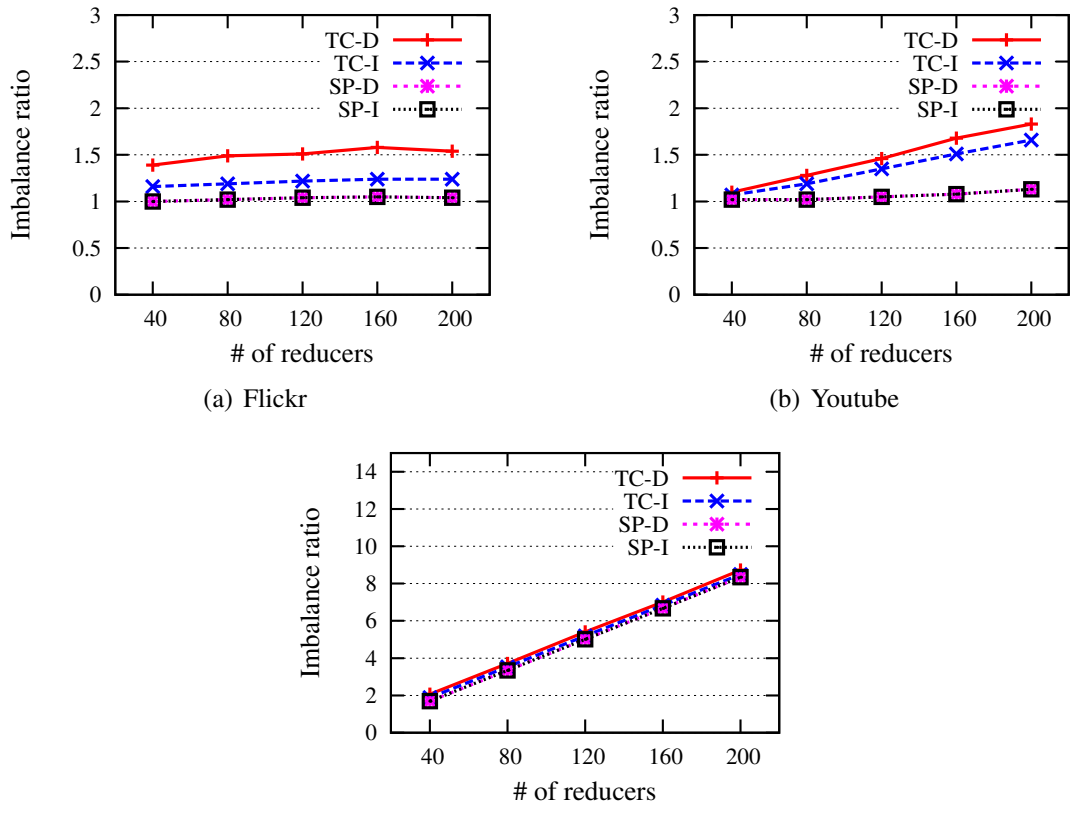
(a) Zipf 0.6

(b) Zipf 0.8

(c) Zipf 1.0

Figure III.7: Reduce-phase imbalance ratio for three Zipf datasets.

We generate two types of datasets from the original Flickr, Youtube and DBLP datasets. We sort key groups according to their sizes either in an increasing order (I) or a decreasing order (D), which generate two new data streams for each dataset. We deploy our SP algorithm and the TC algorithm over those data streams and examine their reduce-phase workload imbalance ratios. The results are presented in Figure III.8. As shown in the figure, our SP algorithm is robust to data arrival sequence and show stable performance. On the other hand, the TC algorithm is highly sensitive to the data arrival order. For Figure III.8(c), as DBLP dataset contains expensive key groups, there is not obvious performance difference between different algorithms.



(a) Flickr (b) Youtube

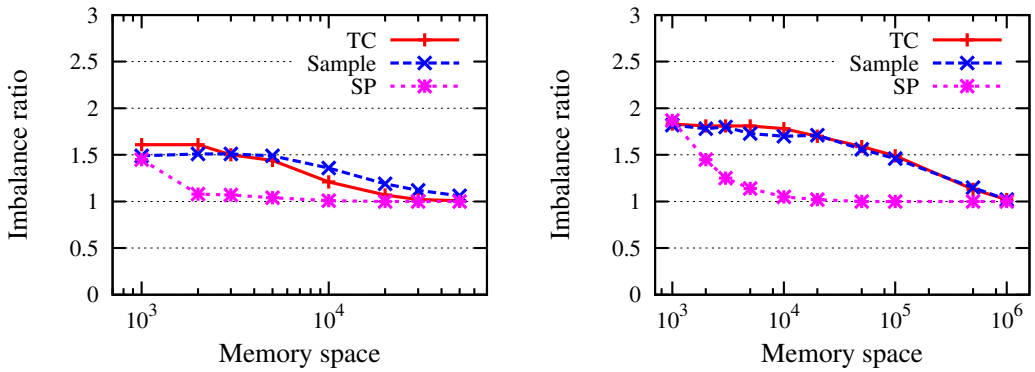
(c) DBLP

Figure III.8: Reduce-phase imbalance ratio with different data arrival sequences.

An interesting observation is that the TC algorithm performs much better with increasing ordered data than decreasing order. This is because that the TC algorithm utilizes a stream summary algorithm [57] to track the top-k key groups. The stream summary algorithm works better with increasing ordered data. We refer readers to [57] for more implementation details.

Impact of Storage Space

As approximate approaches, the performance of the TC, Sample and SP algorithms depends on the given memory space. Here we investigate the impact of memory spaces on the reduce-phase imbalance ratio.



(a) Flickr (b) Youtube

Figure III.9: Reduce-phase imbalance ratio with different memory spaces.

Figure III.9(a) and III.9(b) illustrate the results for Flickr and Youtube datasets. The results shows that the SP algorithm can perform much better with an increasing memory space.

Comparison of Different Sketches

In our implementation, we select Count-Min sketch [22] as the data summary. There are other types of sketch implementations, such as the FastAGMS sketch [21]. Here we compare these two different sketches' performance.

The major difference between the Count-Min and FastAGMS sketches is how the values are updated. The Count-Min sketch always does +1 for each update (as shown in Figure III.2), while the FastAGMS sketch does +1 or -1 (depends on another hash function).

Figure III.10 illustrates the reduce-phase imbalance ratio with three different datasets and 200 reducers. The results show that FastAGMS sketch cannot achieve reduce-phase load balancing and the imbalance ratios are very high. This is because the FastAGMS sketch cannot maintain the key group workload information. A combination of +1 and -1 generates 0 workload, although it is 2 key-value pairs.

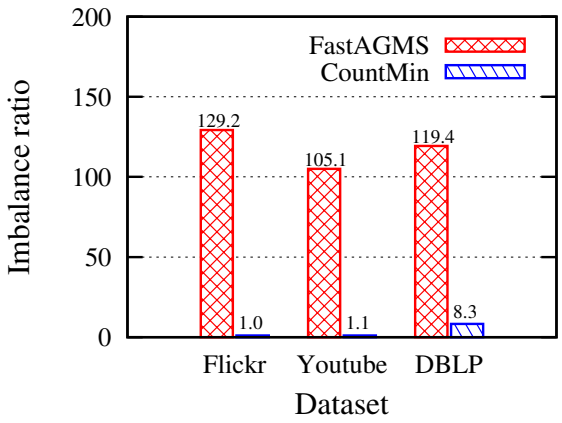


Figure III.10: Reduce-phase imbalance ratio with different sketches.

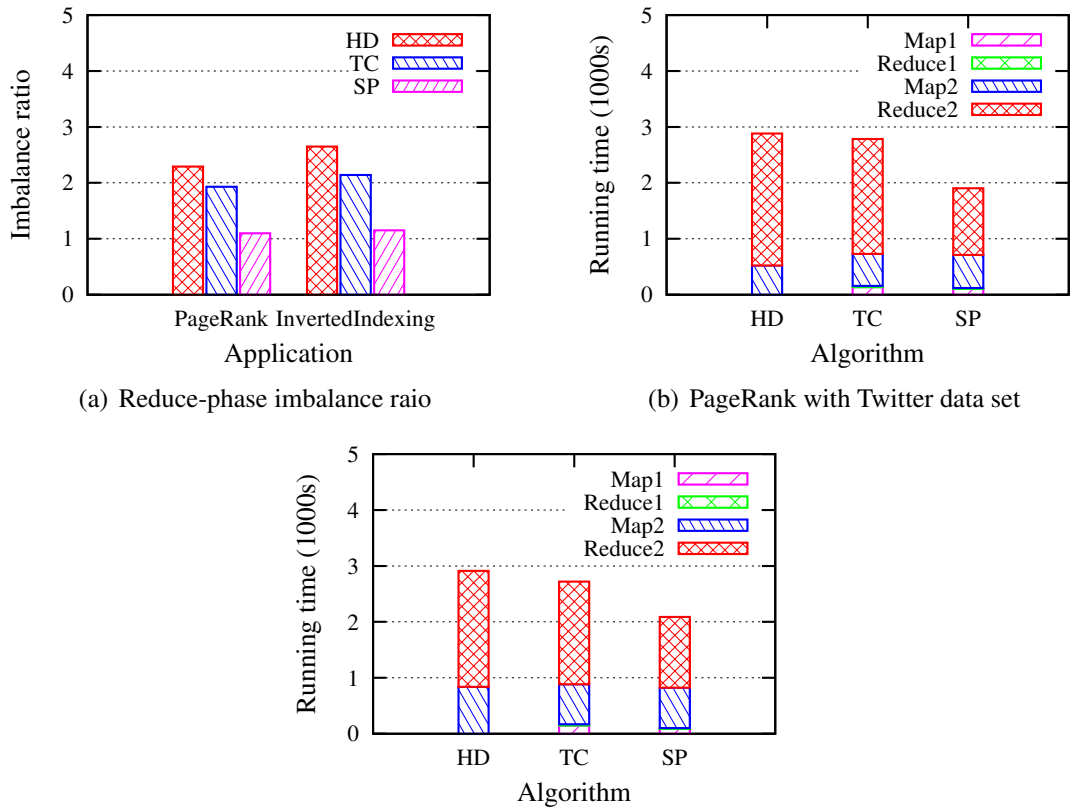
Amazon Elastic MapReduce Environment

To perform experiments with massive-sized datasets and evaluate the job running time, we use the Amazon Elastic MapReduce Service.

Overall Job Running Time

In this section, we compare our SP algorithm with the HD and TC algorithms using two groups of experiments: 1) *PageRank* over the Twitter dataset and 2) *Inverted Indexing* over the Wikipedia dataset. In comparison to HD, both SP and TC introduce another MapReduce job (*profiling*) to build a data profile. As such, their overall job running time also includes the time spent in the *profiling* job. For each group of experiments, we record the running time of the two MapReduce jobs (i.e., *profiling* and *real*) and calculate the reduce-phase imbalance ratio of the *real* job.

Figure III.11(a) summarizes the reduce-phase imbalance ratio. The results further validate that our SP algorithm outperforms both HD and TC algorithms, especially when facing millions or billions of key groups. The TC algorithm operates packing operation over the top-k key groups, as such, it can improve the default HD algorithm. However, for applications having millions of key groups, balancing only the top-k key groups is



(a) Reduce-phase imbalance raio

(b) PageRank with Twitter data set

(c) Inverted Indexing with Wikipedia data set

Figure III.11: Job running time and reduce-phase imbalance ratio with *PageRank* and *Inverted Indexing* applications.

insufficient, and the reduce-phase imbalance ratio is still very high. Our SP algorithm summarizes all key group sizes in the sketch, which largely keeps the original key group workload information. This mechanism helps the SP algorithm largely balance the reduce workload for both applications, and its reduce-phase imbalance ratio is close to 1.0.

Figure III.11(b) and Figure III.11(c) illustrates the entire job running time for *PageRank* and *Inverted Indexing* applications. Here *Map1* and *Reduce1* refer to the *profiling* job, and *Map2* and *Reduce2* refer to the *real* job. As our SP algorithm can balance the reduce-phase workload, it can largely reduce the job running time. In comparison

to the HD algorithm, the SP algorithm can reduce the job running time by 51.4% and 39.6%. Additionally, the *profiling* job of both the TC and SP algorithms incur only a small additional cost (5.7% in average) in comparison to the whole job running time.

Impact on Number of Reducers

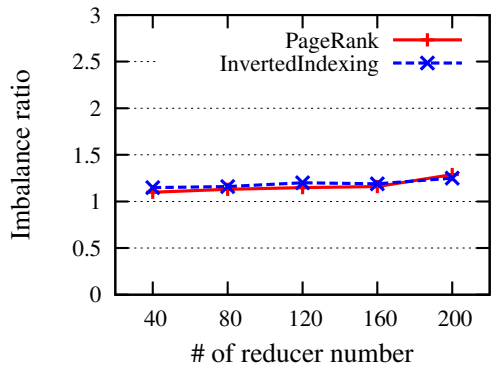
In a production cluster, different MapReduce jobs may have different number of reducers, according to their workload and cluster resources. In this section, we evaluate the impact of number of reducers on the performance of the SP algorithm.

Figure III.12(a) illustrates the reduce-phase imbalance ratio when the number of reducers varies from 40 to 200. The results show that, for both *PageRank* and *Inverted Indexing* applications, our SP algorithm delivers stable performance with an imbalance ratio close to 1.

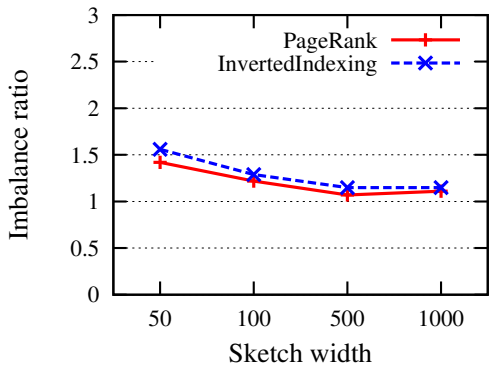
Impact of Sketch Size

According to the analysis in Section III, the performance of the SP algorithm depends on R and w, where R is the number of reducers and w is the sketch width. Given a fixed R, we evaluate the impact of the sketch width w.

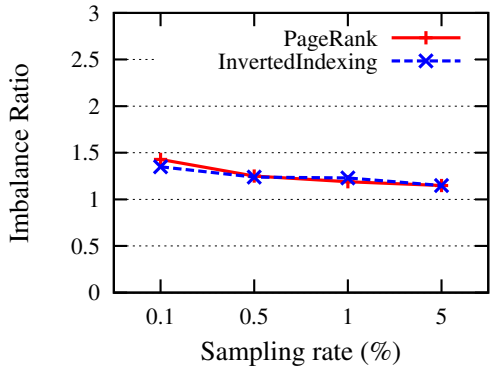
Here we fix $R = 40$ and $d = 5$, and vary w from 50 to 1000. Figure III.12(b) shows that the imbalance ratio is inversely correlated with the sketch width. This phenomenon is also reflected in our theoretical analysis – when the number of reducers is close to the sketch width, the benefit of our SP algorithm will diminish. Our empirical study shows that, when the sketch width is set to 10 to 20 times that of the number of reducers, it leads to the best performance.



(a) Various reducer numbers (sketch size is 1000*5 and sampling rate is 5%)



(b) Various sketch sizes (reducer number is 40 and sampling rate is 5%)



(c) Various sampling rates (sketch size is 1000*5 and sampling rate is 5%)

Figure III.12: Reduce-phase imbalance ratio under various settings.

Impact of Sampling Rate

Block-level sampling is introduced to reduce the running time of *profiling* job, as discussed in Section III. Here, we investigate the impact of the sampling rate. Figure III.12(c) illustrates the reduce-phase imbalance ratio under different block-level sampling rates. As the sampling rate increases, the sketch is more adept at capturing the real data distribution. As such, our SP algorithm performs much better. The empirical study also shows that a 5% block-level sampling rate is sufficient to deliver good load balancing performance.

Comparison with Online Approach

Our implementation in Section III introduces an additional *profiling* MapReduce job to build the sketch. We name this approach as an "offline" approach. Another alternative is to use an "online" approach. Here we briefly describe the operation of the online approach and validate our choice of an offline approach using empirical study.

The online approach does not need a *profiling* job. By changing the implementation inside Hadoop, the online approach lets each map task in the *real* job build its local sketch and send the local sketch to the Hadoop central controller (i.e., *JobTracker* in Hadoop). After the map phase completes, the central controller runs the SP algorithm and sends the generated sketch-cell-to-reducer mappings back to each mapper, where it is used to dispatch the key groups to corresponding reducers. Instead of introducing a *profiling* job, this online approach introduces a profiling phase between map phase and reduce phase to do the profiling job.

To investigate the efficiency of these two mechanisms, we conduct a comparison study between them. As the implementation of the online approach requires the modification of the Hadoop source code, which is not supported by the Amazon Elastic MapReduce service, we implement the online approach in a private cloud environment[7] with similar configurations.

We calculate the ratio of the *profiling* running time compared to the whole job running time. For the offline approach, the *profiling* job takes 5.2% and 6.3% of total running time for *PageRank* and *Inverted Indexing*, respectively. For the online approach, these ratios increase to 21.8% and 23.3%. The online *profiling* approach nearly doubles the running time of the map phase.

[7] ISISCloud with https://cloud.isis.vanderbilt.edu/horizon/.

The reason is that, in the online profiling approach, all the mapper outputs are stored at local disks during profiling and sketch construction period. Up on receiving the sketch-cell-to-reducer mappings, each mapper reorganizes its local mapper outputs. According to the Hadoop design, each mappers output is organized into several files, and each file contains all the intermediate data sent to one reducer. This mechanism introduces another round of disk I/O operation, thus increasing cost when processing massive datasets. An alternative is to store all mapper outputs in memory, until receiving the sketch-cell-to-reducer mappings from the central controller. However, this alternative is impossible when we process massive datasets where each mapper has several GB outputs.

Chapter Summary

In this chapter, we introduced a scalable and robust load balancing solution for MapReduce framework based on a novel sketch-based data structure. We implemented and integrated our solution within Hadoop. Experimental studies using *PageRank* and *Inverted Indexing* over real and synthetic datasets show that our solution outperforms both Hadoop default implementation and the current state-of-the-art solution.

CHAPTER IV

SCALABLE LOAD BALANCING FOR MAPREDUCE-BASED RECORD LINKAGE

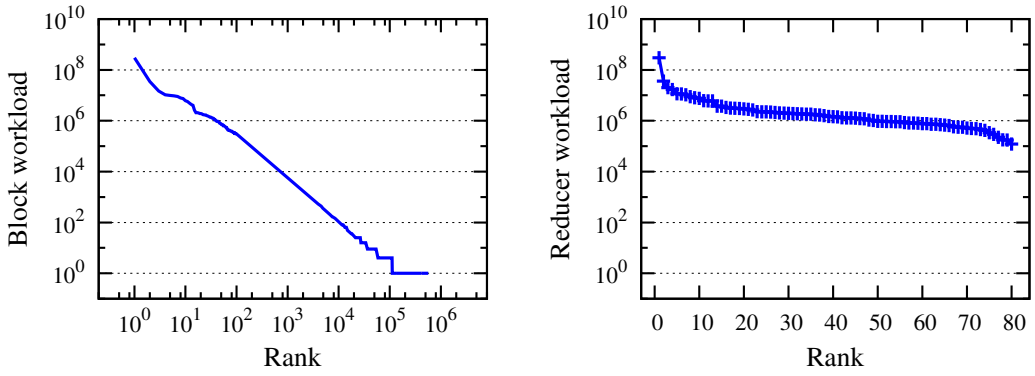
In Chapter III, we illustrate how to utilize sketch-based profiling approaches to solve the reduce-phase skew problem in a scalable way. In this chapter, we further investigate how to deploy sketch-based profiling to solve the skew with more complex applications, such as record linkage [58].

This chapter is organized as follows. We first motivate the need for solving skew problem in MapReduce-based record linkage and analyze the limitations of existing approaches in IV.1. Then we present our approach in Section IV.2, with an experimental evaluation in Section IV.3. We summarize the chapter in Section IV.4.

Motivation

The integration of data from multiple sources (i.e., *record linkage*[1]), where duplicates are merged or removed, is critical to ensure big data repositories are managed efficiently and effectively. Traditional single-machine architectures for record linkage hit significant performance barriers when applied to big data, leading to extremely long running time and high resource consumption [45]. Recently, several techniques [44, 60, 85] have been proposed to parallelize the record linkage process based on the MapReduce platform [25]. In these techniques, the datasets are partitioned into several blocks using *blocking keys* by the *map* tasks and assigned to parallel *reduce* tasks, where the record pairs are constructed for comparison.

[1] In the literature, record linkage is also referred to as deduplication, entity resolution, merge-purge and name matching [29]



(a) Block workload (b) Reducer workload

Figure IV.1: Workload ranking for the DBLP-1 dataset.

A common issue faced by these approaches is reduce-phase *data skew*, which occurs when the block workload is non-uniformly distributed. Figure IV.1(a) illustrates an example of block workload[2] ranking for the DBLP-1 dataset (We refer readers to Table IV.1 for details regarding this dataset). When such data skew occurs, the reducer that receives the highest workload requires a significantly longer amount of time to complete its task compared to its peers. As a result, an overall prolonged reduce-phase running time is introduced, which diminishes the benefits realized through parallelization. Figure IV.1(b) shows an example of the reducer workload ranking when we deploy the DBLP-1 dataset in MapReduce using 80 reducers. Here the most loaded reducer takes 44x more workload than the average. To address this problem, it has been suggested that the size of blocks can be profiled and leveraged for skew avoidance. More specifically, block size profiles have been used to design subkey schemes [55], block division strategies [43], and record pair allocation methods [43] to balance the load of the reducers.

However, establishing an accurate profile for block sizes is extremely challenging for massive datasets with millions or billions of block keys. For example, when one attempts to establish a per-block size profile for large datasets, such as the DBLP-10 or

[2]Here the workload is measured as the number of record pairs in each block.

DBLP-20 (again, we refer the readers to Table IV.1 for details regarding these datasets), the profile needs to record the size for over 5 million blocks. As a result, the program crashes when the profile is loaded onto a virtual machine that assigns 1 GB memory for each task[3]. In general, when blocking keys are linked to the input data, the blocking key's value domain can be extremely large and challenging to predict *a priori*. This clearly limits the applicability of precise profiling, which establishes accurate per-block size information, and its associated load balancing approaches to big datasets where millions to billions of blocks are the norm.

In this chapter, we introduce scalable reduce-phase load balancing solutions for record linkage over the MapReduce framework. To do so, we address two specific problems: (1) how to design an efficient data structure that summarizes the block-related load information and (2) how to leverage information recorded in this structure to assign records to reducers so that their loads are balanced. To address the first problem, we introduce a sketch-based data profiling method [21] for capturing block size statistics. For context, informally, a sketch is a two-dimensional array of cells, each indexed by a set of pairwise independent hash functions. For the second problem, we present two load balancing algorithms – *cell block division* and *cell range division* – that directly operate on sketch-based data profiles to achieve reducer load balancing.

Chapter III also introduced a sketch-based solution to capture key group size statistics and presents *optimal sketch packing* algorithm which assigns the key groups to the reducers in a load balancing manner. However, the *optimal sketch packing* algorithm cannot be deployed to support traditional record linkage applications for several reasons. First, the *optimal sketch packing* algorithm assumes the key group workload is proportional to its size, which is not appropriate in this situation due to the fact that

[3]In the current generation of MapReduce/Hadoop (i.e., YARN [81]), 1 GB is the default allocated memory for each task.

record linkage is, in effect, a join operation. Furthermore, the *optimal sketch packing* algorithm works on the granularity of key groups and treats each key group as an indivisible unit. This characteristic makes the *sketch packing* algorithm a generalized load balancing solution which is appropriate for several applications (e.g., PageRank and inverted indexing, etc.), but also limits its performance with highly skewed datasets (as demonstrated in Figure III.6(c)). Datasets used in record linkage applications are often highly skewed. In the example shown in Figure IV.1(a), where the maximum block takes 55% of total workload. We name this type of block (key group) the *expensive* block (key group). Wherever the *optimal sketch packing* algorithm assigns this expensive block, that reducer would be the straggler. These two reasons limit the *optimal sketch packing* algorithm performance on the record linkage application.

The main contributions of our work are summarized as follows.

- First, our sketch-based data profiling method is (1) *scalable* with the size of the input data and the number of blocks and (2) *efficient* for construction, such that each update takes only constant time.

- Second, our proposed *cell block division* and *cell range division* algorithms can efficiently divide expensive blocks without losing any record pairs that need to be compared.

- Third, our theoretical analysis shows that our load balancing algorithms have a bounded load balancing performance, as well as computational complexity.

- Fourth, we perform an empirical study using several real-world and synthetic datasets to demonstrate that our algorithms, which are limited to a fixed memory size, can achieve near-optimal load balancing performance in comparison to

precise profiling which maintains all block sizes in memory, and incurs only a very small running time overhead.

Background

Record Linkage

Record linkage is the process of matching records on specific entities (e.g., "*John Smith*" and "*J. Smith*" may refer to the same person.) in disparate sources. Large-scale record linkage frameworks involve three fundamental data-intensive steps [29]. The first step is *blocking*, which uses a quick coarse-grained similarity filtering strategy to produce subsets (i.e., blocks) of the record pair set, each of which contains pairs that likely correspond to the same entity (i.e., candidate record pairs). As a commonly used blocking strategy, each input record is assigned with a blocking key k; records with the same k are grouped together into a block[4]. Only records within the same block are compared with each other in the ensuing *comparison* step. This step involves the assessment of multiple fields between a pair of records to produce a similarity vector. The third step is *classification*, which determines the match status of each record pair based on their similarity vectors and outputs the set of matches and non-matches.

As in [43] and [44], we focus on optimization for the first two steps because the final classification step can utilize several existing statistical strategies that are independent of the scale of the problem [30]. Thus, the whole process can be described as: given two sets of records R and S, compute the similarity vector for each pair of records from different datasets with the same blocking key. We refer to the similarity vector

[4]For various blocking mechanisms, we refer readers to [16].

for records r_α and r_β as $\overline{x}$ = [x_1, x_2, ..., x_t] with t components that correspond to the t comparable fields. Each x_i corresponds to the level of agreement of the i^{th} field of the records r_α and r_β. x_i is computed using a similarity function *sim*, such as the edit distance or Q-gram distance. For example, records {*John, Smith, Nashville*} and {*Jon, Smyth, Nashville*} would generate a similarity vector [0.75, 0.80, 1.00] when we use edit distance to calculate the similarity.

Blocking-based Record Linkage in MapReduce

Recently proposed approaches [44, 60, 85] have favored a common design to support the blocking-based record linkage process over MapReduce. In this design, datasets are partitioned by *map* tasks into several blocks using *blocking keys* and subsequently assigned to parallel *reduce* tasks, where record pairs are constructed for comparison. Figure IV.2 illustrates an example of this process, where field a acts as the blocking key. Records with the same blocking key are sent to the same reducer, where the similarity vectors are built.

A common issue faced by the current design of record linkage protocols over the MapReduce framework is reduce-phase *data skew*. When the block size distribution is highly skewed, the default MapReduce hash-based key group assignment mechanism can assign some reducers a much higher workload than others. This results in a prolonged reduce-phase running time. For example, in Figure IV.2, the first reducer needs to compare 7 record pairs, while the other two reducers only have to compare 2 and 4. Data skew manifests because of an imbalanced distribution of block sizes and the MapReduce default hashing partition mechanism.

To achieve reduce-phase load balancing, existing solutions introduce an additional MapReduce job to establish a block size profile [43, 55] which records the number of

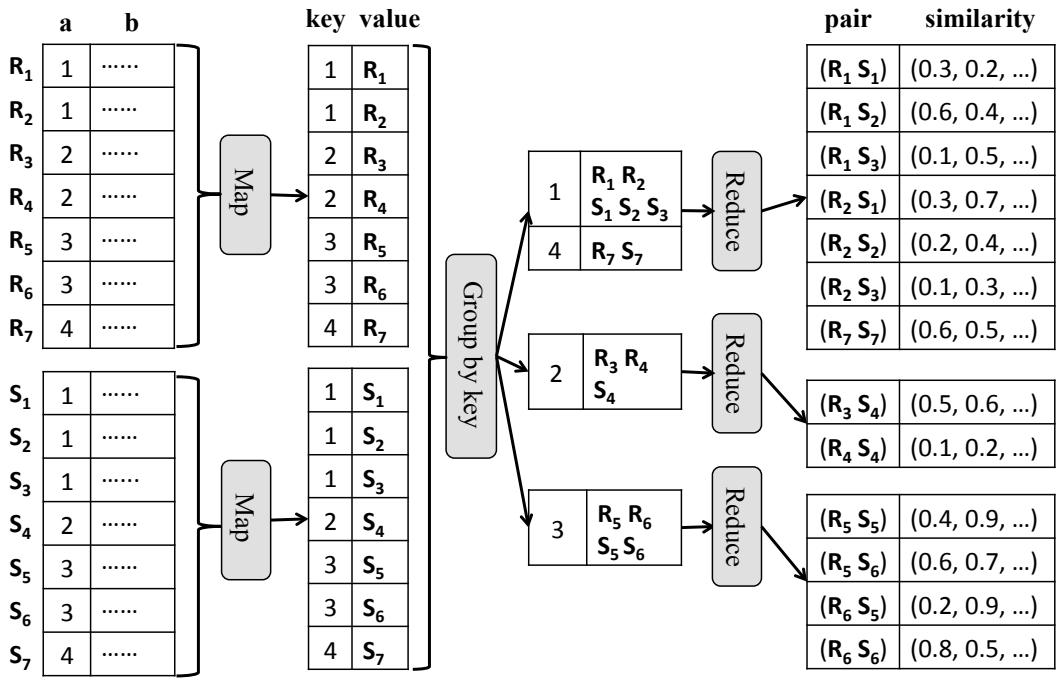


Figure IV.2: An example of blocking-based record linkage using MapReduce.

records within each block. This profile can then be used to design subkey [55], block division [43], or record pair allocation [43] schemes so that the load at the reducers can be balanced. For instance, in [43], a MapReduce job is included to build a block size matrix for the number of records within each block. A global index is then reported for each record pair and each reducer is assigned with an index range of equal length, thus achieving load balancing across reducers. A major limitation of the aforementioned block-based size profiling method is its scalability. In reality, datasets to be linked can be extremely large – on the order of millions or billions. Thus, given current computing architectures, it is impossible to maintain a precise block size matrix in a space-limited environment.

In this chapter, we seek an approximate data profiling method that is both memory- and time-efficient. For such a method to scale to massively-sized datasets, the memory cost should be independent of the number of blocks and the processing time should be

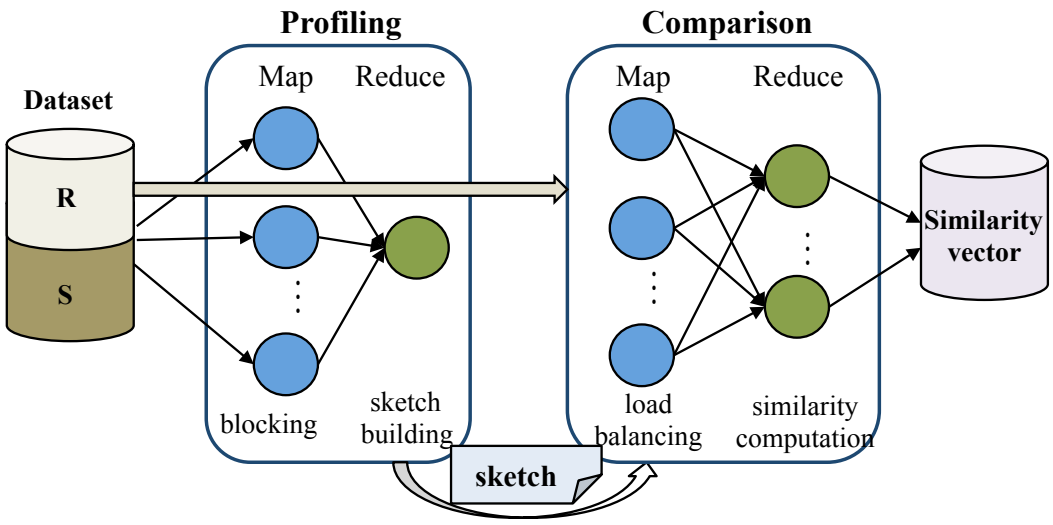


Figure IV.3: The workflow of MapReduce-based record linkage facilitated by sketch-based profiling.

linear (or sublinear) in the size of the input dataset. Moreover, the data profile should introduce a bounded approximation error and yield a highly accurate load balancing strategy, regardless of the data skew.

Sketch-based Profiling and Load Balancing Solution

Figure IV.3 illustrates the overall design of our system with a depiction of linking two datasets R and S. Similar to the approaches adopted in [43], our design is based on two rounds of MapReduce jobs. The *profiling* job analyzes the input datasets and provides the load estimation in terms of *sketch* for input datasets R and S. The load information is supplied by map tasks into the second MapReduce job (*comparison*), where load balancing strategies are applied to perform the actual record linkage task.

We first describe the centerpiece of our approach – how to build, and achieve load balancing based upon, the sketch. A sketch [21, 22] is a data structure that provides space-efficient summaries for massive, rapid-rate data streams. Here, we use the sketch data structure to estimates block sizes for the input data. Specifically, we use the FastAGMS sketch [21] because it provides the most accurate estimation for the size of join operation[5] [69], regardless of data skew. We also evaluate other type of sketches in Section IV, such as Count-Min sketch [22], to verify this statement.

To estimate the workload in terms of record-pair comparisons within each block, we maintain two FastAGMS sketches for datasets R and S, which we refer to as C_R and C_S, respectively. Each FastAGMS sketch $C \in \{C_R, C_S\}$ maintains a two-dimensional array of cells with d rows of width w, which are indexed by a set of pairwise independent hash functions $\mathcal{H} = \{h_i | i = 1, \ldots, d\}$. Each hash function h_i maps a blocking key k into a hashing space of size w (i.e., $h_i(k) \in \{0, 1, \ldots, w-1\}$)[6]. The FastAGMS sketch also maintains a family of ± 1 four-wise independent hash functions $\mathcal{G} = \{g_i | i = 1, \ldots, d\}$[7]. This family of hash functions preserves the dependencies across the counters.

Each cell of the sketch carries a counter. Initially, all of the counters in the array are set to zero.

$$C[i, j] = 0, \text{for all } i \in \{1, ..., d\}, j \in \{1, ..., w\}$$

When a new blocking key k is emitted, the counters are updated as shown in Algorithm 4.

$$C[i, h_i(k)] = C[i, h_i(k)] + g_i(k), \text{for all } i \in \{1, ..., d\}$$

[5]Record linkage can be treated as a join operation where the blocking key act as the join key.

[6]$h_i(k) = (a_i k + b_i) \bmod w$.

[7]$g_i(k) = \begin{cases} 1 & \text{if } (a_i k^3 + b_i k^2 + c_i k + d_i) \bmod 2 = 0 \\ -1 & \text{otherwise} \end{cases}$.

Algorithm 4 Update operation for FastAGMS sketches C_R and C_S

```
1: function UPDATE(r, C_R, C_S, H, G)
2:     // calculate blocking key
3:     k ←calculateBKV(r)
4:     // update sketch
5:     if r ∈ R then
6:         for i = 1 → d do
7:             C_R[i, h_i(k)] ← C_R[i, h_i(k)] + g_i(k)
8:         end for
9:     else if r ∈ S then
10:        for i = 1 → d do
11:            C_S[i, h_i(k)] ← C_S[i, h_i(k)] + g_i(k)
12:        end for
13:    end if
14: end function
```

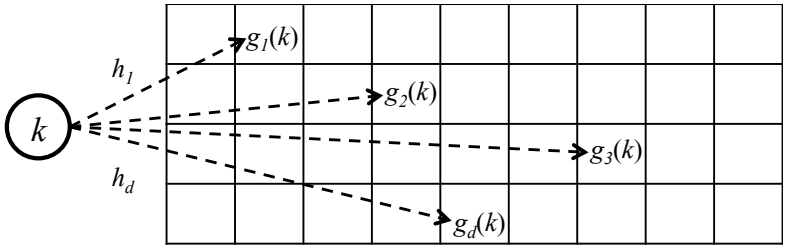


Figure IV.4: An example of the FastAGMS sketch update process ($w = 9$ and $d = 4$).

Specifically, for each row i, $h_i(k)$ determines the cell to be updated, and $g_i(k)$ decides whether to increment or decrement the counter in the corresponding cell. Figure IV.4 illustrates this update process with $w = 9$ and $d = 4$. Here, a key-value pair with key k is mapped to a counter in each row i ($i \in \{1, 2, ..., d\}$) by the hash function h_i and increments the counter by $g_i(k)$.

The sketches C_R and C_S are constructed based upon the same parameters (i.e., $d, w, \mathcal{H}, \mathcal{G}$). Each sketch provides an approximate summary of block sizes $L(k)$ for each dataset, where the sizes of multiple blocks are compressed into one cell. Recall that the record linkage workload in terms of record pair comparisons is the product of the block sizes from these two datasets $L_R(k) \times L_S(k)$. To estimate this workload, we

consider the inner product of the sketches, which is accomplished through two steps. In the first step, we choose the median value of the row inner products. Formally, the row inner product is provided by:

$$C^i = \sum_{j=1}^{w} C_R[i,j] \times C_S[i,j], i \in \{1, 2, ..., d\}$$

Let the value of row $i = \theta$ be the median value among these d row inner products. In the second step, we use this row to build a counter array C^θ with width w as follows.

$$C^\theta[j] = C_R[\theta, j] \times C_S[\theta, j], \text{for all } j \in \{1, ..., w\}$$

The array C^θ provides the estimation of the record-pair comparison workload within blocks and will be applied in load balancing algorithms.

Implementation in MapReduce. The map tasks of the *profiling* job build local sketches (C_R^L and C_S^L) based on the input records from the two datasets. Once completed, the local sketches are sent to one reducer, where they are combined to build the final sketch (C_R and C_S). Sketch combination is straightforward: local sketches with the same sizes are combined by summing them up, entry-wise.

The outputs of the *profiling* job are the final inner product sketch C^θ, as well as the corresponding row vectors from the sketches of the two datasets $C_R[\theta]$ and $C_S[\theta]$. Our load balancing algorithms work directly on these values.

Cell Block Division Algorithm

The C^θ from the *profiling* job is loaded by each map task in the *comparison* job before processing the input records. As noted earlier, C^θ provides an estimation of the record comparison workload, where each cell carries the estimated workload for

multiple blocks. Let $\hat{L}$ represent the estimated overall workload, then $\hat{L} = \sum_{j=1}^{w} C^{\theta}[j]$. The average reducer workload can then be estimated as $\hat{L}/n$, where n is the number of reducers.

Since C^{θ} contains w cells, a simple idea that might come to mind is to pack these cells into n partitions and assign each reducer its own partition. However, some cells may have a workload larger than the average $\hat{L}/n$, so a division procedure is required for large cells (i.e., the function *DivideCell* in Algorithm 5). For each cell in C^{θ}, if its estimated workload is larger than average, we divide it into several subcells; otherwise, we keep the cell as a single subcell. Lines 4–9 in Algorithm 5 illustrate the process of calculating subcells. All subcells are maintained in a set $\mathcal{S}$. Finally, a packing operation is performed on the set $\mathcal{S}$. The result Φ is a mapping from subcells to reducers.

Figure IV.5 presents an example of cell block division with $w = 4$ and an estimated workload of 10. C^{θ} contains four cells, and their workload is $\{1\times1, 2\times3, 1\times1, 2\times1\}$. Now assume there are three reducers and the estimated average workload is 4. Notice, cell $C^{\theta}[2]$ is larger than the average, so it is divided into two subcells. In our implementation, we always follow row-based division, such that cells are divided along the axis correspond to R. After division, $\mathcal{S}$ has five (sub)cells. Next, we perform the packing operation, where each (sub)cell is assigned to the reducer with the minimum workload.

Based on the mappings Φ, the map tasks in the *comparison* round identify the corresponding reducers for each record r, to which r needs to be sent (implemented in function *GetReducer* in Algorithm 5). First, the corresponding cell j for the given record r is calculated (line 19). For each record from dataset R, as we perform row-based division, we only need to send r to one reducer. Since each cell is divided into $D[j]$ subcells with the same size, we can randomly select a subcell and obtain its reducer (lines 22 – 24). For each record from S, we need to send it to all reducers that map to the current cell (lines 26 – 27).

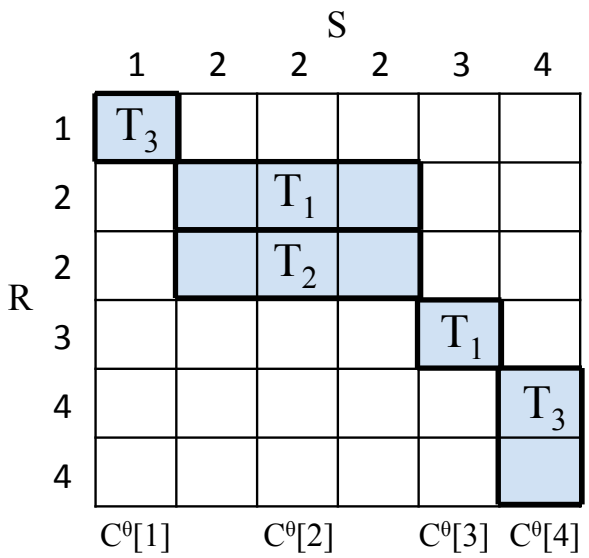


Figure IV.5: An example of cell block division.

For example, in Figure IV.6, assume record r is hashed to $C^{\theta}[2]$ which has two subcells. If r comes from dataset R, we randomly select one subcell and send r to reducer T_1 or T_2. On the other hand, if r is from S, we send r to both reducers T_1 and T_2.

Through such a cell block division approach, we can limit the maximum cell workload and achieve better load balancing performance. In Figure IV.5, the maximum reducer workload is 4 (reducer T_1). If no division introduced, the reducer that is assigned with $C^{\theta}[2]$ would become the straggler and have a workload of 6.

Cell Range Division Algorithm

While the *cell block division* algorithm divides large cells, it may still lead to imbalanced reducer workloads due to variation in the size of the subcells. To account for this problem, we now present a more sophisticated pair-based load balancing strategy that strives to generate a uniform number of pairs for all reduce tasks.

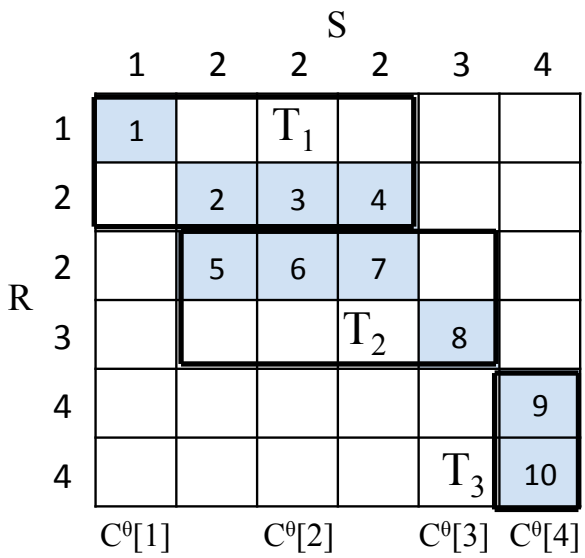


Figure IV.6: An example of cell range division.

Each map task processes C^θ and can therefore enumerate the workload per cell. We label each record pair in C^θ with a global index, and divide all record pairs into n equal-length ranges (lines 4 – 8 in Algorithm 6). For a given cell $C^\theta[j]$, the overall number of record pairs in all preceding cells has to be added as an index offset, which we maintain the offset in an array O (lines 9 – 13). To further simplify the process, our cell range division mechanism treats each row in a cell $C^\theta[j]$ as a unit and thus does not divide it.

Figure IV.6 presents an example of the cell range division with $w = 4$ and an estimated workload of 10. Each record pair is labeled with an index from 1 to 10. Let the reducer number be 3 and the set of record pairs be divided into three ranges. Reducers T_1, T_2, and T_3 will process the record pairs with indexes in the ranges [1,4], [5,8] and [9,10], respectively.

The function *GetReducer* in Algorithm 6 determines the reducers that record r is sent to. First, we find the corresponding cell for r. Next, we calculate the start and end indices for record pairs that are related to r. If r comes from dataset R, we randomly

select a row for r in its corresponding cell. The start index o_1 is calculated as all preceding record pairs (line 21), and the end index o_2 is the end of the selected row (line 22). Each record from S needs to be sent to all reducers that map to current cell (lines 24 – 25). Finally, we determine the reducers by supplying the start and end record pair indexes (line 28).

For example, in Figure IV.6, assume a record r is hashed to $C^{\theta}[2]$. If r comes from R, we randomly select a row in $C^{\theta}[2]$, and send r to reducer T_1 or T_2. If r comes from S, we need to send r to both reducers T_1 and T_2.

Performance Analysis

Here we analyze the memory and computational complexity of our proposed algorithms, and the load balancing performance.

Proposition 3 *The memory complexity of our profiling method is $O(d \times w)$, where d is number of rows and w is the width. The computational complexity of sketch update is $O(d)$.*

Proof 3 *In the* profiling *job, each map/reduce task maintains two sketches with size dw, thus its memory complexity is $O(dw)$. The outputs of the* profiling *job are C^{θ}, $C_R[\theta]$ and $C_S[\theta]$, each of which is an array with width w. The map tasks in the* comparison *job load them into memory and thus introduce a cost of $O(w)$. As a result, the total memory complexity is $O(dw)$.*

Since each update process involves d counters, the computational complexity is $O(d)$.

As in [43, 60], we measure the load balancing performance as the reduce-phase imbalance ratio.

Definition 1 *Reduce-phase imbalance ratio ρ: Let L_i represent the workload of reducer T_i, the imbalance ratio ρ is calculated by normalizing the maximum reducer workload by the average workload.*

$$\rho = \frac{\max_{i=1}^{n} L_i}{\sum_{i=1}^{n} L_i/n}$$

To analyze the maximum reducer workload in our algorithms, we bound the estimated workload $\hat{L}$.

Lemma 2 *According to the analysis in [21], the workload estimation $\hat{L}$ for two FastAGMS sketches C_R and C_S with size $d \times w$ guarantees that $\hat{L} \in (L \pm \varepsilon||R||_2||S||_2)$, with probability at least $1 - \delta$. Here, $\varepsilon = e/w$, $\delta = 1/e^d$, e is the base of the natural logarithm, L is the accurate workload, and $||.||_2$ is the L_2-norm.*

Next, we analyze the load balancing performance bound for the reduce-phase imbalance ratio of our proposed algorithms in the following theorems.

Theorem 2 *The reduce-phase imbalance ratio of the* cell block division *algorithm is at most $(2 - \frac{1}{n})(1 + \frac{\Delta}{L})$, with a probability of at least $1 - \delta$, where $\Delta = \varepsilon||R||_2||S||_2$.*

Proof 4 *Suppose reducer s receives the maximum workload and subcell sc is the last one it is assigned. Let L_s be the workload of reducer s before it receives sc. When subcell sc is assigned, the workload of its reducer is no larger than the other reducers, so every reducer has a larger workload than L_s. Thus, the maximum reducer workload is*

$$L_{max} = L_s + L_{sc} \leq \frac{\hat{L} - L_{sc}}{n} + L_{sc} = \frac{\hat{L}}{n} + (1 - \frac{1}{n})L_{sc}$$

Since each subcell has a workload less than $\hat{L}/n$, it can be stated that

$$L_{max} \leq \frac{\hat{L}}{n} + (1 - \frac{1}{n})\frac{\hat{L}}{n} = (2 - \frac{1}{n})\frac{\hat{L}}{n}$$

Now, let $\Delta = \varepsilon||R||_2||S||_2$. *It is the case that*

$$L - \Delta \leq \hat{L} \leq L + \Delta$$

and the imbalance ratio is at most

$$\rho = \frac{(2 - 1/n)\hat{L}/n}{L/n} \leq \frac{(2 - 1/n)(L + \Delta)}{L} = (2 - \frac{1}{n})(1 + \frac{\Delta}{L}).$$

Theorem 3 *The reduce-phase imbalance ratio of the* cell range division *algorithm is at most* $1 + \frac{\Delta}{L}$ *with a probability of at least* $1 - \delta$, *where* $\Delta = \varepsilon||R||_2||S||_2$.

Proof 5 *Let* $\Delta = \varepsilon||R||_2||S||_2$. *Then, it is the case that*

$$L - \Delta \leq \hat{L} \leq L + \Delta,$$

with a probability at least $1 - \delta$. *Since each reducer has been assigned estimated workload of* $\hat{L}/n$, *the maximum reducer workload* $L_{max} = (L + \Delta)/n$. *As a result, the imbalance ratio is at most*

$$\rho = \frac{(L + \Delta)/n}{L/n} = 1 + \frac{\Delta}{L}.$$

Experimental Evaluation

Experiment Setup

Dataset. Our evaluation is performed with two classes of datasets, the details of which are shown in Table IV.1. The first is based on the **DBLP** dataset[8], which has approximately 1.2 million publications. In this dataset, we use the first two words of the publication title as the blocking key. To scale up our evaluation, we increase the dataset size to β times, where $\beta \in \{1, 5, 10, 20\}$. Specifically, for each record in the dataset, we generate β duplicates, each of which has a new blocking key by adding a random letter to the old blocking key. Through this approach, we can increase the number of blocks when increasing the dataset size. We refer to these datasets as DBLP-β, where DBLP-1 represents the original dataset.

The second class of datasets, which we refer to as **Synth-**α, is synthesized from the DBLP-20 dataset by manipulating the block size distribution. Specifically, we use a Zipf distribution and vary the skew parameter $\alpha \in \{0.5, 1.0, 1.5, 2.0\}$[9]. We first fix the number of blocks and record pairs for all the datasets. For a given α, we calculate the number of record pairs inside each block, and extract the records from the corresponding block in the DBLP-20 dataset. The original dataset acts as R in our experiment, while the dataset S for linkage is generated from R by making random modifications [17] (e.g., deletion, insertion, et al.) to the fields other than the blocking key.

Running Environments. All experiments were performed on a 40-node cluster running Hadoop 1.0.1 with a separate master node. Each node has one 2.4 GHz Intel Core2 CPU with 2 GB of memory. The HDFS block size was set to 64 MB and each node was configured to run at most two map tasks and two reduce tasks concurrently.

[8] http://dblp.uni-trier.de/xml

[9] For a given α, the number of record pairs in the k^{th} block is proportional to $k^{-\alpha}$. As such, α is a proxy for the amount of data skew simulated.

Table IV.1: Summary for experimental datasets used in Chapter IV

Dataset	Records (million)	Blocks (million)	Pairs (billion)
DBLP-1	2.5	0.6	0.5
DBLP-5	12.6	3.3	1.8
DBLP-10	25.2	5.7	4.0
DBLP-20	50.4	7.9	16.9
Synth-0.5	416	5.0	10.0
Synth-1.0	226	5.0	10.0
Synth-1.5	32.8	5.0	10.0
Synth-2.0	21.6	5.0	10.0

We disabled the speculative task execution feature to better analyze the running time of each task. By default, each MapReduce job is configured with 80 reducers.

Baseline Algorithms. We evaluate four algorithms: (1) the Hadoop default (HD) algorithm, which uses the hash-based partition function for key group assignment; (2) the pair-based (PR) algorithm [43], which utilizes the precise block profile; (3) our *cell block division* (CB) algorithm (Algorithm 5); and (4) our *cell range division* (CR) algorithm (Algorithm 6).

The performance of the algorithms is measured in terms of (1) *job running time*, which is the entire running time including both profiling and comparison rounds; (2) *imbalance ratio of reducer workload in the comparison job*, which is calculated by normalizing the maximum reducer workload by the average reducer workload. Here, the reducer workload is measured as the number of record pairs received by each reducer.

Sketch and Profile Settings. Unless stated otherwise, the sketch structure is set to $w = 10000$ and $d = 10$, which requires less than 1 MB of memory.

Performance of CB and CR algorithms

We first evaluate the performance of the CB and CR algorithms using the DBLP datasets. Figure IV.7 shows the running time of each phase. No results are shown for

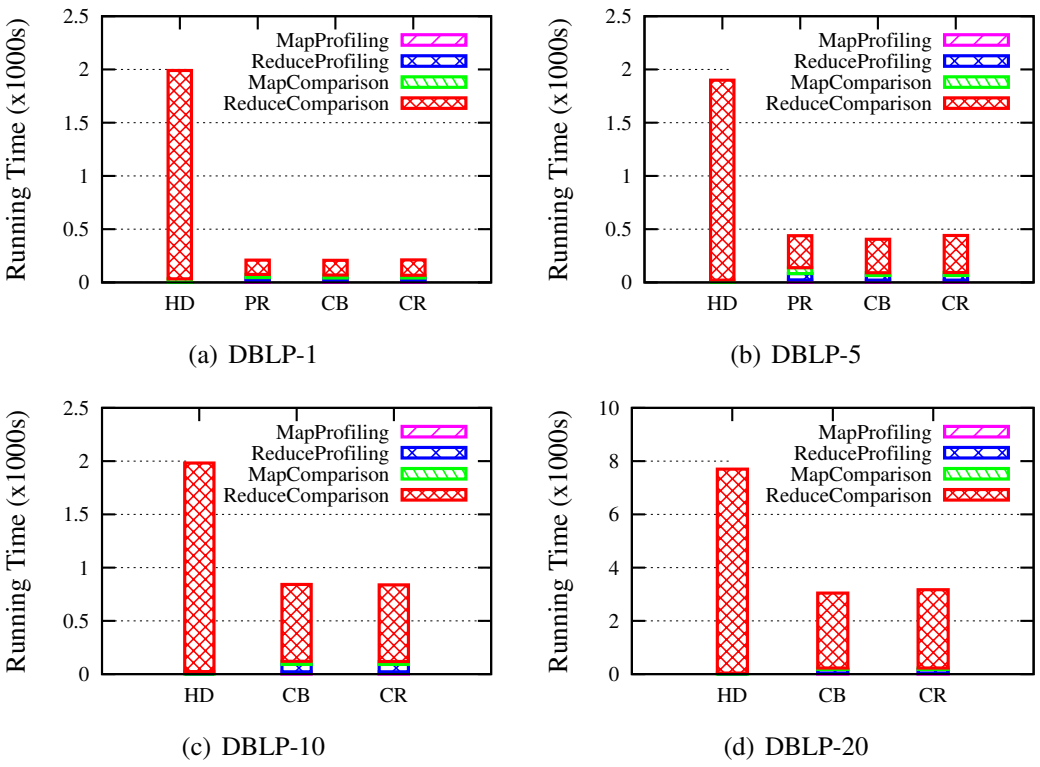


Figure IV.7: Job running time for DBLP datasets.

the PR algorithm in the experiments with the DBLP-10 and DBLP-20 datasets because the program crashed on account of an *OutOfMemory* exception. In other words, the precise block size profile could not be maintained in memory.

In comparison to HD, the other three algorithms introduce another MapReduce job (*profiling*) to build a data profile. As such, their overall job running time also includes the time spent in the *profiling* job. It can be seen that PR, CB and CR, which balance the load among reducers, all significantly reduce the job running time compared with the HD algorithm.

Figure IV.7(a) and IV.7(b) show that PR, CB and CR have similar running time. Further, CB and CR require less time in the *ReduceProfiling* and *MapComparison* (which is

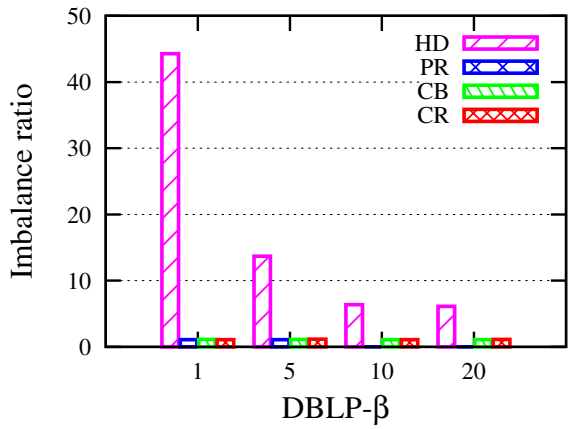
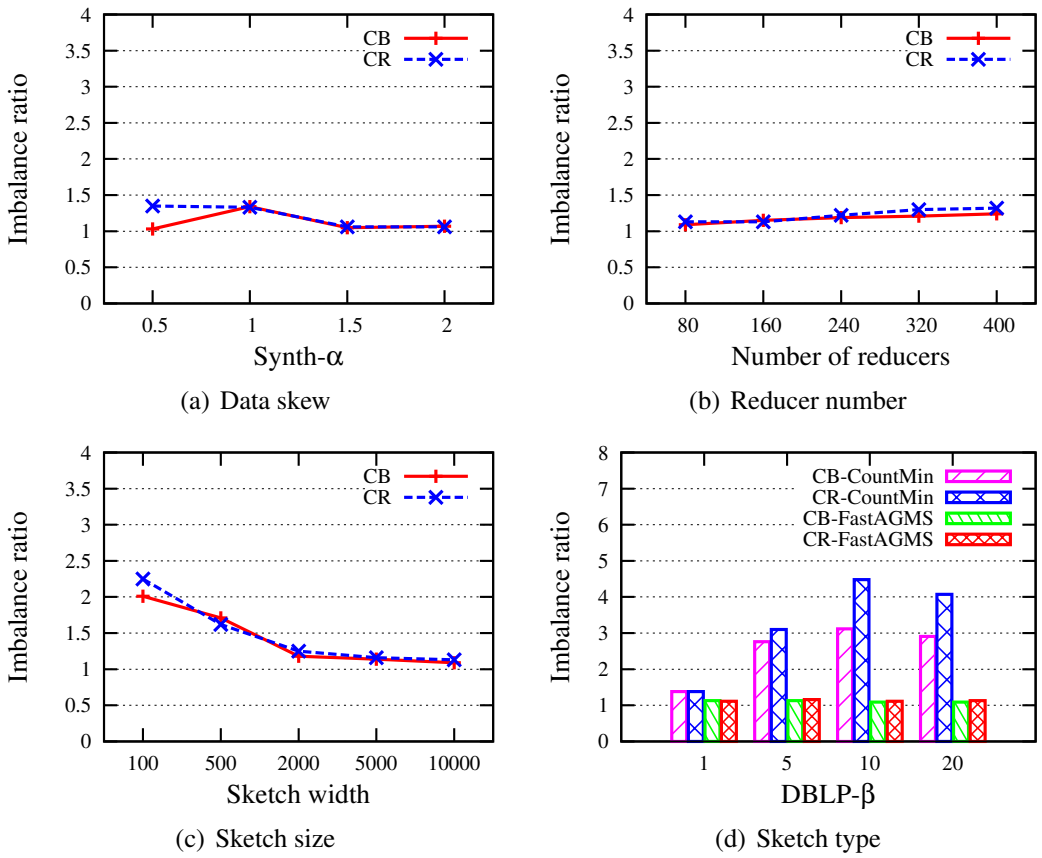


Figure IV.8: Reduce-phase imbalance ratio for DBLP datasets.

particularly noticeable in Figure IV.7(b) with 34.1% reduction). This is because our algorithms operate on the sketch (which are time-efficient for construction and retrieval), while PR maintains a hash table in memory (whose operation time increases dramatically with its size when collisions occur).

In comparison to HD, the *profiling* job in the CB and CR algorithms introduces 3% extra running overhead on average. However, the entire running time is reduced by 71.56% and 70.73%, respectively.

We also report the number of record pairs processed by each reducer and calculate the reduce-phase imbalance ratio. Both CB and CR achieve nearly optimal reducer-side load balancing with an imbalance ratio around 1.1 as shown in Figure IV.8. Moreover, in comparison to PR, our algorithms increase the imbalance ratio by only 2.5%, which indicates that only a very small load balancing performance penalty is introduced by approximate data profiles. The imbalance ratio of HD highly depends on the input data and is always much higher than the other algorithms.



(a) Data skew (b) Reducer number

(c) Sketch size (d) Sketch type

Figure IV.9: Reduce-phase imbalance ratio under various settings.

Performance under Various Data Skew

We further evaluate CB and CR under various data skew scenarios using the Synth-α datasets. In this study, we mainly focus on the reduce-phase imbalance ratio in the *comparison* MapReduce job, which has 80 reducers. As shown in Figure IV.9(a), CB and CR have similar performance. The imbalance ratio remains around 1.25. Since both CB and CR divide large cells, their performance is highly stable with respect to the amount of skew of the dataset.

Performance under Number of Reducers

We now study the influence of the number of reducers n in a fixed cloud environment of 40 nodes. We evaluate CB and CR with the DBLP-20 dataset, and vary the reducer number from 80 to 400. As shown in Figure IV.9(b), both CB and CR maintain stable performance as the number of reducers increases.

Experiments with Sketch Size

Here we fix the number of sketch rows to $d = 10$ and vary the width w from 100 to 10000. Figure IV.9(c) shows that the imbalance ratio is inversely correlated with the sketch width. This phenomenon is also reflected in our theoretical analysis – when the number of reducers increases, the workload estimation error decreases. Our empirical study shows that the best performance is achieved when the sketch width is set between 50 to 100 times that of the number of reducers.

Experiments with Various Types of Sketches

In this chapter, we use the FastAGMS sketch [21] as the default implementation of sketch data structure because it provides the most accurate workload estimation [69]. Yet, there are also other sketch implementations, such as Count-Min sketch [22]. Thus, in this study, we deploy Count-Min sketch in our CB and CR algorithms and compare the results with the default FastAGMS implementation. We evaluate this variation of our method using the DBLP-β datasets, and calculate the reduce-phase imbalance ratio for each group of experiment. As anticipated, Figure IV.9(d) indicates FastAGMS sketch performs significantly better than Count-Min sketch.

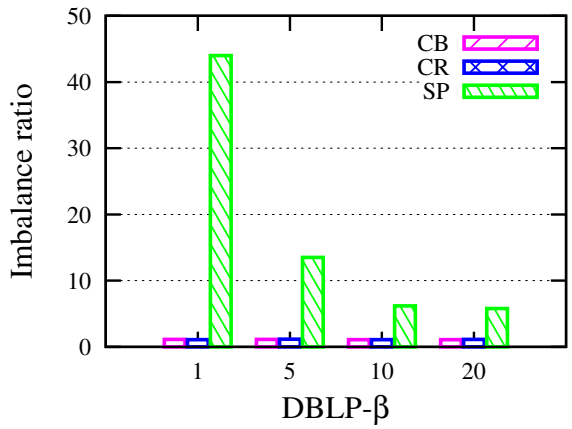


Figure IV.10: Reduce-phase imbalance ratio in comparison to *optimal sketch packing* algorithm.

Comparison with *optimal sketch packing* Algorithm

As discussed in Section IV, C^{θ} provides an estimation of the record comparison workload, where each cell carries the estimated workload for multiple blocks. Since C^{θ} contains w cells, a simple idea is to deploy the *optimal sketch packing* algorithm to pack these cells into n partitions and assign each reducer with one partition.

Here, we evaluate the *optimal sketch packing* approach (SP), in comparison to the cell subdivision strategy invoked by our CB and CR algorithms. To do so, we use the DBLP-β datasets with 80 reducers. Figure IV.10 shows the reducer workload imbalance ratio. Since the SP algorithm does not divide cells, its performance is limited by the data skew. For instance, in the DBLP-1 dataset where the maximum block requires 55% of the entire workload, the maximum reducer workload is 44x ($\approx 80 \times 55\%$) more than the average.

Chapter Summary

In this chapter, we presented a scalable solution to achieve load balanced record linkage over the MapReduce framework. The solution contains two low-memory load

balancing algorithms that work with a sketch-based approximate data profiles. We performed a theoretical and an empirical analysis on both real-world and synthetic datasets to demonstrate that, compared with the state-of-the-art solution, our algorithms have nearly the same load balancing performance while requiring much less memory.

Algorithm 5 Cell Block Division Algorithm

function DIVIDECELL(C^{θ}, n)
    `// calculate estimated workload`
    $\hat{L} \leftarrow \sum_{j=1}^{w} C^{\theta}[j]$
    `// calculate division number for each cell`
    for $j = 1 \rightarrow w$ **do**
        $D[j] \leftarrow$ Math.ceil($C^{\theta}[j] \times n/\hat{L}$)
        $subcells[] \leftarrow$ createSubCells($row, j, D[j]$)
        $\mathcal{S} \leftarrow \mathcal{S} \cup subcells$
    end for
    `// perform packing operation on (sub)cells`
    sort($\mathcal{S}$)
    for all $subcell \in \mathcal{S}$ **do**
        $reducerID \leftarrow$ selectMinLoadedReducer()
        $\Phi(subcell) \leftarrow reducerID$
    end for
end function

function GETREDUCER($r, \mathcal{H}, \theta, D, \Phi$)
    `// cell index for current record` r `in row` θ
    $cell \leftarrow h_{\theta}(r)$
    `// assign record to reducers`
    if $r \in R$ **then**
        `// send to one reducer`
        $rand \leftarrow$ random($D[cell]$)
        $reducerID \leftarrow$ getReducerID($\Phi, cell, rand$)
    else if $r \in S$ **then**
        `// send to all reducers related to current cell`
        $reducerIDs \leftarrow$ getAllReducerID($\Phi, cell$)
    end if
end function

Algorithm 6 Cell Range Division Algorithm

function DIVIDERANGE(C^θ, n)
    // calculate estimated workload
    $\hat{L} \leftarrow \sum_{j=1}^{w} C^\theta[j]$
    // assign each range to one reducer
    $avg \leftarrow \hat{L}/n$
    for $i = 1 \rightarrow n$ **do**
        $\Phi \leftarrow \Phi \cup \{[(i-1) \times avg + 1, i \times avg], i\}$
    end for
    // calculate offset of each cell
    $O[1] \leftarrow 0$
    for $j = 2 \rightarrow w$ **do**
        $O[j] \leftarrow O[j-1] + C^\theta[j-1]$
    end for
end function

function GETREDUCER($r, \mathcal{H}, \theta, C^\theta, C_R[\theta], C_S[\theta], O, \Phi$)
    // cell index for current record r in row θ
    $cell \leftarrow h_\theta(r)$
    // calculate the start and end indexes
    if $r \in R$ **then**
        $rand \leftarrow$ random($C_R[\theta, cell]$)
        $o_1 \leftarrow O[cell] + C_S[\theta, cell] \times rand$
        $o_2 \leftarrow o_1 + C_S[\theta, cell]$
    else if $r \in S$ **then**
        $o_1 \leftarrow O[cell]$
        $o_2 \leftarrow O[cell] + C^\theta[cell]$
    end if
    // assign record to reducers
    $reducerIDs \leftarrow$ getReducerIDs(Φ, o_1, o_2)
end function

CHAPTER V

COORDINATED RESOURCE MANAGEMENT FOR LARGE SCALE INTERACTIVE DATA QUERY SYSTEMS

In Chapter III and IV, we discussed how to achieve reduce-phase load balancing for a single MapReduce job. In this chapter, we investigate the problem of optimizing resource management across multiple jobs. We take interactive ad hoc queries as an example application here.

The chapter is organized as follows. We first motivate the need for solving resource management for interactive ad hoc data queries. In Section V.1, we review the system architecture, as well as the query model. In Section V.2, we formulate the resource allocation problem and propose our solution. We investigate the performance of our optimal resource allocation algorithm via 1) simulation in Section V.3 and 2) a real cluster using TPC-DS workload [1] in Section V.4. We summarize the chapter in Section V.5.

Motivation

Large-scale interactive data analysis has grown increasingly important in many domains for data exploration, decision making and strategy planning. Performing data analysis over massive datasets with interactive response time requires a high degree of parallelism. The recent design of Dremel [56] has demonstrated such capability through massively parallel computing over a shared-nothing parallel data storage architecture. Based on similar design, several open source systems are built and widely deployed in production clusters, including Cloudera Impala [19], Apache Drill [6], Linkedin Tajo [53], and Facebook Presto [63]. These systems support SQL-like queries over

distributed data in a clustering environment and largely democratize big data by providing easy data access for users without any distributed system background, who are previously locked away from large datasets.

In these systems, each query is first compiled into a plan tree, which is then decomposed into several query fragments (refer to Figure V.3 for an example). Each fragment is dispatched to the machines where its data blocks are located, and each machine is assigned one or more fragments. Depending on the query semantics (i.e., SQL operation), the execution of each query is then converted into a set of coordinated tasks, including data retrieval, intermediate result computation and transfer, and result aggregation. For load balancing purposes, the root of the query plan tree can be any machine in the cluster that runs a daemon interacts with clients and coordinates the pipelines between computation stages. As such, each query consumes a different amount of resources (e.g., CPU, memory, and I/O) at each machine.

Since significant benefits can often be realized by sharing the cluster among multiple clients, a principal challenge here is the development of efficient resource management mechanisms to support concurrent interactive queries. Coordinated management of multiple resource of the cluster environment is critical to provide a guarantee on service-level agreement (SLA) for each client. Without any resource coordination, query tasks may create a bottleneck in the system, leading to long query's response time, low resource utilization, and unfairness among different clients.

To address the aforementioned problem, this paper studies coordinated resource management in a multi-tenant cluster that supports interactive ad hoc queries over massive datasets. We adopt a utility-based optimization framework where the objective is to optimize resource utilization, coordinate among multiple resources from different machines, and maintain fairness among different clients.

Concretely, each client is associated with a utility, which corresponds to the query rate it is able to issue. The objective of the optimal resource allocation is to maximize the aggregate utility of all clients, subject to the cluster resource constraints. We solve this utility-based resource allocation problem via a price-based approach. Here, a "price" signal is associated with each type of resource for each machine. For each query, we: (1) collect resource prices from the machines where the query runs its fragments; (2) adjust a new query rate based on the updated prices such that the query's "net benefit", the utility minus the resource cost, is maximized. For each machine, we: (1) collect the new rates for queries that run fragments on current machine; (2) update the price for each type of resource based on the availability. The resource prices and query rates are updated iteratively. We prove that there exists a unique "maximum utility" rate allocation, at which point the cluster resource utilization is Pareto-optimal. Meanwhile, certain fairness objectives (e.g., max-min, and proportionality) can be achieved when we choose appropriate utility functions for queries.

The major contributions of this chapter are

- To the best of our knowledge, this is the first work that identifies and addresses the coordinated resource management problem for massively parallel data query in a clustering environment.

- From a theoretical perspective, this paper provides a model for concurrent queries that are executed in a distributed manner in a clustering environment and captures its performance using a utility-based resource management framework. This allows for a price-based solution which converges to the optimal point, at which the aggregate utility of all queries is maximized.

- From a practical perspective, we implement our proposed resource management solution over open source Impala system [19], and evaluate it in both simulated

environment and a real cluster using TPC-DS workload. Experimental results shows significant gain of our solution, in comparison to others.

Query Model

Example of Query Execution

To better understand the parallel query processing system, its resource usage pattern and resource management requirement, here we introduce a simple example with three tables shown in Figure V.1. Each table is divided into several data blocks, stored at different machines in the cluster. For example, in Figure V.4, data blocks belonging to the table *student* are stored in four machines $\{M_1, M_2, M_4, M_5\}$. Feasible data placement structures include row-stores, column-stores [8], and hybrid-stores [35].

There are two queries Q_1 and Q_2 in Figure V.2. The objective of query Q_1 is to *get the maximum score for each course*, and query Q_2 is to *retrieve the top-10 student names with the highest average scores*. As every machine in the cluster runs a coordinator, the client can connect to any machine and submit Q_1 or Q_2. The query planner at the selected machine compiles the submitted query into a query execution plan and chops that plan into several fragments. After retrieving the data location information from the metastore, the coordinator dispatches each query fragment to the machines that stores its input data.

Figure V.3 illustrates the query execution plans for Q_1 and Q_2. Here, we explain how Q_2's query plan works. Query Q_2 has three query fragments. The fragment F_{23} scans table *score*'s data blocks and broadcasts the results to machines that running fragment F_{22}. Fragment F_{22} first scans table *student*'s data blocks, and then joins the results with

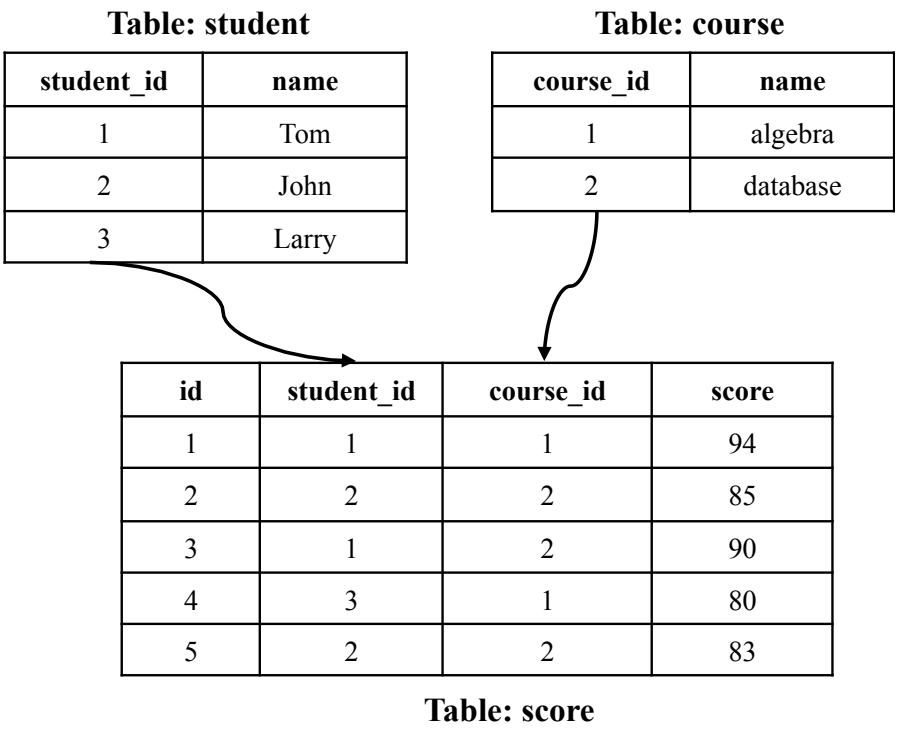


Figure V.1: An example database schema with tables *student*, *course* and *score*.

the data coming from F_{23}. All local join results are aggregated and returned to the query coordinator, where the fragment F_{21} works. Fragment F_{21} aggregates all local results, and return the final results to the client.

Figure V.4 illustrates an example how Q_1 and Q_2 execute in a cluster with five machines. Three tables are stored across the cluster. Query Q_1 chooses machine M_5 as its coordinator, while Q_2 chooses M_1. The query planner at M_1 calculates the execution plan for Q_2 and chops it into three fragments. And then the query coordinator dispatches these fragments to different machines based on the data location information. For example, fragment F_{22} is sent to machines $\{M_1, M_2, M_4, M_5\}$, as table *student*'s data blocks are stored in these machines.

```
Q1: SELECT course_id, max(score)
    FROM score
    GROUP BY course_id

Q2: SELECT student.name, avg(score.score) as ss
    FROM student, score
    WHERE student.student_id = score.student_id
    GROUP BY student.name
    ORDER BY ss DESC LIMIT 10
```

Figure V.2: Two example SQL queries Q_1 and Q_2.

Query Model

In this paper, we consider a set of m queries, denoted as $\mathbf{Q} = \{Q_1, Q_2, ..., Q_m\}$. Each query Q_i is submitted by a client to retrieve different data information, e.g., *monitoring the top-10 active users in the last 10 minutes*. To get the up-to-date information, Q_i needs to be submitted periodically. Let x_i represent the streaming rate of Q_i, e.g., 6 *query/hour* means Q_i is submitted every 10 minutes. Although Q_i can choose a different machine as its query coordinator each time, for performance stability and load balancing, we let each query always chooses the same machine and attach queries to machines evenly. We collect all streaming rates into a rate vector $\mathbf{x} = (x_i, 1 \leq i \leq m)$.

We consider the cluster containing n machines, denoted as $\mathbf{M} = \{M_1, M_2, ..., M_n\}$. As we discuss in Section V, each query is chopped into several query fragments, and each fragment runs across a set of machines. As a result, each query consumes a different amount of resources at each machine. For example, in Figure V.4, query Q_2 consumes some resources at M_1 as its fragments F_{21} and F_{22} run on M_1. Fragment F_{22} would consume disk I/O resource when scanning the table *student* data, and consume CPU, memory and network I/O resources when receiving table *course* data shuffled from other machines. As F_{21} needs to aggregate results coming from machines that run F_{22}, it would also consume some CPU, memory and network I/O resources.

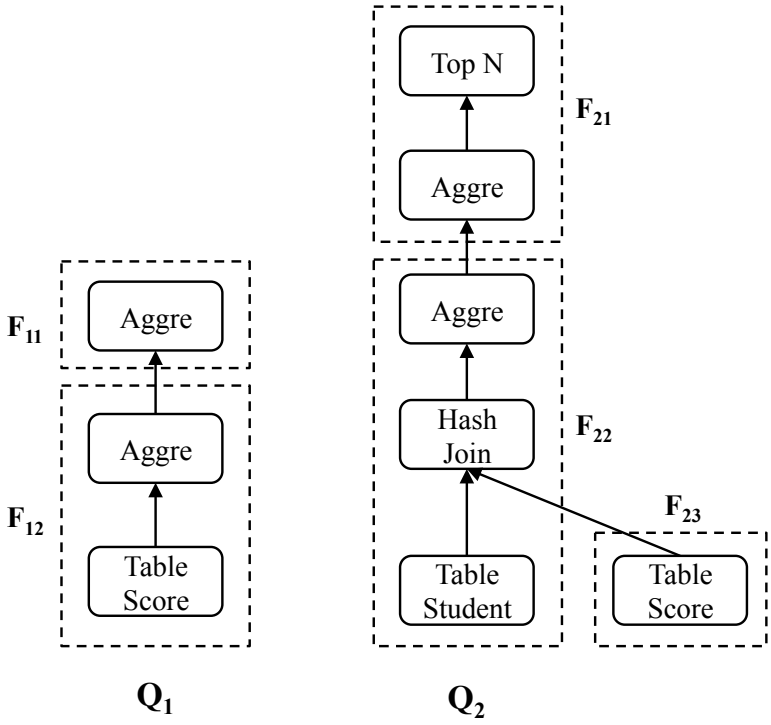


Figure V.3: Query execution plans for Q_1 and Q_2.

To differentiate different types of resources, for each machine M_j, we use a vector $C_j = (C_j^k, 1 \leq k \leq p)$ to describe its available resources. Here C_j^k represents the capacity of resource type k at M_j, and we consider p types of resources in total.

Now, we define a $\mathbf{Q} \times \mathbf{M}$ matrix $\mathbf{A}$, where $A_{ij} = (A_{ij}^k, 1 \leq k \leq p)$ represents the resource vector requested by query Q_i at machine M_j. $\mathbf{A}$ gives the resource usage pattern of queries. It follows that the aggregate resource usages of all queries that run on machine M_j should not exceed its resource capacity C_j. For example, in Figure V.4, assume M_2 has 16 GB memory, one Q_1 needs 4 GB and one Q_2 need 6 GB, the cluster cannot run two Q_1 and two Q_2 queries at the same time, as it would overflow M_2's memory resource.

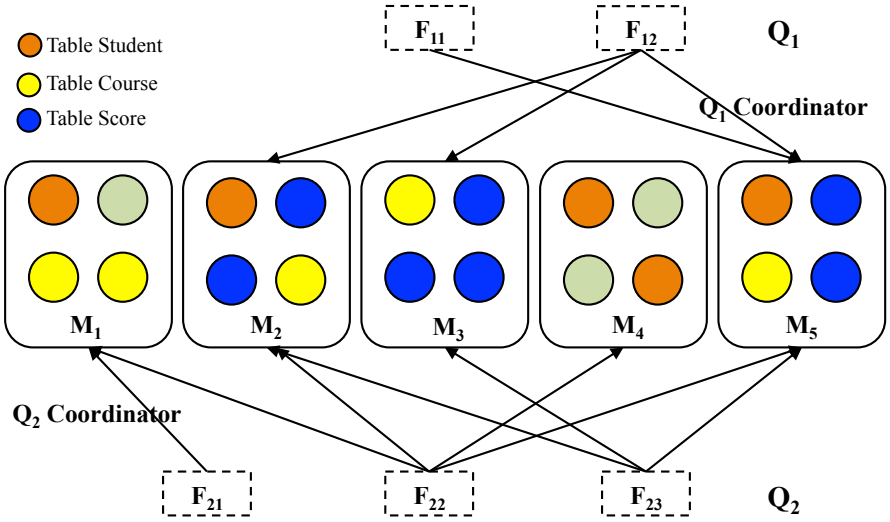


Figure V.4: Examples of how query fragments execute in the cluster.

Formally, such a capacity constraint is expressed as follows:

$$\sum_{i=1}^{m} A_{ij} \cdot x_i \leq C_j, \forall 1 \leq j \leq n \tag{V.1}$$

Here we explain the query model using the example discussed in Section V. According to Figure V.4, we assume the following resource consumption matrix **A** with $p = 2$ (i.e., *CPU* and *memory*) for example queries Q_1 and Q_2:

$$A_1 = \begin{pmatrix} 0 & 0 \\ 1 & 2 \\ 1 & 3 \\ 0 & 0 \\ 5 & 3 \end{pmatrix} \quad A_2 = \begin{pmatrix} 4 & 3 \\ 1 & 3 \\ 1 & 3 \\ 2 & 2 \\ 2 & 3 \end{pmatrix}$$

We also assume each machine (M_1-M_5) has $C_j = (10, 10), 1 \leq j \leq 5$.[1]

[1]Note that each resource is illustrated as aggregate resource. That is, for a machine with 10 GB memory, its memory resource is described as 10 $GB - s$. For a query consumes 5 GB memory and runs 2 second at a machine, its memory request at that machine is 10 $GB - s$.

If the example cluster only runs Q_1, the maximum streaming rate for Q_1 is 2. The value is bounded by the CPU resource at machine M_5, as Q_1 consume 5 CPU resource at M_5.

Optimal Resource Allocation

In this section, we first formulate the resource allocation problem, and then illustrate our optimal algorithm.

Problem Formulation

We associate each query $Q_i \in \mathbf{Q}$ with a utility U_i defined as a function of its streaming rate x_i. We make the following assumptions about $U_i(x_i)$.

- **A1**. On the interval $I_i = [x_i^{\alpha}, x_i^{\beta}]$, the utility function $U_i(x_i)$ is increasing, strictly concave, and twice continuously differentiable.
- **A2**. The curvatures of U_i are bounded away from zero on I_i: $-U_i''(x_i) \geq 1/\kappa_i > 0$.
- **A3**. U_i is additive, so that the aggregated utility of rate allocation $\mathbf{x} = (x_i, Q_i \in \mathbf{Q})$ is $\sum_{i=1}^{m} U_i(x_i)$.

The goal of resource allocation in the cluster is to make resource allocation decision $\mathbf{x}$ wisely, so that the aggregate utility $\sum_{i=1}^{m} U_i(x_i)$ is maximized. So the optimal resource allocation problem can be formulated as the following constrained nonlinear optimization problem:

$$\textbf{P: maximize} \sum_{i=1}^{m} U_i(x_i), \tag{V.2}$$

$$\textbf{subject to} \sum_{i=1}^{m} A_{ij} \cdot x_i \leq C_j, \forall 1 \leq j \leq n, \tag{V.3}$$

$$\textbf{over } x_i \in I_i. \tag{V.4}$$

We now demonstrate that, by optimizing toward such an objective, both *optimal resource utilization* and *certain fairness objectives* can be achieved among all queries.

Pareto Optimality

With respect to optimal resource utilization, we show that the resource allocation is *Pareto optimal* if the optimization problem **P** can be solved. Formally, the Pareto optimality is defined as follows:

Definition 2 ***Pareto optimality***. *A rate allocation $\boldsymbol{x} = (x_i, Q_i \in \mathbf{Q})$ is Pareto optimal if it satisfies the following two conditions: 1) $\boldsymbol{x}$ is feasible, i.e., $\boldsymbol{x} \geq 0$ and V.3 holds, and 2) $\forall \boldsymbol{x}'$ which is feasible, if $\boldsymbol{x}' \geq \boldsymbol{x}$, then $\boldsymbol{x}' = \boldsymbol{x}$. In the second condition, the $\geq$ is defined such that, two vectors $\boldsymbol{x}$ and $\boldsymbol{x}'$ satisfy $\boldsymbol{x}' \geq \boldsymbol{x}$, if and only if for all $Q_i \in \mathbf{Q}$, $x_i' \geq x_i$.*

Proposition 4 *A rate allocation $\boldsymbol{x}$ is Pareto optimal, if it solves the problem **P**, with increasing and strictly concave utility functions $U_i(x_i)$, for $Q_i \in \boldsymbol{Q}$.*

Proof 6 *Let $\boldsymbol{x}$ be a solution of the problem **P**. If $\boldsymbol{x}$ is not Pareto optimal, then there must exist another solution $\boldsymbol{x}' \neq \boldsymbol{x}$, which satisfies constraint V.3 and $\boldsymbol{x}' > \boldsymbol{x}$. As the utility function $U_i(x_i)$ is increasing and strictly concave, we have $\sum_{i=1}^{m} U_i(x_i') > \sum_{i=1}^{m} U_i(x_i)$. This leads to a contradiction, as $\boldsymbol{x}$ is the solution to **P** and hence maximizes $\sum_{i=1}^{m} U_i(x_i)$.*

Fairness

By choosing appropriate utility functions, the optimal resource allocation can implement different fairness models among the queries. We next illustrate this fact using two commonly adopted fairness models: *weighted proportional* and *max-min fairness*.

Definition 3 ***weighted proportional fairness***. *A vector of rates* $\boldsymbol{x} = (x_i, Q_i \in \mathbf{Q})$ *is weighted proportional fair with the vector of weights* w_i *if it satisfies the following two conditions: 1)* $\boldsymbol{x}$ *is feasible, and 2) for any other feasible vector* $\boldsymbol{x}' = (x_i', Q_i \in \mathbf{Q})$*, the aggregated of proportional changes is zero or negative:*

$$\sum_{i=1}^{m} w_i \frac{x_i' - x_i}{x_i} \leq 0 \tag{V.5}$$

Proposition 5 *A rate allocation* $\boldsymbol{x}$ *is weighted proportional fair with the weight vector* w_i*, if and only if it solves the problem* $\boldsymbol{P}$*, with* $U_i(x_i) = w_i \log x_i$ *for* $Q_i \in \mathbf{Q}$.

As shown in [42], by the optimality condition V.2, this proposition can be derived according to the following relation:

$$\sum_{i=1}^{m} \frac{\partial U_i}{\partial (x_i)}(x_i)(x_i' - x_i) = \sum_{i=1}^{m} w_i \frac{x_i' - x_i}{x_i} < 0 \tag{V.6}$$

which is strict inequality follows from the strict concavity of $U_i(x_i)$.

Definition 4 ***max-min fairness***. *A vector of rates* $\boldsymbol{x} = (x_i, Q_i \in \mathbf{Q})$ *is max-min fair if it satisfies the following two conditions: 1)* $\boldsymbol{x}$ *is feasible, and 2) for any* $Q_i \in \mathbf{Q}$*, increasing* x_i *cannot be achieved without decreasing the fair share* $x_{i'}$ *of another query* $Q_{i'} \in \mathbf{Q}$ *that satisfies* $x_i \geq x_{i'}$.

Proposition 6 *A rate allocate* $\boldsymbol{x}$ *is max-min fair if and only of it solves the problem* $\boldsymbol{P}$*, with* $U_i(x_i) = -(-\log x_i)^{\theta}$*,* $\theta \to \infty$ *for* $Q_i \in \mathbf{Q}$.

Again, these results straightforwardly follow their counterparts in [42]. The remainder of this paper largely seeks to solve the optimal resource allocation problem **P** with the given utility function.

By assumption **A1**, the objective function V.2 is differentiable and strictly concave. Also, the feasible region of constraint V.3 is compact. By nonlinear optimization theory, there exists a maximizing value of argument **x** for the above optimization problem, which can be solved by Lagrangian method. Let us consider the Lagrangian form of this optimization problem:

$$L(\mathbf{x}, \boldsymbol{\mu}^k) = \sum_{i=1}^{m} U_i(x_i) - \sum_{k=1}^{p} \mu^k (A^k \cdot \mathbf{x} - C^k). \tag{V.7}$$

Here $\boldsymbol{\mu}^k = (\mu_j^k, M_j \in \mathbf{M})$ is the vector of Lagrangian multipliers. Equation V.7 can be further derived as follows:

$$\begin{aligned} L(\mathbf{x}, \boldsymbol{\mu}^k) &= \sum_{i=1}^{m} U_i(x_i) - \sum_{j=1}^{n} \sum_{k=1}^{p} \mu_j^k (\sum_{i=1}^{m} A_{ij} x_i - C_j^k) \\ &= \sum_{i=1}^{m} U_i(x_i) - \sum_{i=1}^{m} x_i \sum_{j=1}^{n} \sum_{k=1}^{p} \mu_j^k A_{ij} + \sum_{j=1}^{n} \sum_{k=1}^{p} \mu_j^k C_j^k \end{aligned} \tag{V.8}$$

We then define new vectors $\boldsymbol{\lambda}^k = (\lambda_i^k, 1 \leq i \leq n)$, where $1 \leq k \leq p$ as follows:

$$\lambda_i^k = \sum_{j=1}^{n} \mu_j^k A_{ij} \tag{V.9}$$

Now, V.8 becomes

$$\begin{aligned} L(\mathbf{x}, \boldsymbol{\mu}^k) &= \sum_{i=1}^{m} U_i(x_i) - \sum_{i=1}^{m} x_i \sum_{k=1}^{p} \lambda_i^k + \sum_{j=1}^{n} \sum_{k=1}^{p} \mu_j^k C_j^k \\ &= \sum_{i=1}^{m} U_i(x_i) - \sum_{k=1}^{p} \boldsymbol{\lambda}^k \mathbf{x} + \sum_{k=1}^{p} \boldsymbol{\mu}^k C. \end{aligned} \tag{V.10}$$

For $\boldsymbol{\mu}^k$, μ_j^k is the price of resource k at machine M_j. Consequently, for $\boldsymbol{\lambda}^k$, λ_i^k is the summation of prices of all machines that Q_i has assigned tasks, or in other words, the

price of each type of resource that Q_i has to pay. This vector corresponds to the resource constraint stated in V.3.

The vector of prices $(\mu^1, \mu^2, ..., \mu^p)$ will be used as incentives so that localized self-optimizing decision can implement the global optimum.

Resource Allocation Problem

Solving the objective function V.2 requires global coordination of all queries. Here we first look at the dual problem of **P** as follows:

$$\mathbf{D}: \min_{\mu^k \geq 0, 1 \leq k \leq p} D(\mu^k, 1 \leq k \leq p). \tag{V.11}$$

where

$$\begin{aligned} D(\boldsymbol{\mu}^k, 1 \leq k \leq p) &= \max_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\mu}^k) \\ = \max_{\mathbf{x}} \sum_{i=1}^{m} (\underbrace{U_i(x_i) - \sum_{k=1}^{p} \lambda_i^k x_i}_{\Phi(x_i)}) &+ \sum_{j=1}^{n} \sum_{k=1}^{p} \mu_j^k C_j^k. \end{aligned} \tag{V.12}$$

Since λ_i^k is the price for resource k of query Q_i, it is clear that $\sum_{k=1}^{p} \lambda_i^k x_i$ is the overall cost for Q_i. Then, $\Phi(x_i)$ is Q_i's "benefit", i.e., the difference of its utility and cost. By the separation nature of Lagrangian form, maximizing $L(\mathbf{x}, \boldsymbol{\mu}^k)$ can be decomposed into separately maximizing $\Phi(x_i)$ for each query $Q_i \in \mathbf{Q}$. Now, we have

$$D(\boldsymbol{\mu}^k, 1 \leq k \leq p) = \sum_{i=1}^{m} \max_{x_i \in I_i} \{\Phi(x_i)\} + \sum_{j=1}^{n} \sum_{k=1}^{p} \mu_j^k C_j^k. \tag{V.13}$$

By Assumption **A1**, U_i is strictly concave and twice continuously differentiable. Therefore, a unique maximizer of $\Phi(x_i)$ exists when

$$\frac{d\Phi(x_i)}{dx_i} = U_i'(x_i) - \sum_{k=1}^{p} \lambda_i^k = 0. \tag{V.14}$$

We define the maximizer as below:

$$x_i(\boldsymbol{\mu}^k, 1 \leq k \leq p) = \arg\max_{x_i \in I_i}\{\Phi(x_i)\} = [U_i'^{-1} \sum_{k=1}^{p} \lambda_i^k]_{x_i^\alpha}^{x_i^\beta}. \tag{V.15}$$

By Assumption **A1**, $I_i = [x_i^\alpha, x_i^\beta]$ is the feasible region of $U_i(x_i)$. Therefore, x_i must be no greater than x_i^β and no less than x_i^α. Since U_i is concave and the constraint V.3 is linear. there is no duality gap. Also, the optimal prices for Lagrangian multipliers $(\boldsymbol{\mu}^k, 1 \leq k \leq p)$ exist, denoted as $(\boldsymbol{\mu}^{k*}, 1 \leq k \leq p)$. If $(\boldsymbol{\mu}^{k*} \geq 0, 1 \leq k \leq p)$ are optimal, then $x_i(\boldsymbol{\mu}^{k*}, 1 \leq k \leq p)$ is also primal optimal, given that x_i is primal feasible.

Now, we can claim that once the optimal prices $(\boldsymbol{\mu}^{k*}, 1 \leq k \leq p)$ are available, the optimal rate x_i^* can be achieved by solving V.15. The role of $(\boldsymbol{\mu}^{k*}, 1 \leq k \leq p)$ is two-fold. First, they serve as the pricing signal for a query Q_i to adjust its rate x_i. Second, they decouple the primal problem **P** (global utilization optimization) into individual rate optimization by each query $Q_i \in \mathbf{Q}$.

Optimal Resource Allocation Algorithm

We solve the problem **D** using the gradient projection method. In this method, $(\boldsymbol{\mu}^k, 1 \leq k \leq p)$ are adjusted in opposite direction to the gradient $\triangledown D(\boldsymbol{\mu}^k, 1 \leq k \leq p)$:

$$\mu_j^k(t+1) = [\mu_j^k(t) - \gamma \frac{\partial D(\boldsymbol{\mu}^k(t), 1 \leq k \leq p)}{\partial \mu_j^k}]^+, 1 \leq k \leq p. \tag{V.16}$$

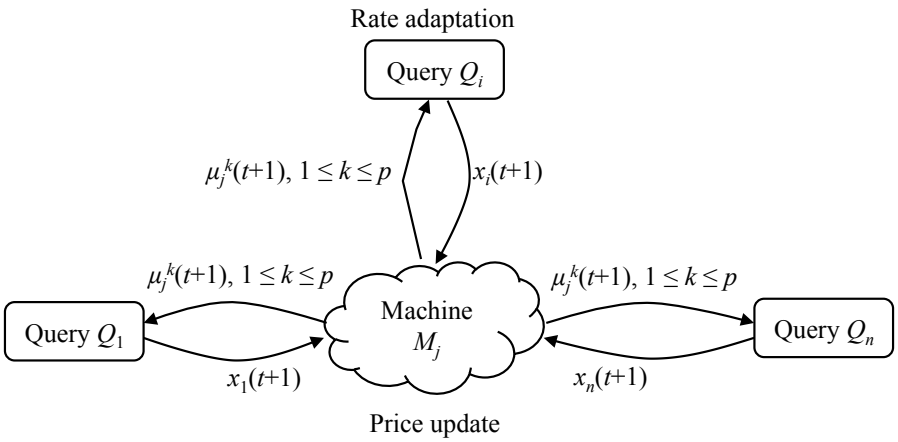


Figure V.5: The iterative process of resource price update.

γ is a step size. Substituting V.15 into V.14, we have

$$\begin{aligned} D(\boldsymbol{\mu}^k, 1 \le k \le p) = \sum_{i=1}^{m} (U_i(x_i(\boldsymbol{\mu}^k, 1 \le k \le p)) \\ - \sum_{k=1}^{p} \lambda_i^k x_i(\boldsymbol{\mu}^k, 1 \le k \le p)) + \sum_{j=1}^{n} \sum_{k=1}^{p} \mu_j^k C_j^k. \end{aligned} \tag{V.17}$$

$D(\boldsymbol{\mu}^k, 1 \le k \le p)$ is continuously differentiable since U_i is strictly concave. Thus, it follows that

$$\frac{\partial D(\boldsymbol{\mu}^k, 1 \le k \le p)}{\partial \mu_j^k} = C_j - \sum_{i=1}^{m} x_i(\boldsymbol{\mu}^k, 1 \le k \le p), 1 \le k \le p \tag{V.18}$$

Substituting V.18 into V.16, we have

$$\mu_j^k(t+1) = [\mu_j^k(t) + \gamma(\sum_{i=1}^{m} x_i(t) A_{ij} - C_j^k)]^+, 1 \le k \le p. \tag{V.19}$$

Equation V.19 reflects the law of supply and demand. If the demand for resource at machine M_j exceeds its supply C_j, the resource constraint is violated. Thus, the resource price $(\mu_j^k, 1 \le k \le p)$ is raised. Otherwise, $(\mu_j^k, 1 \le k \le p)$ is reduced.

Algorithm 7 Resource price update by machine M_j: at times t=1,2,...

1: Receive rates $x_i(t)$ from all queries $Q_i \in \mathbf{Q}$
2: // update price for each type of resource
3: **for** $k = 1 \rightarrow p$ **do**
4: $\quad \mu_j^k(t+1) = [\mu_j^k(t) + \gamma(\sum_{i=1}^{m} x_i(t)A_{ij} - C_j^k)]^+$
5: **end for**
6: Send $(\mu_j^k(t+1), 1 \leq j \leq p)$ to all queries $Q_i \in \mathbf{Q}$

Algorithm 8 Query rate adaptation by query Q_i: at times t=1,2,...

1: Receive resource prices $(\mu_j^k(t), 1 \leq k \leq p)$ from $(M_j, 1 \leq j \leq n)$
2: **for** $k = 1 \rightarrow p$ **do**
3: $\quad \lambda_i^k \leftarrow \sum_{j=1}^{n} \mu_j^k A_{ij}^k$
4: **end for**
5: // adjust rate
6: $x_i(t+1) \leftarrow x_i(\lambda_i^k, 1 \leq k \leq p)$
7: Send $x_i(t+1)$ to all machines $M_j \in \mathbf{M}$

This price-based iterative solution can be interpreted as follows. Each machine sets its resource price vector μ_j, under which each query will maximize its surplus to use resource. Based on all the returned x_i, each machine aims to update the price vector iteratively, such that **x** produced by the surplus-maximizing queries will eventually converge to the optimal resource allocation.

Such a process is illustrated in Figure V.5, and the Algorithm 7 and 8 illustrate the implementation details of the resource price update and query rate adaptation.

Simulation Results

In this section, we present the simulation results of our optimal resource management algorithm to verify the design. We assume 10 queries $\mathbf{Q} = \{Q_1, ..., Q_{10}\}$, and 5 machines $\mathbf{M} = \{M_1, ..., M_5\}$. We consider two types of resources ($p = 2$) in our simulation, representing CPU and memory. The resource capacity for each machine is

Algorithm	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	$\sum_{i=1}^{10} U_i(x_i)$
Optimal	748	387	569	704	450	636	688	418	1000	660	64.02
Fair	568	568	568	568	568	568	568	568	568	568	63.42
FIFO	560	640	548	542	549	570	585	573	549	547	63.40

Table V.1: Query rate and utility comparison for the simulation workload with $w_i = 1$ for queries.

(16, 64), representing 16 CPU cores and 64 GB memory. We simulate a 1-hour time period, where the per-machine aggregate resource is 57,600 *core-s* and 230,400 *GB-s*. For the per-machine resource requirement for each query, we randomly generate values between (1 *core-s*, 1 *GB-s*) and (16 *core-s*, 64 *GB-s*). The minimum and maximum rate requirements of queries are $\{x_i^{\alpha} = 1$ *query/hour* and $x_i^{\beta} = 1000$ *query/hour*$\}$, for all Q_i. It is obvious that the minimum rate requirement can be guaranteed.

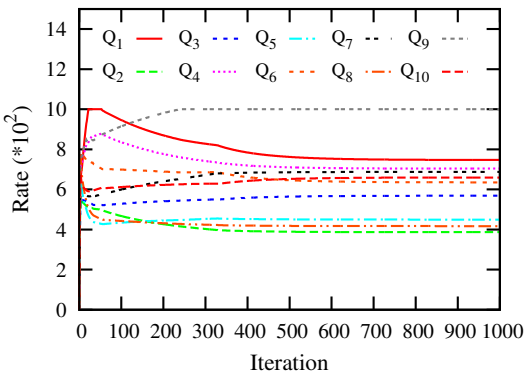
Convergence

We first illustrate the convergence of the price update mechanism. Here the utility function of each query Q_i is set as $U_i(x_i) = w_i \log x_i$, where represents weighted proportional fairness. We set $w_i = 1$ for all queries, and the step size $\gamma = 2 \times 10^{-11}$.

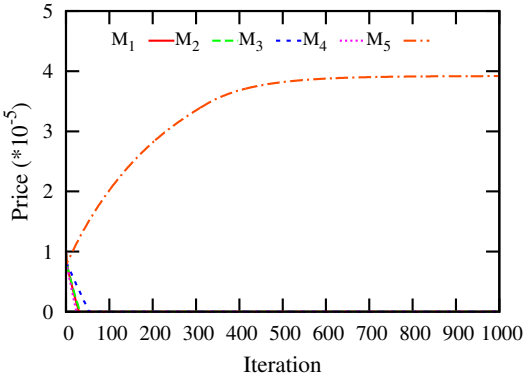
As shown in Figure V.6, the algorithm converges to a global cluster equilibrium within around 600 iterations. The final optimal rates of all queries (labeled as *Optimal*) are shown in Table V.1.

Performance Comparisons

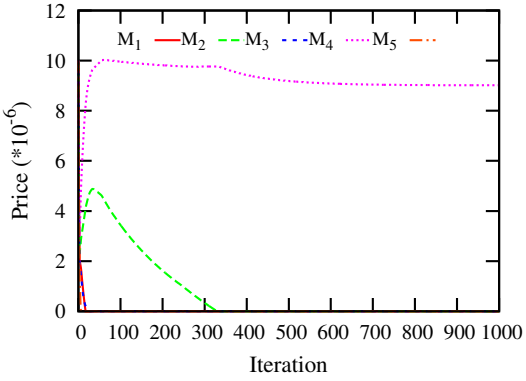
In this set of experiments, we show how our algorithm (*Optimal*) achieves better performance in comparison with fair sharing and FIFO mechanisms. For the fair sharing mechanism (*Fair*), we refer to the solution which assigns an equal streaming rate for all



(a) Query rate for $Q_i \in \mathbf{Q}$



(b) CPU price for $M_j \in \mathbf{M}$



(c) Memory price for $M_j \in \mathbf{M}$

Figure V.6: Convergence of the *Optimal* algorithm on simulation workload with $w_i = 1$ for all queries.

queries. Under the given resource constraints for the given example cluster setup, the maximum allowed streaming rate for each query is 568 *query/hour*. For the latter FIFO mechanism (*FIFO*), we build a query generator which randomly selects a query from the query candidate set (including all queries initially). The randomly selected query will be adopted by the cluster only if it requests less than the available resource[2]; otherwise, we remove this query from the candidate set. The generator continuously generates queries until the candidate set is empty.

Table V.1 illustrates the comparison results for these three algorithms. The aggregate utilities of the *Fair* and *FIFO* mechanisms are 63.42 and 63.40, which are suboptimal to the result 64.02 of our *Optimal* mechanism.

Higher aggregate utility also means high cluster resource utilization ratio. We calculate the cluster CPU/memory utilization for all mechanisms. The numbers for our *Optimal* solution are 79.3% CPU usage and 93.8% memory usage, which are higher than the *Fair* (71.2% CPU usage and 86.8% memory usage) and *FIFO* (71.4% CPU usage and 86.6% memory usage).

Weighted Workload

Different from the above experiments that assume $w_i = 1$ for all queries, here we attach a different weight for each query. The first third of the queries have $w_i = 1$, the next third has 2, and the last third has 3. We re-run the three mechanisms (*Optimal*, *Fair*, and *FIFO*). Table V.2 illustrates the results. Under the weighted setup, our *Optimal* algorithm still performs much better than the *Fair* and *FIFO*, in terms of aggregate utility, and cluster resource utilization.

[2]For the example illustrated in Section V, if the cluster already accepts two Q_2, it cannot accept Q_2 anymore as the M_1 only has 2 $core - s$, while each Q_2 requests 4 $core - s$ at M_1.

Algorithm	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	$\sum_{i=1}^{10} U_i(x_i)$
Optimal	552	237	234	867	410	751	814	640	1000	799	136.31
Fair	265	265	265	530	530	530	795	795	795	795	134.52
FIFO	585	577	570	583	586	586	578	564	570	530	133.24

Table V.2: Query rate and utility comparison for the simulation workload with various weights for queries.

Evaluation

In this section, we explore the performance of our optimal resource allocation algorithm using a TPC-DS workload [1] on a real cluster.

Setup

Hardware configuration

For the experiments presented in this section, we use a 11 machine cluster. One of the machines hosts the metastore. The remaining 10 machines are designed as "compute" machines. Each machine in the cluster is configured with 4 CPU cores and 8 GB memory.

Software configuration

For our experiments, we use Impala version 1.4.0 on top of CDH-5.1. Each compute machine runs an *impalad* daemon, which accepts query requests and coordinate query executions.

Workload

Our workload is a variant of the Transaction Processing Performance Council's decision-support benchmark (TPC-DS) [79]. The TPC-DS benchmark was designed

to model multiple users running a variety of decision-support queries including reporting, interactive OLAP, and data mining queries. All of the users run in parallel; each user run the queries in series in a random order. The benchmark models data from a retail product supplier about product purchases. We use a subset of 20 queries [1] that was selected in an existing industry benchmark, published by Impala developers. For better readability, here we rename the query able to Q_1 to Q_{20}. We use a TPC-DS database with a scale factor of 100 GB. We were not able to scale to larger TPC-DS datasets because of Impala's limitation to require the workload's working set to fit in the cluster's aggregate memory.

Query Profiling

As we discussed in Section V, each query is decomposed into several fragments, and each fragment is dispatched to the machines containing the corresponding data blocks. Here we first profile the resource consumption at different machines for each query.

We submit each TPC-DS query independently, and collect the peak CPU/memory consumption and running time for that query at different machines, whose product can be measured as the aggregate resource consumption. For example, for a query consuming 2 GB memory and running 10 seconds at a particular machine, its memory resource consumption at that machine is 20 *GB-s*. Although each query can randomly select one machine as its query coordinator, we choose a pre-determined machine for each query in order to profile the resource consumption during coordination. Figure V.7 illustrates the normalized aggregate CPU/memory consumption across all machines. Results show that different queries require different amount of CPU/memory resources. This is because that some queries need to process large amount of data blocks, while some others only need to process fewer.

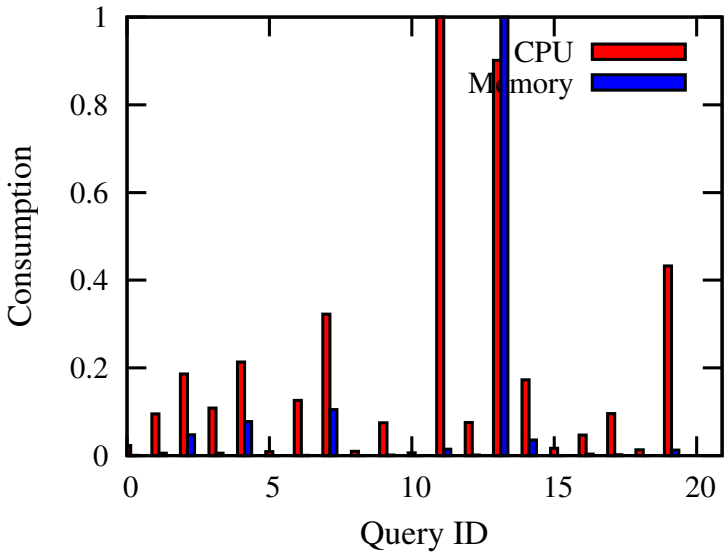


Figure V.7: Normalized aggregate CPU/memory consumption for $Q_i \in \mathbf{Q}$ for TPC-DS workload.

Rate Convergence

We first evaluate the performance of our optimal resource allocation algorithm at converging to the optimal point. Here we use the resource requests profiled in Section V. We set the minimum and maximum rate requirements to 1 *query/hour* to 300 *query/hour*, and the step size is 10^{-7}. As shown in Figure V.8, the algorithm converges to a global resource equilibrium within about 200 iterations, and the streaming rate for each query becomes stable.

Performance Comparisons

Similar to the simulation experiments presented in Section V, here we deploy a set of comparison experiments with other different mechanisms. Note that in current Impala implementation, the client can issue as many queries as possible to the cluster. However, some queries may be canceled due to resource oversubscription. For example, if too

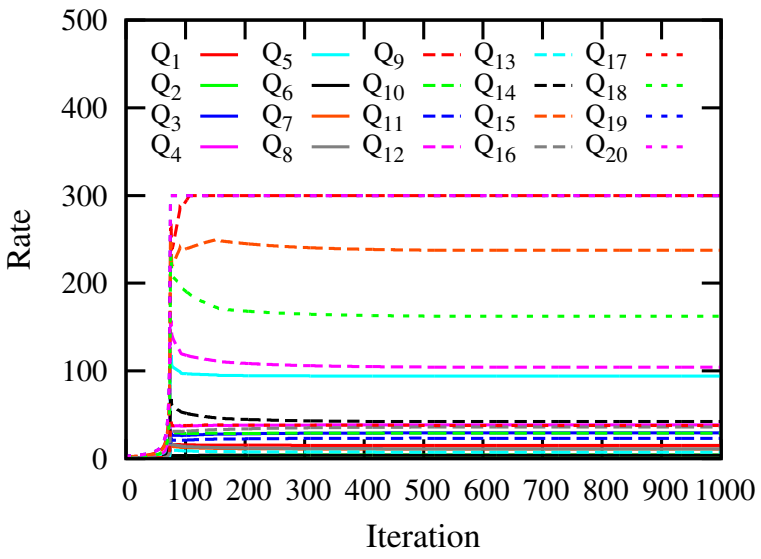


Figure V.8: Query rates for $Q_i \in \mathbf{Q}$ for TPC-DS workload.

many queries are dispatched to one machine, and that machine cannot provide enough memory space to run all, some queries will be marked as failed and canceled.

We conduct experiments using *Optimal*, *Fair* and *FIFO*.

- For *Optimal*, we use the streaming rate calculated in Section V and illustrated in Figure V.8.
- For *Fair*, we let each query client submit their queries with an equal streaming rates. For example, $Fair(10)$ represents that all query clients can submit their queries 10 times in 1-hour interval.
- For *FIFO*, we build a query generator which randomly select one query from $\{Q_1, ..., Q_{20}\}$ and submit to the cluster, with an interval of 10 per minute.

For each group experiment, we keep the cluster running for 1 hour, and record the number of queries that have finished successfully. We only count the successful finished queries when calculating the streaming rate for each query. Table V.3 illustrates the actual streaming rate for each query. Results show that our *Optimal* achieves a higher

average streaming rate than *Fair* and *FIFO*. And for *Fair*, we can also find out that simply increasing the issuing rate cannot get a higher streaming rate, i.e., *Fair(300)* performs worse than *Fair(30)*. This is because in Impala, too many queries submitted may cause some machines overloaded (e.g., *out of memory*); and if that happens, all queries execute at those machines would be canceled and marked as failed.

We also calculate the aggregate utility for each group experiment, and Figure V.9 presents the results. Our *Optimal* receives a better utility than the others.

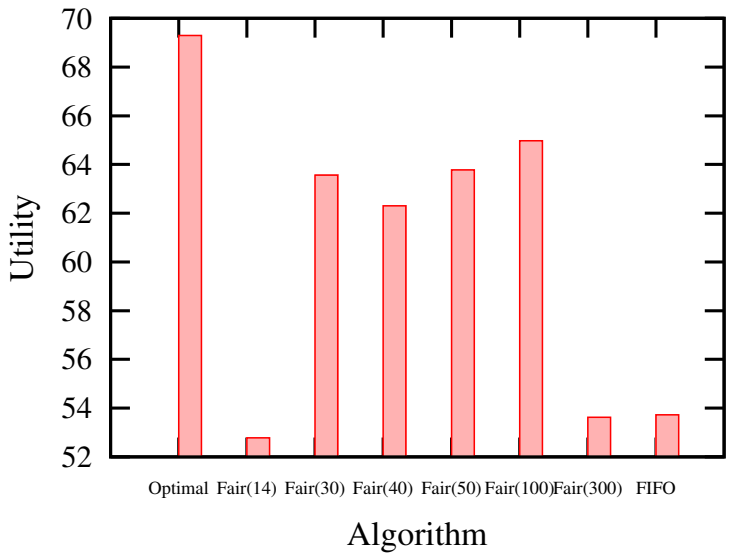


Figure V.9: Aggregate utility for $\sum_{i=1}^{20} U_i(x_i)$ for TPC-DS workload.

Data Placement Structure

In database area, columnar data organization is always introduced to reduce disk I/O, and enable better compression and encoding schemes that significantly benefit analytical queries [77, 2]. Impala has also implemented its own columnar storage format,

namely Parquet [8], and the aforementioned TPC-DS benchmark has been already optimized using Parquet. For better illustrating our optimal resource allocation algorithm with different setup, we build another TPC-DS benchmark by removing this columnar optimization. We name this new benchmark as TPC-DS2.

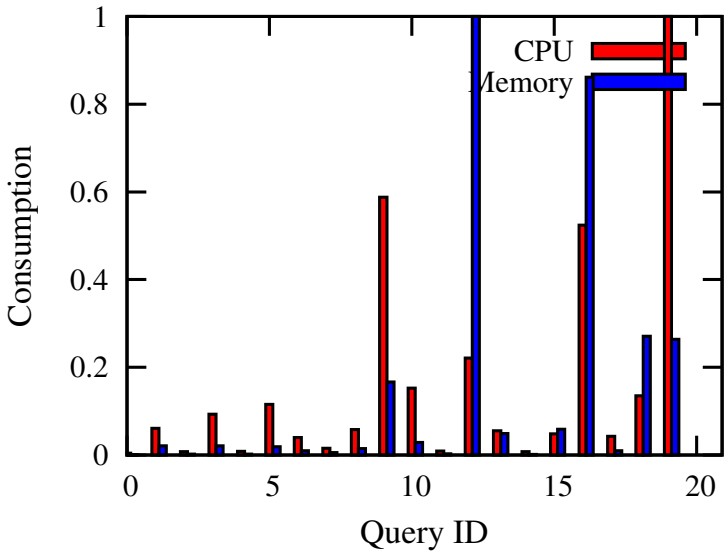


Figure V.10: Normalized aggregate CPU/memory consumption for $Q_i \in \mathbf{Q}$ for TPC-DS2 workload.

We re-measure the per-machine resource consumption for each query, illustrated in Figure V.10. We run our optimal resource allocation algorithm using the new profiled results, and reset the I_i to [1 *query/hour*, 100 *query/hour*]. The query rate convergence process is shown in Figure V.11.

We also deploy the comparison experiments to compare our *Optimal* algorithm with the *Fair* and *FIFO*. Table V.4 records the streaming rates. Compared to Table V.3, the rates have been largely reduced. This is because that, without columnar optimization, each query takes more time to finish. Figure V.12 illustrates the aggregate utility, and our *Optimal* mechanism performs much better than others.

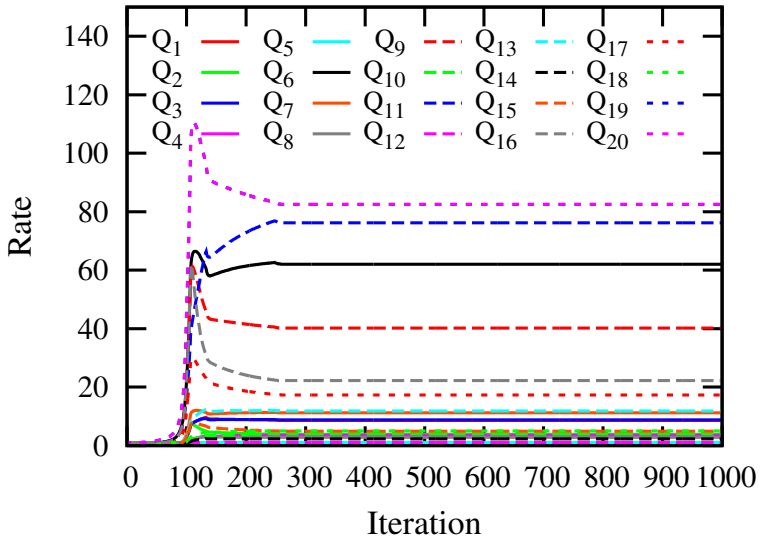


Figure V.11: Query rates for $Q_i \in \mathbf{Q}$ for TPC-DS2 workload.

Chapter Summary

In this chapter, we target the problem of optimal resource allocation for massively parallel data query. We model this problem as an utility maximization problem, and present a pricing-base solution. It is proven that our algorithm can converge to an optimal point, at which the aggregate utility is maximized. We implement out algorithm in the open source Impala system and conduct a set of experiments in a clustering environment using the TPC-DS workload. Experimental results show that our coordinated resource management solution can increase the aggregate utility by at least 15.4% compared with simple fair resource share mechanism, and 63.5% compared with the FIFO resource management mechanism.

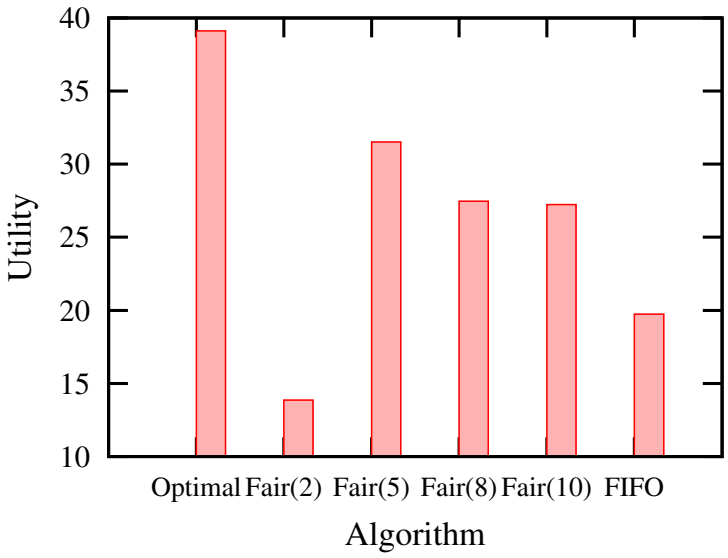


Figure V.12: Aggregate utility for $\sum_{i=1}^{20} U_i(x_i)$ for TPC-DS2 workload.

Algorithm	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}	x_{20}
Optimal	93	28	15	29	10	300	23	7	237	37	300	4	38	3	11	104	42	35	162	7
Fair(14)	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
Fair(30)	26	18	22	27	20	30	26	21	30	28	29	17	28	17	26	27	28	27	30	14
Fair(40)	27	10	25	28	17	39	26	15	29	26	36	19	25	8	37	35	31	28	34	6
Fair(50)	15	13	46	37	14	30	30	8	49	15	46	33	40	5	32	29	14	43	46	26
Fair(100)	12	9	20	46	21	8	18	29	37	54	14	180	24	12	32	63	79	36	57	4
Fair(300)	25	9	42	27	15	22	16	11	31	14	31	2	30	1	2	80	49	30	69	0
FIFO	13	13	12	11	19	16	13	12	12	16	15	16	12	15	13	22	14	20	18	17

Table V.3: Query rate comparison for TPC-DS workload.

Algorithm	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}	x_{20}
Optimal	8	3	8	3	1	62	11	2	40	3	76	1	11	2	4	22	17	5	82	1
Fair(2)	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Fair(5)	4	5	5	5	5	5	4	5	5	5	5	4	5	5	5	5	5	5	5	5
Fair(8)	1	4	6	4	3	6	5	5	6	5	6	0	5	4	5	6	6	5	6	1
Fair(10)	1	6	8	4	5	7	4	3	8	4	6	0	4	6	4	5	3	6	5	0
FIFO	3	1	1	3	4	2	6	3	2	9	5	1	1	1	3	2	5	9	6	2

Table V.4: Query rate comparison for TPC-DS2 workload.

CHAPTER VI

CONCLUSION AND FUTURE WORK

Big data analysis has collected much attention recently and several large-scale data processing systems have been developed, as well as deployed, to support various data processing applications. Optimal resource management is an essential requirement to optimize resource utilization and job performance. The work presented in this dissertation provides fundamental results towards addressing the challenges involving resource management within a job and across multiple jobs.

Summary of Contributions

Resource management is a challenging problem in a large-scale processing system. As shown in this dissertation, it often requires significant effort to understand the job requirement (both in resource and performance) and allocate the cluster resources in an efficiently way. The underlying principle of the work presented in this dissertation is the use of a data-driven approach for resource management. That is, we first understand the input jobs better by profiling the input data and runtime resource consumption for each job/task, and then we use that profile to optimize the resource utilization. Here, we take a moment to summarize the contributions of this dissertation.

First, we studied the problem of optimal resource management for a single data processing job. We use MapReduce as our target application and optimize the job completion time with the given allocated resource. In the real world, data is often highly skewed, which may cause workload imbalance among parallel running tasks. In this dissertation, we study the problem of reduce-phase skew in MapReduce applications,

where reduce tasks are often assigned imbalance load (in terms of key groups). We proposed two techniques to manage skew in MapReduce.

- We introduce a sketch-based data structure for capturing MapReduce key group size statistics and present an optimal packing algorithm which assigns the key groups to the reducers in a load balancing manner. We perform an empirical evaluation with several real and synthetic datasets over two distinct types of applications. The results show that our load balancing algorithm can strongly mitigate the reduce-phase skew. It can decrease the overall job completion time by 45.5% of the default settings in Hadoop and by 38.3% in comparison to the state-of-the-art solution.

- The above sketch-based solution can help solve the skew problem for most MapReduce applications. However, it cannot be deployed to some complex MapReduce applications, whose task workload is not directly depended on the data size (like record linkage application). To solve this problem, we further study the sketch-based solution and perform load balancing mechanisms for record linkage application. We propose two load balancing algorithms to work over sketch-based profiles while solving the data skew problem associated with record linkage. We provide an analytical analysis and extensive experiments (using Hadoop), with real and controlled synthetic data sets, to illustrate the effectiveness of our solution. The experimental results show that our load balancing algorithms can decrease the overall job completion time by 71.56% and 70.73% of the default settings in Hadoop using a set of DBLP data sets, which have 2.5 to 50.4 million records.

Second, we studied the optimal resource management across multiple data processing jobs. We use interactive *ad hoc* query systems as our target application. Interactive

ad hoc data query systems becomes more and more popular, and allow users run queries directly on Hadoop-type systems. However, without carefully designed coordination, resource collisions may happen. As a result, query tasks may create system bottlenecks, leading to long query response times, low resource utilization, and unfairness across different clients.

We adopt a utility-based optimization framework to solve this coordinated resource management problem in a multi-tenant cluster that supports interactive ad hoc queries over massive datasets. The objective here is to optimize the cluster resource utilization, coordinate among multiple resources from different machines, and maintain fairness among different clients. Concretely, each client is associated with a utility, which corresponds to the query rate it is able to issue. The objective of the optimal resource allocation is to maximize the aggregate utility of all clients, subject to the cluster resource constraints.

We solve this utility-based resource allocation problem via a price-based approach. Here, a "price" signal is associated with each type of resource (e.g., CPU, memory) for each machine. For each query, we: (1) collect resource prices from the machines where the query runs its fragments; (2) adjust a new query rate based on the updated prices such that the query's "net benefit", the utility minus the resource cost, is maximized. For each machine, we: (1) collect the new rates for queries that run fragments on current machine; (2) update the price for each type of resource based on the availability. The resource prices and query rates are updated iteratively until converge. We implement our algorithm in the open source Impala system and conduct a set of experiments in a clustering environment using the TPC-DS workload. Experimental results show that our coordinated resource management solution can increase the aggregate utility by at least 15.4% compared with simple fair resource share mechanism, and 63.5% compared with the FIFO resource management mechanism.

Discussion and Future Directions

The work presented in this dissertation has initiated several discussions and motivated several further work in area of resource management in large-scale data processing systems. Here we mainly discuss the work which can help enhance our resource management framework move to production.

The most straightforward future work is to extend I/O bandwidth as a resource to the coordinated resource management framework for large-scale interactive ad hoc query systems. In Chapter V, we deploy a set of experiments which mainly consider CPU and memory as the resources to be allocated. However, when we move to a production cluster, disk and network bandwidth become critical, especially for the query systems. In such systems, each query task needs to read and write data from local disk, which consumes disk bandwidth. Additionally, data shuffling and aggregation happen between different concurrent query tasks, which requires network bandwidth. By taking disk and network bandwidth into consideration, the coordinated resource management framework presented in Chapter V can be strengthened and perform better in production clusters.

Besides the I/O bandwidth management, another key advancement needed for production clusters is to include the data change to the coordinated resource management framework. In production clusters, data is keeping updated. As presented in Chapter V, the approach deployed depends on the pre-profiled per-query resource consumption. That is, we collect the resource consumption for each query at different machines and utilize the associated profile to calculate the optimal resource allocation. Such a mechanism has a limitation that, in production clusters, the data processed by each query keeps changing. Using a static, predetermined resource consumption profile may bring deviation to the resource allocation results. An online approach can be developed to solve

this limitation. That is, we track the resource consumption for each query in real time, and recalculate the optimal resource allocation every given interval. By always using the latest resource consumption profile, the algorithm can avoid an outdated profile and generate a more accurate resource allocation mechanism.

The resource allocation mechanism delivered in Chapter V provides a maximum streaming rate for each query. In most production clusters, the resources are shared among multiple departments instead of single queries. Each department is assigned a dedicated resource pool and all queries coming from one department will be submitted to the department's own resource pool. To ensure high performance of the algorithm in this scenario, we need to upgrade the solution from the query-level to the resource-pool level. That is, we need to build a resource consumption profile for each resource pool, instead of for a single query. The calculated results will determinate the number of queries accepted by each resource pool.

Another future work is to develop a universal resource management framework that supports various data processing frameworks. The work communicated in this dissertation is primarily designed to support resource management mechanisms for single-type applications, such as for MapReduce applications, for data query systems. However, the explosion in the complexity and variety of large-scale data processing systems has fueled a shift from single-purpose clusters that only support one type of workload to multi-purpose clusters running a mix of jobs. That is, batch computation (MapReduce, Pregel) and interactive query (Dremel/Impala) may run together. Such sharing environments present new resource management challenges. Different types of jobs use different metrics to measure their performances. For example, batch computation jobs are always measured by job completion time, why interactive query mostly cares about throughput. The new designed resource management framework should consider this performance diversity.

BIBLIOGRAPHY

[1] A TPC-DS like benchmark for Cloudera Impala. https://github.com/cloudera/impala-tpcds-kit.

[2] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[3] Foto N Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D Ullman. Vision paper: Towards an understanding of the limits of Map-Reduce computation. *arXiv preprint arXiv:1204.1754*, 2012.

[4] Ganesh Ananthanarayanan, Christopher Douglas, Raghu Ramakrishnan, Sriram Rao, and Ion Stoica. True Elasticity in Multi-tenant Data-intensive Compute Clusters. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC)*, pages 24:1–24:7, 2012.

[5] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2010.

[6] Apache Drill. http://incubator.apache.org/drill/.

[7] Apache Giraph project. http://giraph.apache.org/.

[8] Apache Parquet. http://parquet.apache.org/.

[9] Pramod Bhatotia, Alexander Wieder, İstemi Ekin Akkuş, Rodrigo Rodrigues, and Umut A Acar. Large-scale incremental data processing with change propagation. In *Proceedings of the 3rd USENIX conference on Hot Topics in Cloud Computing (HotCloud)*, pages 18–18, 2011.

[10] Arka A Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. Hierarchical scheduling for diverse datacenter workloads. In *Proceedings of the 4th annual Symposium on Cloud Computing (SOCC)*, page 4, 2013.

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

[12] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the 7th International Conference on World Wide Web (WWW)*, pages 107–117, 1998.

[13] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. HaLoop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4, 2008.

[15] Brian Cho, Muntasir Rahman, Tej Chajed, Indranil Gupta, Cristina Abad, Nathan Roberts, and Philbert Lin. Natjam: design and evaluation of eviction policies for supporting priorities and deadlines in MapReduce clusters. In *Processings of the 4th ACM Symposium on Cloud Computing (SOCC)*, 2013.

[16] Peter Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 24(9):1537–1555, 2012.

[17] Peter Christen and Agus Pudjijono. Accurate Synthetic Generation of Realistic Personal Information. In *Proceedings of the 13th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 507–514, 2009.

[18] Brent Nee Chun and David E Culler. *Market-based proportional resource sharing for clusters*. Computer Science Division, University of California Berkeley, 2000.

[19] Cloudera Impala. http://impala.io.

[20] Edward G Coffman, Jr, Michael R Garey, and David S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[21] Graham Cormode and Minos Garofalakis. Sketching streams through the net: distributed approximate query tracking. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 13–24, 2005.

[22] Graham Cormode and S. Muthukrishnan. An improved data stream summery: the Count-Min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[23] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, pages 1–14, 2014.

[24] Jeffrey Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the 2nd ACM International Conference on Web Search and Data Mining*, pages 1–1, 2009.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[26] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB)*, pages 27–40, 1992.

[27] Prateek Dhawalia, Sriram Kailasam, and Dharanipragada Janakiram. Chisel: A Resource Savvy Approach for Handling Skew in MapReduce Applications. In *2013 IEEE 6th International Conference on Cloud Computing (CloudCom)*, pages 652–660, 2013.

[28] 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 Distributed Computing (HPDC)*, pages 810–818, 2010.

[29] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[30] Ivan P Fellegi and Alan B Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[31] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, 2003.

[32] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 24–24, 2011.

[33] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE)*, pages 522–533, 2012.

[34] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. Handling Data Skew in MapReduce. In *Proceedings of the 2011 International Conference on Cloud Computing and Services Sciences*, pages 574–583, 2011.

[35] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, pages 1199–1208, 2011.

[36] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation (NSDI)*, pages 22–22, 2011.

[37] Shadi 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*, pages 17–24, 2010.

[38] Skewed Join in Pig. http://wiki.apache.org/pig/PigSkewedJoinSpec/.

[39] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 59–72, 2007.

[40] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 261–276, 2009.

[41] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[42] Frank P Kelly, Aman K Maulloo, and David KH Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, pages 237–252, 1998.

[43] Lars Kolb, Andreas Thor, and Erhard Rahm. Load Balancing for MapReduce-based Entity Resolution. In *Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE)*, pages 618–629, 2012.

[44] Lars Kolb, Andreas Thor, and Erhard Rahm. Multi-pass sorted neighborhood blocking with MapReduce. *Computer Science-Research and Development*, 27(1):45–63, 2012.

[45] Hanna Kopcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proceedings of the VLDB Endowment*, 3(1-2):484–493, 2010.

[46] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the 2011 SIGMOD Workshopon Networking Meets Databases*, 2011.

[47] James F. Kurose and Rahul Simha. A microeconomic approach to optimal resource allocation in distributed computer systems. *IEEE Transactions on Computers*, 38(5):705–717, 1989.

[48] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. A Study of Skew in MapReduce Applications. In *The 5th Open Cirrus Summit*, 2011.

[49] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. SkewTune: mitigating skew in MapReduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 25–36, 2012.

[50] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Early accurate results for advanced analytics on MapReduce. *Proceedings of the VLDB Endowment*, 5(10):1028–1039, 2012.

[51] Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon, and P. Bruce Berra. Index structures for structured documents. In *Proceedings of the 1st ACM International Conference on Digital Libraries*, pages 91–99, 1996.

[52] Jimmy Lin. MapReduce is Good Enough? If All You Have is a Hammer, Throw Away Everything That's Not a Nail! *Big Data*, 1(1):28–37, 2013.

[53] LinkedIn Tajo. http://tajo.apache.org/.

[54] 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 International Conference on Management of Data (SIGMOD)*, pages 135–146, 2010.

[55] N McNeill, Hakan Kardes, and Andrew Borthwick. Dynamic record blocking: efficient linking of massive databases in MapReduce. In *Proceedings of the 9th International Workshop on Quality in DataBases*, 2012.

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

[57] Ahmed Metwally, Divyakant Agrawal, and Amr Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th international conference on Database Theory (ICDT)*, pages 398–412, 2005.

[58] HB Newcombe, JM Kennedy, SJ Axford, and AP James. Automatic linkage of vital records. *Science*, 130(3381):954, 1959.

[59] James Norris, Keith Coleman, Armando Fox, and George Candea. OnCall: Defeating spikes with a free-market application cluster. In *Proceedings of the 2004 International Conference on Autonomic Computing (ICAC)*, pages 198–205, 2004.

[60] Alper Okcan and Mirek Riedewald. Processing theta-joins using MapReduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 949–960, 2011.

[61] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1099–1110, 2008.

[62] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[63] Presto. http://prestodb.io.

[64] Apache Hadoop project. http://hadoop.apache.org/.

[65] Apache Hive project. http://hive.apache.org/.

[66] Storm project. http://storm-project.net/.

[67] 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 (SOCC)*, page 16, 2012.

[68] Big Data Vendor Revenue and Market Forecast 2012-2017. http://goo.gl/OsqbwP/.

[69] Florin Rusu and Alin Dobra. Statistical analysis of sketch estimators. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 187–198, 2007.

[70] Thomas Sandholm and Kevin Lai. MapReduce optimization using regulated dynamic prioritization. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 299–310, 2009.

[71] Hadoop YARN Capacity Scheduler. http://goo.gl/297J8z/.

[72] Hadoop YARN Fair Scheduler. http://goo.gl/QiLDm0/.

[73] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuoSys)*, pages 351–364, 2013.

[74] Prorated Supercomputing Fun! Self-Service. http://goo.gl/CiIDLZ/.

[75] Amazon Web Services. http://aws.amazon.com/.

[76] Michael Stonebraker, Paul M Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: a wide-area distributed database system. *The VLDB Journal*, 5(1):48–63, 1996.

[77] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 553–564, 2005.

[78] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 147–156, 2014.

[79] Transaction Processing Performance Council (TPC). TPC Benchmark DS Standard Specification. http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf.

[80] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[81] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benhamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Processings of the 4th ACM Symposium on Cloud Computing (SOCC)*, 2013.

[82] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC)*, pages 235–244, 2011.

[83] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Two sides of a coin: Optimizing the schedule of mapreduce jobs to minimize their makespan and improve cluster performance. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 11–18, 2012.

[84] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Orchestrating an ensemble of MapReduce jobs for minimizing their makespan. *IEEE Transactions on Dependable and Secure Computing*, 10(5):314–327, 2013.

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

[86] Y. Yemini. Selfish optimization in computer networks. In *20th IEEE Conference on Decision and Control including the Symposium on Adaptive Processes*, pages 374–379, Dec 1981.

[87] Matei Zaharia, Dhruba Borthakur, J Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user MapReduce clusters. *EECS Department, University of California, Berkeley, Technical Report USB/EECS-2009-55*, 2009.

[88] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems (EuoSys)*, pages 265–278, 2010.

[89] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, pages 10–10, 2010.

[90] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating Systems design and Implementation (OSDI)*, pages 29–42, 2008.

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

[92] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. PrIter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, page 13, 2011.