Replication Aspects in Distributed Systems

Thesis submitted in partial fulfillment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY

by

Shantanu Sharma

Submitted to the Senate of Ben-Gurion University of the Negev

February 17, 2016

Beer-Sheva

Replication Aspects in Distributed Systems

Thesis submitted in partial fulfillment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY

by

Shantanu Sharma

Submitted to the Senate of Ben-Gurion University of the Negev

Approved by the advisor: [illegible]

Approved by the Dean of the Kreitman School of Advanced Graduate Studies:

February 17, 2016

Beer-Sheva

This work was carried out under the supervision of
Professor Shlomi Dolev
In the Department of Computer Science
Faculty of Natural Science

Research-Student's Affidavit when Submitting the Doctoral Thesis for Judgment

I, Shantanu Sharma, whose signature appears below, hereby declare that:

✓ I have written this Thesis by myself, except for the help and guidance offered by my Thesis Advisors.

✓ The scientific materials included in this Thesis are products of my own research, culled from the period during which I was a research student.

Date: February 17, 2016 Student's name: Shantanu Sharma Signature: Shantanu Sharma

Digitally signed by Shantanu Sharma
DN: cn=Shantanu Sharma, o=UCI, ou=CS, email=shantnu.sharma@uci.edu, c=US
Date: 2016.09.04 00:47:34 -07'00'

Acknowledgements

First, I would like to thank almighty "God" who gave me the inspiration to pursue research as my beautiful career and to have rendezvous with two distinguished professors, Prof. Shlomi Dolev and Prof. Jeffrey D. Ullman, during the whole journey of my graduate study.

Without Prof. Dolev's guidance, this work would be impossible. I am heartily thankful to Prof. Dolev for giving me a golden opportunity to work with him and his large research group. He introduced me to the worlds of "Self-Stabilization" and "MapReduce." He encouraged me to work on MapReduce, which was a black-box for me in the beginning. Undoubtedly, I am the luckiest student to have him as my Doctoral degree's advisor. It is extremely rare to have a mentor like him who is very cool and brilliant. It is not an exaggeration that I learnt a lot from him in my personal and professional life.

I would also like to thank Prof. Jeffrey D. Ullman for assisting me in understanding the models of MapReduce. He worked very closely with me on all research papers related to MapReduce. It was great to have continuous guidance from Prof. Ullman throughout the duration of my PhD. He encouraged me to think differently and allowed frequent discussion on almost everything. He always provided me thoughtful insights and spent a lot of time improving my writing skills. Prof. Ullman is extremely energetic, and I treasure the time when we discussed Meta-MapReduce (Chapter 6) over an entire day, while we both were in the United States (Prof. Ullman had arrived from India in the same morning). Beyond the thesis work, he helped me in non-technical aspects as well; I learnt how to become a better human being and a better student/researcher of computer science. I hope that I will be able to sustain his acquaintances throughout my professional and personal life.

I am thankful to Prof. Foto N. Afrati (National Technical University of Athens), who also discussed key-points in our research papers related to MapReduce. Prof. Afrati gave many remarks to better understand and design efficient algorithms for MapReduce and their proofs. I am also thankful to Prof. Jonathan Ullman (Northeastern University) for his significant contribution in our data cube related paper (Chapter 8). I had the opportunity to work with Prof. Ephraim Korach (BGU), who helped me in developing mathematical proofs. I was delighted to work with Prof. Elad M. Schiller (Chalmers University of Technology), who explained the core of distributed computing systems, in addition to papers on self-stabilizing communication. I am also thankful to Prof. Ehud Gudes (BGU) for assisting me over a long period of time and involving me in EMC's World-Wide-Hadoop project. Prof. Gudes was also active during two survey papers under the same project.

I am also pleased to work with other students of Prof. Dolev: Ariel Hanemann who worked on self-stabilizing communication, Philip Derbeko who helped me in a review paper related to privacy aspects of MapReduce, Yin Li who assisted me in figuring out security issues in MapReduce, and Nisha Panwar who helped me to explore a new area, the fifth generation of mobile communication.

Beyond the scope of this thesis, I am very thankful to Ximing Li and, again, Nisha Panwar. Nisha helped me in many aspects that are completely beyond this thesis. I would like to thank the Head of Department of Computer Science and all the secretaries of the department who helped me at various stages.

This part is incomplete without acknowledging my master's advisor, Prof. Awadhesh Kumar Singh (National Institute of Technology Kurukshetra). Prof. Singh is the person who showed the right path for my career in my master degree. His endless contributions made me aware of the technical world.

I am also thankful to Prof. Sukumar Ghosh (University of Iowa) who inspired me to pursue the higher study abroad. I was also honored by Prof. Maria Potop-Butucaru (University Pierre et Marie Curie), Prof. Ajoy K. Datta (University of Nevada, Las Vegas), Prof. Keren Censor-Hillel (Technion), Prof. Neeraj Mittal (The University of Texas at Dallas), Dr. Marco Pistoia (IBM TJ Watson Research Center), and Prof. Moshe Vardi (Rice University) for providing opportunities to present my research work.

Most importantly, I must thank my family for their support, patience, and understanding my need to focus my time and effort in my thesis. Without my family, I would not be here on this Earth to complete this thesis, let alone any other accomplishments in the past or future.

Table of Contents

Abstract

While hardware and software failures are common in distributed computing systems, *replication* is a way to achieve failure resiliency (availability, consistency, isolation, and reliability) by creating and maintaining several copies of hardware, data, and computing protocols. In this thesis, we propose several approaches that deal with replication of data (or messages) to design a self-stabilizing end-to-end communication algorithm and models for MapReduce computations.

The first part presents a self-stabilizing end-to-end communication algorithm that delivers messages in FIFO order over a non-FIFO, *duplicating*, and bounded capacity channel. The duplication of messages is an essential replication aspect used to overcome loss of messages, but at the same time introduces a challenging end-to-end control that is designed in this work. The new algorithm delivers messages, in the same order as they are sent by the sender, to the designated receiver only once without omission and duplication.

The second part deals with design of models and algorithms for MapReduce. In MapReduce, an input is replicated to several reducers, and hence, the replication of an input dominates the *communication cost*, which is a performance measure of a MapReduce algorithm. A new MapReduce model is introduced here, where, for the first time, the realistic settings in which inputs have non-identical memory sizes is taken into account. The new model opens the door for many useful algorithmic challenges part of which are solved in the sequel. In particular, we present two classes of matching problems, namely the *all-to-all* and the *X-to-Y* matching problems, and show that these matching problems are NP-hard in terms of optimization of the communication cost. We also provide several near-optimal approximation algorithms for both the problems. Next, we provide a new algorithmic approach for MapReduce algorithms that decreases the communication cost significantly, while regarding the *locality* of data and mappers-reducers. The suggested technique avoids the movement of data that does not participate in the final output.

In the second part, we also evaluate impacts on the communication cost for solving the problem of interval join of overlapping intervals and provide 2-way interval join algorithms regarding the reducer memory space. We also investigate the lower bounds on the communication cost for the interval join problem for different types of intervals.

Finally, we consider the problem of computing many marginals of a data cube at one reducer. We find the lower bound on the communication cost for the problem and provide several almost optimal algorithms for computing many marginals.

The third part focuses on the security and privacy aspects in MapReduce. We figure out security and privacy challenges and requirements in the scope of MapReduce, while considering a variety of adversarial capabilities. Then, we provide a privacy-preserving technique for MapReduce computations based on replication of information of the form of Shamir's secret-sharing. We show that though the creation of secret-shares of each value results in increased replication, it makes impossible for an honest-but-curious (or a malicious) cloud to learn the database and computations. We provide MapReduce-based privacy-preserving algorithms for count, search, fetch, equijoin, and range-selection.

Chapter 1

Introduction

A distributed system consists of independent computing nodes that aim to solve problems over distributed information. The distributed system creates and maintains several copies of identical data at different computing nodes, known as *replicas*, either as a consequence of the computing protocol or as a part of the system design. The mechanism of creating and maintaining replicas is known as *replication*. Replication (Chapter 14 of [36] and Chapter 16 of [59]) is a way to achieve availability, consistency, isolation, and reliability of data and computing protocols. In other words, the fault tolerant nature of data and computing protocols can be accomplished using replication of the data and the computing protocols. However, the replication mechanisms require a careful coordination, execution, and agreement among replicated data and computing protocols.

In this thesis, we focus on replication of data in both the aspects, *i.e.*, the replication of data as a consequence of the underlining redundancy choice of the system infrastructure (the replication of packets that leads to a non-FIFO reception), and the replication of data as a part of the system design. Specifically, we focus on replication of packets in end-to-end communication channels to provide a self-stabilizing end-to-end communication algorithm and replication of data in MapReduce to provide the desired output and privacy-preserving computations. In this chapter, we provide an overview of end-to-end communication algorithms (in Section 1.1), an overview of MapReduce framework (in Section 1.2), the problem statements of the thesis (in Section 1.3), and our contribution (in Section 1.4).

1.1 End-to-End Communication Algorithms

End-to-end communication and data-link algorithms are fundamental for any network protocol [103], where a *sender* transmits messages over unreliable communication links to a *receiver* in an exactly one fashion. In addition, errors are introduced during the transmission of packets. Noise in transmission media is also a significant source of errors that result in omission, duplication, and reordering of a message during message transmission. Therefore, most of the end-to-end communication and data-link algorithms assume an initial synchronization between senders and receivers. In addition, error detecting and correcting techniques are employed as an integral part of the transmission in communication networks. Still, when there is a large volume of communication sessions, the probability that an error will not be detected increases, which leads to a possible malfunction of the communication algorithm. In fact, it may lead the algorithm to an arbitrary state from which the algorithm may never recover unless it is *self-stabilizing* [42].

Afek and Brown [4] present a self-stabilizing alternating bit protocol (ABP) for FIFO packet channels without the need for initial synchronization. Self-stabilizing token passing was used as the basis for self-stabilizing ABP over unbounded capacity and FIFO preserving channels in [62, 48]. Dolev and Welch [54] consider the bare network to be dynamic networks with FIFO non-duplicating communication links, and use source routing over the paths to cope with crashes. In [23], an algorithm for self-stabilizing unit capacity data link over a FIFO physical link is assumed. Cournier et al. [37] consider a snap-stabilizing algorithm [29] for message forwarding over message switched network. They ensure one time delivery of the emitted message to the destination within a finite time using a destination based buffer graph and assuming underline FIFO packet delivery.

In the context of dynamic networks and mobile ad hoc networks, Dolev et al. [53, 51, 52] presented self-stabilizing algorithms for token circulation, group multicast, group membership, resource allocation and estimation of network size. Dolev et al. [43] presented a self-stabilizing data link algorithm for reliable FIFO message delivery over bounded non-FIFO and non-duplicating channels. The algorithm [43] delivers a message to a receiver *exactly* once and is also applicable to arbitrary state of dynamic networks of bounded capacity that omit and reorder packets. This algorithm deals *only* with duplication of messages by the sender; however, the algorithm is not able to handle duplication of packets by the (possibly overlay) channel. To the best of our knowledge, there is no algorithm that handles duplication of packets by the channel.

1.2 MapReduce

Data mining operations on large-scale data require data division on several machines, and then, merge the output results. Such a parallel execution provides outputs in a timely manner. However, there are several open challenges to be considered in such a parallel computing model, including distribution of data, failure of machines, ordering of the outputs, scalability of the system, load balancing, and synchronization among machines.

In order to resolve these issues, a programming model, MapReduce [38], was introduced by Google in 2004. MapReduce executes parallel processing using a cluster of computing nodes over large-scale data, without any costly and dedicated computing node like a supercomputer. Since then MapReduce is a standard benchmark for parallel processing over large-scale data, while other companies, including Yahoo, Amazon, and Facebook, are also using MapReduce. Details about MapReduce can be found in Chapter 2 of [76].

Applications and models of MapReduce. Matrix multiplication [14], similarity join [106, 111, 13, 24], detection of near-duplicates [87], interval join [31], spatial join [66, 65, 104], graph processing [11, 85], pattern matching [79], data cube processing [91, 100, 107], skyline queries [12], k-nearest-neighbors finding [115, 80], star-join [117], theta-join [92, 116], and image-audio-video-graph processing [113] are a few applications of MapReduce in the real world. Some models for an efficient MapReduce computation are presented by Karloff et al. [72], Goodrich [61], Lattanzi et al. [73], Pietracaprina et al. [98], Goel and Munagala [60], Ullman [105], Afrati et al. [15, 18], and Fish et al. [57].

1.2.1 Overview of MapReduce

MapReduce, see Figure 1.1, works in two phases: the *Map phase* and the *Reduce phase*, where two user-defined functions, namely, the *map function* and the *reduce function*, are executed over large-scale data. In MapReduce, the data is represented of the form of ⟨*key, value*⟩ pairs.

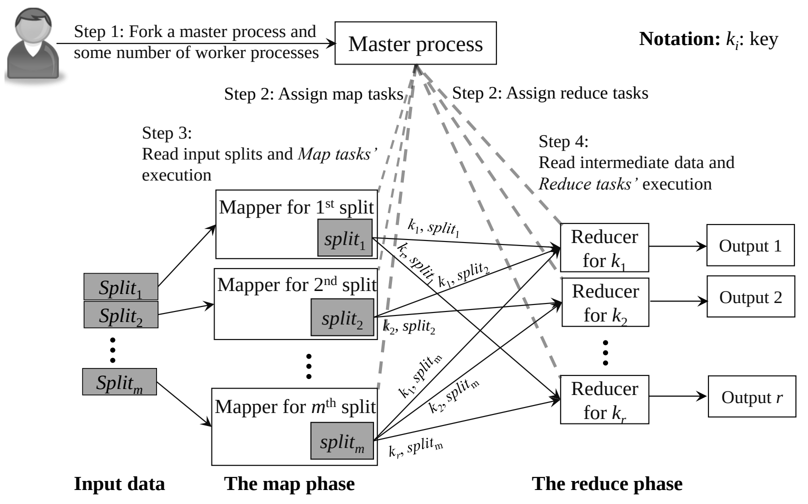


Figure 1.1: An execution of a MapReduce algorithm.

The Map Phase. A MapReduce computation starts from the Map phase where a user-defined map function works on a single input and produces intermediate outputs of the form $\langle key, value\rangle$ pairs. A single input, for example, can be a tuple of a relation. An application of the map function to a single input is called a *mapper*. Several mappers execute in parallel and provide intermediate outputs of the form of $\langle$*key, value*$\rangle$ pairs.

The Reduce Phase. The Reduce phase provides the final output of MapReduce computations. The reduce phase executes a user-defined reduce function on its inputs, *i.e.*, outputs of the Map phase. An application of the reduce function to a single *key* and its associated list of *values* is called a *reducer*. Since there are several *keys* in the indeterminate output, there are also multiple reducers that work in parallel.

Word count example. Word count is a traditional example to illustrate a MapReduce computation, where the task is to count the number of occurrences of each word in a collection of documents. In this example, the original input data is a collection of documents, and a single input is a single document. Each mapper takes a document and implements a user-defined map function that results in a set of $\langle$*key, value*$\rangle$ pairs ($\{\langle w_1, 1\rangle, \langle w_2, 1\rangle, \ldots, \langle w_n, 1\rangle\}$), where each key, w_i, represents a word of the document, and each value is 1. The reduce task is executed subsequently, where a user-defined reduce function aggregates all the occurrences of a particular word w_i and outputs a $\langle w_i, n\rangle$ pair, where w_i is a word that appears in at least one of the given documents, and n is the total number of occurrences of w_i in all the given documents. Recall that there is a reducer for each word, w_i, and this reducer adds all the occurrences of the word, w_i.

Apache Hadoop [1] is a well-known and widely used open-source software implementation of MapReduce for distributed data storage and processing over large-scale data. More details about Hadoop and its Hadoop distributed file system (HDFS) may be found in Chapter 2 of [78]. YARN [2] is the latest version of Hadoop-0.23, details about YARN may be found in [90]. The current MapReduce and Hadoop systems are designed to process data at a single location, *i.e.*, locally distributed processing. Thus, they are not able to process data at geo(graphically)-distributed multiple-clusters. There are some frameworks based on MapReduce for processing geo-distributed datasets without moving them to a single location, and these frameworks are reviewed in our paper [44].

1.3 Overview of the Tasks Investigated

In this thesis, we deal with six problems concerned with data replication for designing communication protocols, computation protocols, and system frameworks, as follows:

Problem 1: Self-stabilizing end-to-end communication over duplicating channels. The first problem deals with data replication for designing a self-stabilizing end-to-end communication protocol. The first problem is how to design an algorithm for a bounded capacity, duplicating, and non-FIFO network, while (*i*) ensuring exactly one copy of packet delivery in the same order as it was sent, (*ii*) handling corruption, omission, and duplication of messages by the channel, (*iii*) ensuring applicability to dynamic networks, and (*iv*) not worrying about starting configurations. The solution to this problem is given in Chapters 2 and 3.

Problem 2: Different-sized inputs in MapReduce. Now, all the following problems deal with replication of data and computations in MapReduce. We define two problems where exactly two inputs are required for computing an output:

All-to-All problem. In the *all-to-all* (*A2A*) problem, a list of inputs is given, and each pair of inputs corresponds to one output.

X-to-Y problem. In the *X-to-Y* (*X2Y*) problem, two disjoint lists X and Y are given, and each pair of elements $\langle x_i, y_j \rangle$, where $x_i \in X, y_j \in Y, \forall i, j$, of the list X and Y corresponds to one output.

Computing common friends on a social networking site and the drug-interaction problem [105] are examples of A2A problems. Skew join is an example of a X2Y problem.

A *mapping schema* defines a MapReduce algorithm. A *mapping schema* assigns inputs to reducers, so that no reducer exceeds its maximum capacity (*i.e.*, a constraint on the lists of inputs that can be assigned a reducer) and all pairs of inputs (in A2A problem) or all pairs of X-to-Y inputs (in X2Y problem) meet in some reducers.

We, for the first time, consider that the inputs may have non-identical sizes. An important and real parameter to measure the performance of a MapReduce algorithm is the *communication cost* — the total amount of data that has to be transferred from the map phase to the reduce phase. The communication cost comes with a tradeoff in the degree of parallelism at the reduce phase. A mapping schema is *optimal* if there is no other mapping schema with a lower communication cost. In this scope, we investigate how to construct optimal mapping schemas or good approximations for the A2A and the X2Y problems. The solution to this problem is given in Chapters 4 and 5.

Problem 3: Meta-MapReduce. In the third problem, we are interested in reducing the amount of data to be transferred to the site of the cloud executing MapReduce computations (and the amount of data transferred from the map phase to the reduce phase). In several problems, the final output depends on *some* inputs, and there, it is not required to send the whole input data to the site of mappers, followed by (intermediate output) data from the map phase to the reduce phase. Hence, the next problem is how to design an algorithmic approach for MapReduce algorithms regarding localities of data and computations, while avoiding the movement of data that do not participate in the final output. The solution to this problem is given in Chapter 6.

Problem 4: Interval join. This problem deals with the designing communication cost efficient algorithms for the problem of interval join of overlapping intervals, where two relations X and Y are given. Each relation contains binary tuples that represent intervals, *i.e.*, each tuple corresponds to an interval and contains the starting-point and ending-point of this interval. The problem lies in assigning each pair of intervals $\langle x_i, y_j \rangle$, where $x_i \in X$ and $y_j \in Y$, $\forall i, j$, such that intervals x_i and y_j share at least one common time, to reducers, while minimizing the communication cost. The solution to this problem is given in Chapter 7.

Problem 5: Computing marginals of a data cube. A data cube [63, 67, 64] is a useful tool for analyzing high dimensional data. A marginal of a data cube is the aggregation of the data in all those tuples that have fixed values in a subset of the dimensions of the cube. In this scope, the problem is how to compute all the marginals of a data cube using a single round of MapReduce, while minimizing the communication cost. The solution to this problem is given in Chapter 8.

Problem 6: Secret shared private MapReduce. The last problem is how to provide information-theoretically secure data and computation outsourcing and query execution using MapReduce. If we can build a technique for information-theoretically secure data and computation outsourcing, and as many MapReduce-based queries' executions, then a user can retrieve only the desired result without involving the source of the data. In addition, during the execution of queries, users and clouds cannot breach the privacy of data and computations, as a consequence. The solution to this problem is given in Chapters 9 and 10.

1.4 Our Contributions and Thesis Outline

Here, we provide an outline of the thesis that is divided into three parts. The thesis is based on our six research papers [47, 9, 6, 10, 16, 50] and three survey papers [44, 39, 95].

The first part focuses on replication aspects in terms of the duplication of messages that is used to overcome loss of messages and design a self-stabilizing end-to-end communication algorithm. The second part focuses on replication aspects in MapReduce in terms of the duplication of data and MapReduce computations; specifically, the focus is on (*i*) building a new model for MapReduce, where, for the first time, the realistic settings in which inputs have non-identical memory sizes is considered, (*ii*) designing an algorithmic technique that regards localities of data and computation sites, and (*iii*) solving the problems of interval join of overlapping intervals and computing the marginals of a data cube. The third part focuses on replication aspects in terms of creating secret-shares of data for ensuring privacy-preserving MapReduce-based computations in the public clouds.

Part I

Chapter 2. In this chapter, we provide assumptions behind the development of the self-stabilizing end-to-end communication algorithm, such as unreliable communication channels, the interleaving model, and asynchronous executions.

Chapter 3. This chapter begins with a description of a simple end-to-end communication algorithm, which is self-stabilizing. However, this algorithm creates a huge overhead on the network. Nevertheless, the simple self-stabilizing end-to-end communication algorithm provides an outline to understand our proposed algorithm that contains the following characteristics, as: (*i*) able to deliver a message to a designated receiver *exactly* one time; (*ii*) applicable to dynamic networks of bounded capacity that *replicate*, omit, and reorder packets; and (*iii*) applicable to an arbitrary state of the network. The content of this chapter appeared in SSS 2012 [47], and the full version of this paper is under review in a journal.

Part II

Chapter 4. In this chapter, we focus on techniques to decrease the communication cost in MapReduce computations and introduce the term *reducer capacity*. The communication cost can be optimized by minimizing the amount of data, which is directly dependent on the number of reducers. However, a reducer cannot hold more inputs whose sum of sizes is greater than the capacity of the reducer, we call it the *reducer capacity*. A single reducer of big enough capacity can hold all the inputs and results in the minimum communication cost and the minimum replication rate (the number of reducers to which an input is sent). However, the use of a single reducer results in no parallelism at the reduce phase and increases in the clock time to finish the MapReduce job.

In this chapter, we consider different-sized inputs and show the relevance of the reducer capacity by considering two classes of problems. We show that these two problems are NP-hard in terms of optimization of the communication cost, and hence, we cannot achieve optimal communication cost. We also study three tradeoffs, as: (*i*) a tradeoff between the reducer capacity and the number of reducers; (*ii*) a tradeoff between the reducer capacity and the parallelism at the reduce phase; and (*iii*) a tradeoff between the reducer capacity and the communication cost.

Chapter 5. We present several near optimal approximation algorithms for both the problems studied in Chapter 4. Specifically, we first present preliminary results and a bin-packing-based approximation algorithm that provides near optimal mapping schemas. The bin-packing-based approximation algorithm applies a bin-packing algorithm on inputs, and then, treats each of the bins as a single input. In addition, we present algorithms, which are based on the bin-packing-based algorithm, to construct optimal mapping schemas in certain cases. For each algorithm, we find upper bounds on the communication cost, the replication rate, and the number of reducers. The content of Chapter 4 and this chapter appeared in DISC 2014 as a brief announcement [5] and in BeyondMR 2015 as an extended abstract [8]. The full version of this paper is accepted in ACM Transactions on Knowledge Discovery from Data (TKDD) [9].

Chapter 6. The federation of the cloud and big data activities is the next challenge, where MapReduce should be modified to avoid (big) data migration across remote (cloud) sites. This is exactly the scope of this chapter, where only essential data for obtaining the result is transmitted, reducing communication, and preserving data privacy as much as possible. In this chapter, we propose an algorithmic technique for MapReduce algorithms, called *Meta-MapReduce*, that decreases the communication cost by allowing us to process and move metadata to clouds and from the map phase to reduce phase. In addition, we explore hashing and filtering techniques for reducing the communication cost and attempt moving only relevant data to the site of mappers-reducers. The ability to use these algorithms depends on the capability of the platform, but many systems today, such as Spark [114], Pregel [84], or recent implementations of MapReduce offer the necessary capabilities. The content of this chapter appeared in SSS 2015 as a brief announcement [6]. An extended abstract of this paper is under review in a conference/workshop.

Chapter 7. In the previous three chapters, we restricted a reducer from holding inputs whose sum of sizes is more than the capacity of the reducer. In this chapter, we consider equal-sized inputs, and hence, all the reducers hold an equal number of inputs. We now define the *reducer size* [15] as the maximum number of inputs that can be assigned to a reducer. We find the lower bound on the replication rate and the communication cost for the problem of interval join of overlapping intervals. We extend the algorithm for interval join proposed in [31] while regarding the reducer size. We consider three types of intervals such as unit-length and equally spaced, variable-length and equally spaced, and equally spaced with specific distribution of the various lengths. The content of this chapter appeared in BeyondMR 2015 [10], and the full version of this paper is under review in a journal.

Chapter 8. In this chapter, we consider the problem of computing the data cube marginals of a fixed order k, we call it k^{th}-other marginals (*i.e.*, all those marginals that fix $n - k$ dimensions of an n-dimensional data cube and aggregate over the remaining k dimensions), using a single round of MapReduce. We show that the replication rate is minimized when the reducers receive all the necessary inputs to compute one marginal of higher order. We define the problem in terms of covering sets of k dimensions with sets of a larger size m, a problem studied under the name "covering numbers [22, 34]." We present several algorithms (for different values of k and m) that meet or come close to yield the minimum possible replication rate for a given reducer size. The content of this chapter [16] is under review in a conference/workshop.

Part III

Chapter 9. While MapReduce is not directly related to the cloud, in the current days, several public clouds, *e.g.*, Amazon Elastic MapReduce, Google App Engine, IBM's Blue Cloud, and Microsoft Azure, enable users to perform MapReduce cloud computations without considering physical infrastructures and software installation. Thus, the deployment of MapReduce in public clouds enables users to process large-scale data in a cost-effective manner and establishes a relationship between two independent entities, *i.e.*, clouds and MapReduce. However, public clouds do not guarantee the rigorous security and privacy of computations as well as of stored data. In this chapter, we highlight security and privacy challenges in MapReduce, privacy requirements for MapReduce, and adversarial models in the context of the privacy in MapReduce. The content of this chapter appeared in Elsevier Computer Science Review [39].

Chapter 10. The main obstacle for providing a privacy-preserving framework for MapReduce in the adversarial (public) clouds is computational and storage efficiency. An adversarial cloud may breach the privacy of data and computations. Hence, in this chapter, we are interested in making a secure and privacy-preserving computation execution and storage-efficient technique for MapReduce computations in the clouds. We look at information-theoretically secure data and computation outsourcing and query execution using MapReduce. Specifically, our focus is on four types of privacy-preserving queries, as follows: `count`, `search` and `fetch`, `equijoin`, and fetch tuples with a value belonging in a `range`. By developing privacy-preserving data and computation outsourcing techniques, a user receives only the desired result without knowing the whole database; moreover, the clouds are also unable to learn the database and the query. The content of this chapter appeared in DBSec 2016 as an extended abstract [50], and the full version of this paper is under review in a journal.

Part I

Replication Aspects in a Communication Algorithm

Chapter 2

Background of a Self-Stabilizing End-to-End Communication Algorithm

Contemporary communication and network technologies enhance the need for automatic recovery. Having a self-stabilizing, predictable, and robust basic end-to-end communication primitive for dynamic networks facilitates the construction of high-level applications. Such applications are becoming extremely essential nowadays where countries' main infrastructures, such as the electrical smart-grid, water supply networks, and intelligent transportation, are based on cyber-systems. We can abstract away the exact network topology, dynamicity and churn and provide (efficient) exactly once message transmission using packets by considering communication networks that has bounded capacity and yet allow omissions, duplications and reordering of packets.

In practice, error detection is a probabilistic mechanism that may not detect a corrupted message, and therefore, the message can be regarded legitimate, driving the system to an arbitrary state after which, availability and functionality may be damaged forever, unless there is human intervention. There is a rich research literature about Automatic Repeat reQuest (ARQ) techniques for obtaining fault-tolerant protocol that provide end-to-end message delivery. However, when initiating a system in an arbitrary state, a non-self-stabilizing algorithm provides no guarantee that the system will reach a legal state after which the participants maintain a coherent state.

Fault-tolerant systems that are *self-stabilizing* [42, 40] can recover after the occurrence of transient faults, which can drive the system to an arbitrary system state. The system designers consider *all* configurations as possible configurations from which the system is started. One significant challenge is to provide an ordering for message transmitted between the Sender and the Receiver. Usually, new messages are identified by a new message number; a number greater than all previously used numbers. Counters of 64-bits, or so, are usually used to implement such numbers. Such designs were justified by claiming that 64-bit values suffice for implementing (practically) unbounded counters. However, a single transient fault may cause the counter to reach the upper limit at once.

In this chapter, we describe our assumptions about the system and network for building a self-stabilizing end-to-end communication protocol.

2.1 Unreliable Communication Channels

We consider a *(communication) graph* of N *nodes* (or processors), p_1, p_2, $\ldots$, p_N. The graph has *(direct communication) links*, (p_i, p_j), whenever p_i can directly send packets to its *neighbor*, p_j (without the use of network layer protocols). The system establishes bidirectional communication between the Sender, p_s, and the Receiver, p_r, which may not be connected directly. Namely, between p_s and p_r there is a unidirectional *(communication) channel* (modeled as a packet set) that transfers packets from p_s to p_r, and another unidirectional channel that transfer packets from p_r to p_s.

When node p_i sends a packet, $pckt$, to node p_j, the operation $send$ adds a copy of $pckt$ to the channel from p_i to p_j, as long as the system follows the assumption about the upper bound on the number of packets in the channel, where $(p_i = p_s) \wedge (p_j = p_r)$ or $(p_i = p_r) \wedge (p_j = p_s)$. We intentionally do not specify (the possibly unreliable) underlying mechanisms that are used to forward a packet from the Sender to the Receiver, *e.g.*, flood routing and shortest path routing, as well as packet forwarding protocols. Once $pckt$ arrives at p_j, p_j triggers the $receive$ event, and deletes $pckt$ from the channel set. We assume that when node p_i sends a packet, $pckt$, infinitely often through the channel from p_i to p_j, node p_j receives $pckt$ infinitely often.

Our proposed self-stabilizing algorithm is oblivious to the channel implementation, which we model as a packet set that has no guarantees for reliability or FIFO order preservation. We assume that, at any given time, the entire number of packets in the system does not exceed a known bound, which we call $capacity$. This bound can be calculated by considering the possible number of network links, number of system nodes, the (minimum and maximum) packet size and the amount of memory that each node allocates for each link. Thus, at any time the sent packets may be omitted, reordered, and duplicated, as long as the system does not violate the channel capacity bound. Note that transient faults can bring the system to consist of arbitrary, and yet capacity bounded, channel sets from which convergence should start and consistency regained.

2.2 The Interleaving Model

Self-stabilizing algorithms do not terminate [42]. The non-termination property can be easily identified in the code of a self-stabilizing algorithm: the code is usually a do forever loop that contains communication operations with the neighbors. An *iteration* is said to be complete if it starts in the loop's first line and ends at the last.

Every node, p_i, executes a program that is a sequence of *(atomic) steps*, where a step starts with local computations and ends with a communication operation, which is either $send$ or $receive$ of a packet. For ease of description, we assume the interleaving model, where steps are executed atomically; a single step at any given time. An input event can either be a packet reception or a periodic timer going off triggering p_i to send. Note that the system is asynchronous. The non-fixed spontaneous node actions and node processing rates are irrelevant to the correctness proof.

The *state*, s_i, of a node p_i consists of the value of all the variables of the node including the set of all incoming communication channels. The execution of an algorithm step can change the node's state, and the communication channels that are associated with it. The term *(system) configuration* is used for a tuple of the form $(s_1, s_2, \cdots, s_N)$, where each s_i is the state of node p_i (including packets in transit for p_i). We define an *execution (or run)* $R = c_0, a_0, c_1, a_1, \ldots$ as an alternating sequence of system configurations, c_x, and steps a_x, such that each configuration c_{x+1} (except the initial configuration c_0) is obtained from the preceding configuration, c_x, by the execution of the steps a_x. We often associate the step index notation with its executing node p_i using a second subscript, *i.e.*, a_{i_x}. We represent the omissions, duplications, and reordering using environment steps that are interleaved with the steps of the processors in R.

2.2.1 Asynchronous executions that allow progress

We say that an asynchronous execution, R, allows progression when every algorithm step that is applicable infinitely often in R is executed infinitely often in R. Moreover, we require that R allows progression with respect to communication. Namely, p_i's infinitely often $send$ operations of a packet, $pckt$, to p_j, imply infinitely often $receive$ operations of $pckt$ by p_j. Thus, the communication graph may often change and the communication delays may change, as long as they respect the upper bound, N, on the number of nodes, the network capacity and the above requirements. We allow any churn rate, assuming that joining processors reset their own memory, and by that prevent the introduction of information about packets other than the ones that exist in $\{p_1, p_2, \ldots, p_N\}$, *i.e.*, respecting the assumed bounded packet capacity of the entire network.

When considering system convergence to legal behavior, we measure the number of *asynchronous rounds*. We define the first asynchronous round in an execution R as the shortest prefix, R', of R in which node p_i sends at least one packet to p_j via their communication channel, and p_j receives from this channel at least one packet that was sent from p_i, where $(p_i = p_s \wedge p_j = p_r)$ or $(p_i = p_r \wedge p_j = p_s)$. The second asynchronous round, R'', is the first asynchronous round in R's suffix that follows the first asynchronous round, R', and so on. Namely, $R = R' \circ R'' \circ R''' \ldots$, where $\circ$ is the concatenation operator.

2.3 The Task

We define the system's task by a set of executions called *legal executions* (LE) in which the task's requirements hold. A configuration c is a *safe configuration* for an algorithm and the task of LE provided that any execution that starts in c is a legal execution, which belongs to LE. An algorithm is *self-stabilizing* with relation to the task LE when every (unbounded) execution of the algorithm reaches a safe configuration with relation to the algorithm and the task.

The proposed *self-stabilizing end-to-end communication* (S^2E^2C) algorithm (Chapter 3) provides FIFO and exactly once-delivery guarantees for bounded networks that omit, duplicate, and reorder packets within the channel. Moreover, the algorithm considers arbitrary starting configurations and ensures error-free message delivery. In detail, given a system execution, R, and a pair, p_s and p_r, of sending and receiving nodes, we associate the message sequence $s_R = im_0, im_1, im_2, \ldots$, which are fetched by p_s, with the message sequence $r_R = om_0, om_1, om_2, \ldots$, which are delivered by p_r. Note that we list messages according to the order they are fetched (from the higher level application) by the Sender, thus two or more (consecutive or non-consecutive) messages can be identical. The S^2E^2C task requires that for every legal execution, $R \in LE$, there is an infinite suffix, R', in which infinitely many messages are delivered, and $s_{R'} = r_{R'}$. It should be noted that packets are not actually received by the Receiver in their correct order, but eventually it holds that the Receiver delivers the messages to its application layer by the order in which they were fetched by the Sender from its application layer.

When demonstrating safety properties, such as the order of message delivery, we consider asynchronous executions that allow progression, which can include omission steps. Note an adversarial execution can include (selective) packet omission in a way that will prevent packet exchange between the Sender and the Receiver. Thus, when demonstrating liveness properties, such as how long does it take the system to reach a legal execution, we consider nice executions. I.e., we say that an asynchronous execution R that allows progression is *nice* when it does not include any omission step.

Chapter 3

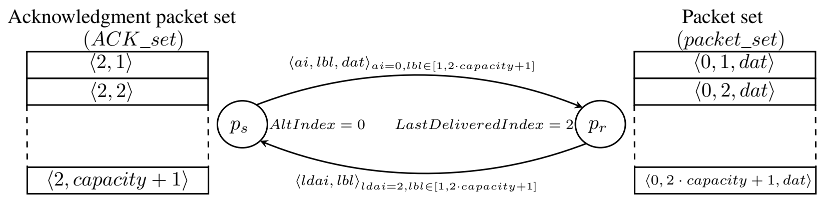
Self-Stabilizing End-to-End Algorithm

In this chapter, we first provide a simple (first attempt solution) end-to-end communication algorithm that is self-stabilizing and copes network faults, such as packet omissions, duplications, and reordering. This first attempt algorithm has has a large overhead, but it prepares the presentation of our proposal for an efficient solution (Section 3.2) that is based on error correcting codes.

3.1 A First Attempt Solution

We regard two nodes, p_s and p_r, as sender, and respectively, receiver; see our first attempt sketch of an end-to-end communication protocol in Figure 3.1. The goal is for p_s to fetch messages, m, from its application layer, send m over the communication channel, and for p_r to deliver m to its application layer exactly once and in the same order by which the Sender fetched them from its application layer. The Sender, p_s, fetches the message m and starts the transmission of $(2 \cdot \mathit{capacity} + 1)$ copies of m to p_r, and p_r acknowledges m upon arrival. These transmissions use distinct labels for each copy, *i.e.*, $(2 \cdot \mathit{capacity} + 1)$ labels for each of m's copies. The Sender, p_s, does not stop retransmitting m's packets until it receives $(\mathit{capacity} + 1)$ distinctly labeled acknowledgment packets from p_r; see details in Figure 3.1.

Let us consider the set of packets $X = \{\langle ai, \ell, dat\rangle\}_{\ell \in [1, 2 \cdot \mathit{capacity}+1]}$ that p_r receives during a legal execution, where $ai = 0$, as in the example that appears in Figure 3.1. We note that X includes a majority of packets in X that have the same value of dat, because the channel can add at most $\mathit{capacity}$ packets (due to channel faults, such as message duplication, or transient faults that occurred before the starting configuration), and thus, p_s has sent at least $(\mathit{capacity} + 1)$ of these packets, *i.e.*, the majority of the arriving packets to p_r have originated from p_s, rather than the communication channel between p_s and p_r (due


Acknowledgment packet set
(ACK_set)
$\langle 2,1\rangle$
$\langle 2,2\rangle$
$\langle 2, capacity+1\rangle$
$\langle ai, lbl, dat\rangle_{ai=0, lbl\in[1,2\cdot capacity+1]}$
p_s
$AltIndex = 0$
$LastDeliveredIndex = 2$
p_r
$\langle ldai, lbl\rangle_{ldai=2, lbl\in[1,2\cdot capacity+1]}$
Packet set
$(packet\_set)$
$\langle 0,1,dat\rangle$
$\langle 0,2,dat\rangle$
$\langle 0, 2\cdot capacity+1, dat\rangle$

The communication channels do not indicate to their receiving ends whether the transmitted packets were subject to duplication, omission or reordering. The algorithm facilitates the correct delivery of m by letting p_s send $(2 \cdot capacity + 1)$ copies of the message $m = \langle dat\rangle$ to p_r, and requiring p_r to receive $(2 \cdot capacity + 1)$ packets, where the majority of them are copies of m. Namely, p_s maintains an *alternating index*, $AltIndex \in [0, 2]$, which is a counter that is incremented modulo 3 every time m is fetched and by that allow recovery from an arbitrary starting configuration. Moreover, p_s transmits to p_r a set of packets, $\langle ai, lbl, dat\rangle$, where $ai = AltIndex$, and lbl are packet labels that distinguish this packet among all of m's copies. The example illustrated above shows that when transmitting the packet set $\{\langle 0, 1, dat\rangle$, $\langle 0, 2, dat\rangle$, ..., $\langle 0, 2 \cdot capacity + 1, dat\rangle\}$, the alternating index, 0, distinguishes between this transmission set, and its predecessor set, which has the alternating index 2, as well as the successor sets, which has the alternating index 1. This transmission ends once p_r receives a packet set, $\{\langle 0, \ell, dat\rangle\}_{\ell\in[1,2\cdot capacity+1]}$, that is distinctly labeled by ℓ with respect to the alternating index 0. (Note that when receiving a packet with a label that exists in the received packet set, the Receiver replaces the existing packet with the arriving one.) During legal executions, the set of received packets includes a majority of packets that have the same value of dat. When such a majority indeed exists, p_r delivers $m = \langle dat\rangle$. After this decision, p_r updates $LastDeliveredIndex \leftarrow 0$ as the value of the last delivered alternating index.

The correct packet transmission depends on the synchrony of m's alternating index at the sending-side, and $LastDeliveredIndex$ on the Receiver side, as well as the packets that p_r accumulates in $packet\_set_r$. The Sender repeatedly transmits this packet set until it receives $(capacity + 1)$ distinctly labeled acknowledgment packets, $\langle ldai, lbl\rangle$, from the Receiver for which it holds that $ldai = AltIndex$. The Receiver acknowledges the Sender for each incoming packet, $\langle ai, lbl, dat\rangle$, using acknowledgment packet $\langle ldai, lbl\rangle$, where $ldai$ refers to the value, $LastDeliveredIndex$, of the last alternating index for which there was a receiving-side message delivery to the application layer. Thus, with respect to the above example, p_s does not fetch another application layer message before it receives at least $(capacity + 1)$ acknowledgment packets; each corresponding to one of the $(2 \cdot capacity + 1)$ packets that p_r received from p_s. On the receiving-side, p_r delivers the message, $m = \langle dat\rangle$, from one of the $(capacity + 1)$ (out of $(2 \cdot capacity + 1)$) distinctly labeled packets that have identical dat and ai values. After this delivery, p_r assigns ai to $LastDeliveredIndex$, resets its packet set and restarts accumulating packets, $\langle ai', lbl', dat'\rangle$, for which $LastDeliveredIndex \neq ai'$.

Figure 3.1: An end-to-end communication protocol (first attempt)

to channel faults or transient faults that occurred before the starting configuration). The protocol tolerates channel reordering faults, because the Sender fetches one message at a time, and since it does not fetch another before it receives an acknowledgment about the delivery of the current one. The protocol marks each packet with a distinct label in order to allow a packet selection that is based on majority in the presence of duplication faults.

The above first-attempt solution delivers each message exactly once in its (sending-order) while producing a large communication overhead. The proposed solution (Section 3.2) uses error correction codes and has a smaller overhead. It fetches a number of messages, m, on the sending-side. Then, it concurrently transmits them to the other end after transforming them into packets that are protected by error correction codes, and then, delivering them at their sending order without omission, or duplication. We explain how to circumvent the difficulty that the communication channel can introduce up to capacity erroneous packets by considering the problem of having up to capacity erroneous bits in any packet.

3.1.1 Error correction codes for payload sequences

Error correction codes [89] can mitigate bit-flip errors in (binary) words, where such words can represent payload in single data packet, or as we show here, can be used to help recover wrong words in a sequence of them. These methods use redundant information when encoding data, so that after the error occurrence, the decoding procedure will be able to recover the originally encoded data without errors. Namely, an error correction code $ec()$ encodes a payload w (binary word) of length wl with payload $c = ec(w)$ of length cl, where $cl > wl$. The payload w can be later restored from c' as long as the Hamming distance between c' and c is less than a known *error threshold*, t_{ecc}, where the Hamming distance between c' and c is the smallest number of bits that one has to flip in c, in order to get c'.

Existing methods for error correction codes can also be used for a sequence of packets and their payloads, see Figure 3.2. These sequences are encoded on the sender-side, and sent over a bounded capacity, omitting, duplicating and non-FIFO channel, before decoding them on the receiver-side. On that side, the originally encoded payload sequence is decoded, as long as the error threshold is not smaller than the channel capacity, *i.e.*, $t_{ecc} \geq capacity$.

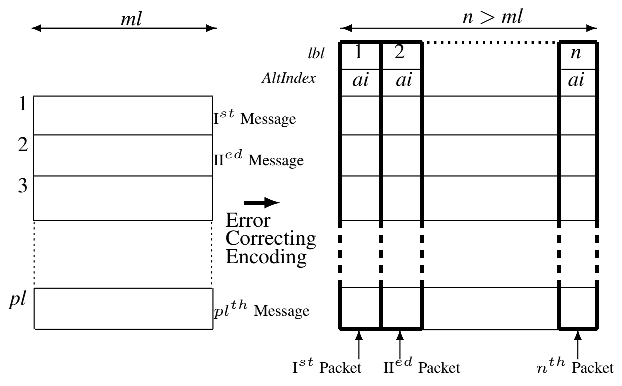
This method removes the issue of having up to $capacity$ erroneous packets by considering the problem of having up to $capacity$ erroneous bits in any packet. This problem is solved by using error correction codes to mask the erroneous bits. The proposed solution allows correct message delivery even though up to $capacity$ of packets are erroneous, *i.e.*, packets that appeared in channel (due to transient faults that occurred before the starting configuration) rather than added to the channel by Sender, or due to channel faults during the system execution.

3.2 Self-Stabilizing End-to-End Algorithm (S^2E^2C)

We propose an efficient S^2E^2C algorithm that fetches a number of messages, m, and encodes them according to the method presented in Figure 3.2. The Sender then concurrently transmits m's encoded packets to the receiving end until it can decode and acknowledge m. Recall that the proposed method for error correction can tolerate communication channels that, while in transit, omit, duplicate and reorder m's packets, as well as add up to $capacity$ packets to the channel (due to transient faults that have occurred before the starting configuration rather than packets that the Sender adds to the channel during the system execution). We show how the Sender and the Receiver can use the proposed error correction method for transmitting and acknowledging m's packets. The pseudocodes of the Sender Algorithm 5 and the Receiver Algorithm 6 are presented in Appendix A.

Reliable, ordered, and error-checked protocols in the transport layer, such as TCP/IP, often consider the delivery of a stream of octets between two network ends. The algorithm presented in Figure 3.2 considers a transport layer protocol that repeatedly fetches another part of the stream. Upon each such fetch, the protocol breaks that part of the stream into m sub-parts, and the protocol refers to m sub-parts as the application layer messages. Note that the size of each such message can be determined by the (maximal) payload size of packets that the Sender transmits to the Receiver, because the payload of each transmitted packet needs to accommodate one of the n-bit words that are the result of transposing m stream sub-parts.

The S^2E^2C algorithm extends the first attempt end-to-end communication protocol (Figure 3.1), *i.e.*, the Sender, p_s, transmits the packets $\langle ai, lbl, dat\rangle$, and the Receiver, p_r, acknowledges using the $\langle ai, lbl\rangle$ packets, where $ai \in [0,2]$ is the state alternating index, and lbl are packet labels that are distinct among all of the packets that are associated with messages $m = \langle dat\rangle$. Moreover, it uses the notation of the proposed error correction method (Figure 3.2), *i.e.*, the Sender fetches batches of pl application layer messages of length ml bits that are coded by n bit payloads that tolerate up to $capacity$ erroneous bits. The Sender, p_s, fetches pl (application layer) messages, $m = \langle dat\rangle$, encodes them into n (distinctly labeled) packets, $\langle ai = AltIndex_s, lbl, dat\rangle$, according to the proposed error correction method (Figure 3.2), and repeatedly transmits these n (distinctly labeled) packets to p_r until p_s receives from p_r (at least) $(capacity + 1)$



The (sending-side) encoder considers a batch of (same length) messages as a (bit) matrix, where each message (bit representation) is a matrix row. It transposes these matrices by sending the matrix columns as encoded data packets. Namely, the Sender fetches, $[m_j]_{j\in[1,pl]}$, a batch of pl messages from the application layer each of length ml bits, and calls the function $Encode([m_j]_{j\in[1,pl]})$. This function is based on an error correction code, ecc, that for ml bits word, m_j, codes an n bits words, cm_j, such that cm_j can bear up to $capacity$ erroneous bits, *i.e.*, ecc's error threshold, t_{ecc}, is $capacity$. The function then takes these pl (length n bits) words, cm_j, and returns n (length pl bits) packet payloads, $[pyld_k]_{k\in[1,pl]}$, that are the columns of a bit matrix in which the jth row is cm_j's bit representation, see the image above for illustration.
The (receiver-side) uses the function $Decode([pyld'_k]_{k\in[1,pl]})$, which requires n packet payloads, and assumes that at most $capacity$ of them are erroneous packets that appeared in channel (due to transient faults that occurred before the starting configuration) rather than added to the channel by the Sender, or due to channel faults during the system execution, *i.e.*, from $[pyld_k]_{k\in[1,pl]}$.
This function first transposes the arrived packet payloads, $[pyld'_k]_{k\in[1,pl]}$ (in a similar manner to $Encode()$), before using ecc for decoding $[m_j]_{j\in[1,pl]}$ and delivering the original messages to the Receiver's application layer.
Namely when the Receiver accumulates n distinct label packets, $capacity$ of the packets may be wrong or unrelated. However, since the i^{th} packet, out of the n distinctly labeled packets, encodes the i^{th} bits of all the pl encoded messages, if the i^{th} packet is wrong, the decoder can still decode the data of the original pl messages each of length $ml < n$. The i^{th} bit in each encoded message may be wrong, in fact, capacity of packets maybe wrong yielding capacity of bits that may be wrong in each encoded message. However, due to the error correction, all the original pl messages of length ml can be recovered, so the Receiver can deliver the correct pl messages in the correct order. Note that in this case, although the channel may reorder the packets, the labels maintain the sending-order, because the i^{th} packet is labeled with i. In this proposed solution, the labels also facilitate duplication fault-tolerance, because the Receiver always holds at most one packet with label i, *i.e.*, the latest.

Figure 3.2: Packet formation from messages

(distinctly labeled) acknowledgment packets $\langle \mathit{ldai}', \mathit{lbl}' \rangle$, for which after convergence $\mathit{ldai}' = \mathit{AltIndex}$. The Receiver repeatedly transmits the acknowledgment packets $\langle \mathit{ldai}', \mathit{lbl}' \rangle$, which acknowledge the messages in the previous batch that it had delivered to its application layer that had the alternating index, $ai = \mathit{LastDeliveredIndex}$. Note that the Receiver repeatedly sends $(\mathit{capacity} + 1)$ acknowledgment packets, as a response to the n received packets, rather than a particular packet that has arrived. Namely, p_r accumulates arriving packets, $\langle ai, \mathit{lbl}, \mathit{dat} \rangle$, whose alternating indexes, ai, is different from the last delivered one, $\mathit{LastDeliveredIndex}$. Moreover, once p_r has n (distinctly labeled) packets, which are $\{\langle ai, \ell, \mathit{dat} \rangle\}_{\ell \in [1,n]} : ai \neq \mathit{LastDeliveredIndex}$, the Receiver p_r updates $\mathit{LastDeliveredIndex}$ according to ai, as well as use the proposed error correction method (Figure 3.2) for decoding m before delivering it.

Note that p_s transmits to p_r a set of n (distinctly labeled with respect to a single alternating index) of m's packets, ***i.e.***, m's packets, which the channel can omit, duplicate and reorder. Thus, once p_r receives n packets, p_r can use the proposed error correction method (Figure 3.2) as long as their alternating index is different from the last delivered one, $\mathit{LastDeliveredIndex}$, because at least $(n - \mathit{capacity})$ of these packets were sent by p_s. Similarly, p_r transmits to p_s a set of $(\mathit{capacity} + 1)$ (distinctly labeled with respect to a single alternating index) of m's acknowledgment packets. Thus, once $(\mathit{capacity} + 1)$ of m's acknowledgment packets (with ldai matching to $\mathit{AltIndex}$) arrive at the sending-side, p_s can conclude that at least one of them was transmitted by p_r, as long as their alternating index, ldai, is the same as the one used for m, $\mathit{AltIndex}$.

The correctness arguments show that eventually we will reach an execution in which the Sender fetches a new message batch infinitely often, and the Receiver will deliver the messages fetched by the Sender before its fetches the next message batch. Thus, every batch of pl fetched messages is delivered exactly once, because after delivery the Receiver resets its packet set and changes its $\mathit{LastDeliveredIndex}$ to be equal to the alternating index of the Sender. The Receiver stops accumulating packets from the Sender (that their alternating index is $\mathit{LastDeliveredIndex}$) until the Sender fetches the next message batch, and starts sending packets with a new alternating index. Note that the Sender only fetches new messages after it gets $(\mathit{capacity} + 1)$ distinctly labeled acknowledgments, $\langle \mathit{ldai}, \mathit{lbl} \rangle$ (that their alternating index, ldai, equals to p_s's $\mathit{AltIndex}$). When the Receiver holds n (distinctly labeled) packets out of which at most $\mathit{capacity}$ are erroneous ones, it can convert the packets back to the original messages, see (Figure 3.2).

The correctness of Algorithms 5 and 6 is given in Appendix A.2.

Part II

Replication Aspects in MapReduce

Chapter 4

Intractability of Mapping Schemas

In the second and the third parts of the thesis, we will investigate impacts of replications in designing models and algorithms for MapReduce.

A MapReduce algorithm can be described by a *mapping schema*, which assigns inputs to a set of reducers, such that for each required output there exists a reducer that receives all the inputs that participate in the computation of this output. Reducers have a capacity, which limits the sets of inputs that they can be assigned. However, individual inputs may vary in terms of size. We consider mapping schemas where input sizes are part of the considerations and restrictions. One of the significant parameters to optimize in any MapReduce job is the communication cost — the total amount of data that is required to move from the map phase to the reduce phase. The communication cost can be optimized by minimizing the number of copies of inputs sent to the reducers.

In this chapter, we consider a family of problems where it is required that each input meets with each other input in at least one reducer. In this chapter, we define and prove that the *all-to-all mapping schema problem* is NP-hard for $z > 2$ identical capacity reducers (in Sections 4.2.1 and 4.3.1). Also, we define the *X-meets-Y mapping schema problem* and prove that the same problem is NP-hard for $z > 1$ identical capacity reducers (in Sections 4.2.2 and 4.3.2).

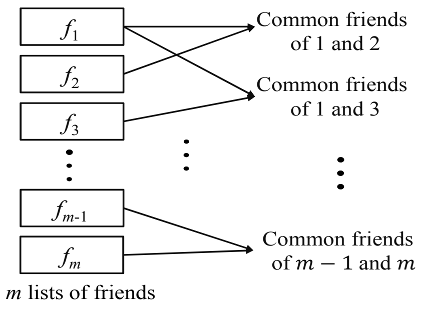
4.1 Preliminarily and Motivating Examples

Communication cost. An important performance measure for MapReduce algorithms is the amount of data transferred from the *mappers* (the processes that implement the map function) to the *reducers* (the processes that implement the reduce function). This is called the *communication cost*. The minimum communication cost is, of course, the size of the desired inputs that provide the final output, since we need to transfer all these inputs from the mappers to the reducers at least once. However, we may need to transfer the same input to several reducers, thus increasing the communication cost. Depending on various factors of our setting, each reducer may process a larger or smaller amount of data. The amount of data each reducer processes however affects the wall clock time of our algorithms and the degree of parallelization. If we send all data in one reducer, then we have low communication (equal to the size of the data) but we have low degree of parallelization, and thus, the wall clock time increases.

Reducer capacity. The maximum amount of data a reducer can hold is a constraint when we build our algorithm. We define *reducer capacity* to be the upper bound on the sum of the sizes of the $values$ that are assigned to the reducer. For example, we may choose the reducer capacity to be the size of the main memory of the processor on which the reducer runs or we may arbitrarily set a low reducer capacity if we want high parallelization. We always assume in this chapter that all the reducers have an identical capacity, denoted by q.

Examples.

Example 4.1 *Computing common friends. An input is a list of friends. We have such lists for m persons. Each pair of lists of friends corresponds to one output, which will show us the common friends of the respective persons. Thus, it is mandatory that lists of friends of every two persons are compared. Specifically, the problem is: a list $F = \{f_1, f_2, \ldots, f_m\}$ of m friends is given, and each pair of elements $\langle f_i, f_j\rangle$ corresponds to one output, common friends of persons i and j; see Figure 4.1.*



f_1
f_2
f_3
f_{m-1}
f_m
m lists of friends
Common friends of 1 and 2
Common friends of 1 and 3
Common friends of $m-1$ and m

Figure 4.1: Computing common friends example.

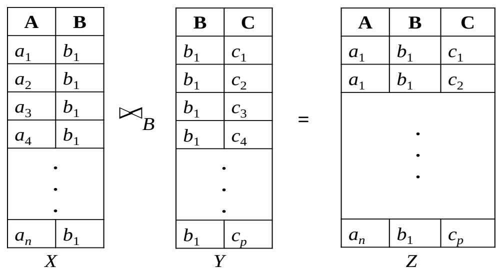


Figure 4.2: Skew join example for a heavy hitter, b_1.

Example 4.2 *Skew join of two relations $X(A, B)$ and $Y(B, C)$. The join of relations $X(A, B)$ and $Y(B, C)$, where the joining attribute is B, provides output tuples $\langle a, b, c\rangle$, where (a, b) is in A and (b, c) is in C. One or both of the relations X and Y may have a large number of tuples with an identical B-value. A value of the joining attribute B that occurs many times is known as a* heavy hitter. *In skew join of $X(A, B)$ and $Y(B, C)$, all* *the tuples of both the relations with an identical heavy hitter should appear together to provide the output tuples.*

In Figure 4.2, b_1 is considered as a heavy hitter; hence, it is required that all the tuples of $X(A, B)$ and $Y(B, C)$ with the heavy hitter, $B = b_1$, should appear together to provide the desired output tuples, $\langle a, b_1, c\rangle$ ($a \in A, b_1 \in B, c \in C$), which depend on exactly two inputs.

4.2 Mapping Schema and Tradeoffs

Our system setting is an extension of the standard system setting [15] for MapReduce algorithms, where we consider, for the first time, inputs of different sizes. In this section, we provide formal definitions and some examples to show the tradeoff between communication cost and degree of parallelization.

Mapping Schema. A mapping schema is an assignment of the set of inputs to some given reducers so that the following two constraints are satisfied:

- A reducer is assigned inputs whose sum of the sizes is less than or equal to the reducer capacity q.
- For each output, we must assign its corresponding inputs to at least one reducer in common.

Optimal Mapping Schema. A mapping schema is optimal when the communication cost is minimum. The number of reducers we use often is minimal for an optimal mapping schema but this may not always be the case. We offer insight about communication cost and number of reducers used in Examples 4.3 and 4.4.

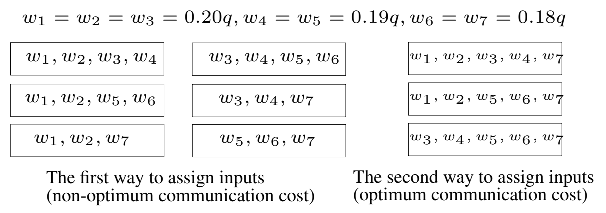
Tradeoffs. The following tradeoffs appear in MapReduce algorithms and in particular in our setting:

- A tradeoff between the reducer capacity and the number of reducers. For example, large reducer capacity allows the use of a smaller number of reducers.
- A tradeoff between the reducer capacity and the parallelism. For example, if we want to achieve a high degree of parallelism, we set low reducer capacity.
- A tradeoff between the reducer capacity and the communication cost. For example, in the case reducer capacity is equal to the total size of the data then we can use one reducer and have minimum communication (of course, this goes at the expense of parallelization).

In the subsequent subsections, we present two mapping schema problems, namely the *A2A mapping schema problem* and the *X2Y mapping schema problem*. In addition, the readers will also see the impact of the above mentioned tradeoff in the mapping schema problems and ways for obtaining an optimal mapping schema.

4.2.1 The *All-to-All Mapping Schema Problem*

An instance of the *A2A mapping schema problem* consists of a list of m inputs whose input size list is $W = \{w_1, w_2, \ldots, w_m\}$ and a set of z identical reducers of capacity q. A solution to the *A2A mapping schema problem* assigns every pair of inputs to at least one reducer in common, without exceeding q at any reducer.

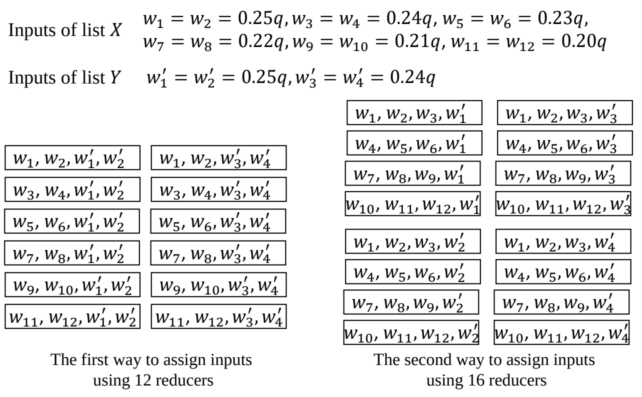


$w_1 = w_2 = w_3 = 0.20q, w_4 = w_5 = 0.19q, w_6 = w_7 = 0.18q$
w_1, w_2, w_3, w_4
w_3, w_4, w_5, w_6
w_1, w_2, w_3, w_4, w_7
w_1, w_2, w_5, w_6
w_3, w_4, w_7
w_1, w_2, w_5, w_6, w_7
w_1, w_2, w_7
w_5, w_6, w_7
w_3, w_4, w_5, w_6, w_7
The first way to assign inputs (non-optimum communication cost)
The second way to assign inputs (optimum communication cost)

Figure 4.3: An example to the *A2A mapping schema problem.*

Example 4.3 *We are given a list of seven inputs* $I = \{i_1, i_2, \ldots, i_7\}$ *whose size list is* $W = \{0.20q, 0.20q, 0.20q, 0.19q, 0.19q, 0.18q, 0.18q\}$ *and reducers of capacity* q*. In Figure 4.3, we show two different ways that we can assign the inputs to reducers. The best we can do to minimize the communication cost is to use three reducers. However, there is less parallelism at the reduce phase as compared to when we use six reducers. Observe that when we use six reducers, then all reducers have a lighter load, since each reducer may have capacity less than* $0.8q$*.*

The communication cost for the second case (3 reducers) is approximately $3q$*, whereas for the first case (6 reducers) it is approximately* $4.2q$*. Thus, in tradeoff, in the 3-reducers case we have low communication cost but also lower degree of parallelization, whereas in the 6-reducers case we have high parallelization at the expense of the communication cost.*



Inputs of list X
$w_1 = w_2 = 0.25q, w_3 = w_4 = 0.24q, w_5 = w_6 = 0.23q,$
$w_7 = w_8 = 0.22q, w_9 = w_{10} = 0.21q, w_{11} = w_{12} = 0.20q$
Inputs of list Y
$w'_1 = w'_2 = 0.25q, w'_3 = w'_4 = 0.24q$
w_1, w_2, w'_1, w'_2
w_1, w_2, w'_3, w'_4
w_3, w_4, w'_1, w'_2
w_3, w_4, w'_3, w'_4
w_5, w_6, w'_1, w'_2
w_5, w_6, w'_3, w'_4
w_7, w_8, w'_1, w'_2
w_7, w_8, w'_3, w'_4
w_9, w_{10}, w'_1, w'_2
w_9, w_{10}, w'_3, w'_4
$w_{11}, w_{12}, w'_1, w'_2$
$w_{11}, w_{12}, w'_3, w'_4$
w_1, w_2, w_3, w'_1
w_1, w_2, w_3, w'_3
w_4, w_5, w_6, w'_1
w_4, w_5, w_6, w'_3
w_7, w_8, w_9, w'_1
w_7, w_8, w_9, w'_3
$w_{10}, w_{11}, w_{12}, w'_1$
$w_{10}, w_{11}, w_{12}, w'_3$
w_1, w_2, w_3, w'_2
w_1, w_2, w_3, w'_4
w_4, w_5, w_6, w'_2
w_4, w_5, w_6, w'_4
w_7, w_8, w_9, w'_2
w_7, w_8, w_9, w'_4
$w_{10}, w_{11}, w_{12}, w'_2$
$w_{10}, w_{11}, w_{12}, w'_4$
The first way to assign inputs using 12 reducers
The second way to assign inputs using 16 reducers

Figure 4.4: An example to the *X2Y mapping schema problem.*

4.2.2 The *X2Y Mapping Schema Problem*

An instance of the *X2Y mapping schema problem* consists of two disjoint lists X and Y and a set of identical reducers of capacity q. The inputs of the list X are of sizes $w_1, w_2, \ldots, w_m$, and the inputs of the list Y are of sizes $w'_1, w'_2, \ldots, w'_n$. A solution to the *X2Y mapping schema problem* assigns every two inputs, the first from one list, X, and the second from the other list, Y, to at least one reducer in common, without exceeding q at any reducer.

Example 4.4 *We are given two lists, X of 12 inputs, and Y of 4 inputs (see Figure 4.4) and reducers of capacity q. We show that we can assign each input of the list X with each input of the list Y in two ways. In order to minimize the communication cost, the best way is to use 12 reducers. Note that we cannot obtain a solution for the given inputs using less than 12 reducers. However, the use of 12 reducers results in less parallelism at the reduce phase as compared to when we use 16 reducers.*

4.3 Intractability of Finding a Mapping Schema

In this section, we will show that the *A2A* and the *X2Y* mapping schema problems do not possess a polynomial solution. In other words, we will show that the assignment of *two required inputs* to the minimum number of identical-capacity reducers to find solutions to the *A2A* and the *X2Y* mapping schema problems cannot be achieved in polynomial time.

4.3.1 NP-hardness of the *A2A Mapping Schema Problem*

A list of inputs $I = \{i_1, i_2, \ldots, i_m\}$ whose input size list is $W = \{w_1, w_2, \ldots, w_m\}$ and a set of identical reducers $R = \{r_1, r_2, \ldots, r_z\}$, are an input instance to the *A2A mapping schema problem*. The *A2A mapping schema problem* is a decision problem that asks whether or not there exists a mapping schema for the given input instance such that every input, i_x, is assigned with every other input, i_y, to at least one reducer in common. An answer to the *A2A mapping schema problem* will be "yes," if for each pair of inputs ($\langle i_x, i_y \rangle$), there is at least one reducer that holds them.

In this section, we prove that the *A2A mapping schema problem* is NP-hard in the case of $z > 2$ identical reducers. In addition, we prove that the *A2A mapping schema problem* has a polynomial solution to one and two reducers.

If there is only one reducer, then the answer is "yes" if and only if the sum of the input sizes $\sum_{i=1}^{m} w_i$ is at most q. On the other hand, if $q < \sum_{i=1}^{m} w_i$, then the answer is "no." In case of two reducers, if a single reducer is not able to accommodate all the given inputs, then there must be at least one input that is assigned to only one of the reducers, and hence, this input is not paired with all the other inputs. In that case, the answer is "no." Therefore, we achieve a polynomial solution to the *A2A mapping schema problem* for one and two identical-capacity reducers.

We now consider the case of $z > 2$ and prove that the *A2A mapping schema problem* for $z > 2$ reducers is at least as hard as the partition problem.

Theorem 4.5 *The problem of finding whether a mapping schema of m inputs of different input sizes exists, where every two inputs are assigned to at least one of $z \geq 3$ identical-capacity reducers, is NP-hard.*

The proof appears in Appendix B.

4.3.2 NP-hardness of the *X2Y Mapping Schema Problem*

Two lists of inputs, $X = \{i_1, i_2, \ldots, i_m\}$ whose input size list is $W_x = \{w_1, w_2, \ldots, w_m\}$ and $Y = \{i'_1, i'_2, \ldots, i'_n\}$ whose input size list is $W_y = \{w'_1, w'_2, \ldots, w'_n\}$, and a set of identical reducers $R = \{r_1, r_2, \ldots, r_z\}$ are an input instance to the *X2Y mapping schema problem*. The *X2Y mapping schema problem* is a decision problem that asks whether or not there exists a mapping schema for the given input instance such that each input of the list X is assigned with each input of the list Y to at least one reducer in common. An answer to the *X2Y mapping schema problem* will be "yes," if for each pair of inputs, the first from X and the second from Y, there is at least one reducer that has both those inputs.

The *X2Y mapping schema problem* has a polynomial solution for the case of a single reducer. If there is only one reducer, then the answer is "yes" if and only if the sum of the input sizes $\sum_{i=1}^{m} w_i + \sum_{i=1}^{n} w'_i$ is at most q. On the other hand, if $q < \sum_{i=1}^{m} w_i + \sum_{i=1}^{n} w'_i$, then the answer is "no." Next, we will prove that the *X2Y mapping schema problem* is an NP-hard problem for $z > 1$ identical reducers.

Theorem 4.6 *The problem of finding whether a mapping schema of m and n inputs of different input sizes that belongs to list X and list Y, respectively, exists, where every two inputs, the first from X and the second from Y, are assigned to at least one of $z \geq 2$ identical-capacity reducers, is NP-hard.*

The proof appears in Appendix B.

Chapter 5

Approximation Algorithms for the Mapping Schema Problems

In this chapter, we will provide approximation algorithms for the *A2A* and the *X2Y* mapping schema problems.

5.1 Preliminary Results

Since the *A2A Mapping Schema Problem* is NP-hard, we start looking at special cases and developing approximation algorithm to solve it. We propose several approximation algorithms for the *A2A mapping schema problem* that are based on bin-packing algorithms, selection of a prime number p, and division of inputs into two sets based on their sizes.

Each algorithm takes the number of inputs, their sizes, and the reducer capacity (see Table 5.2). The approximation algorithms have two cases depending on the sizes of the inputs, as follows:

1. Input sizes are upper bounded by $\frac{q}{2}$.
2. One input is of size, say w_i, greater than $\frac{q}{2}$, but less than q, and all the other inputs have size less than or equal to $q - w_i$. In this case most of the communication cost comes from having to pair the large input with every other input.

Of course, if the two largest inputs are greater than the given reducer capacity q, then there is no solution to the *A2A mapping schema problem* because these two inputs cannot be assigned to a single reducer in common.

Parameters for analysis. We analyze our approximation algorithms on the *communication cost*, which is the sum of all the bits that are required, according to the mapping schema, to transfer from the map phase to the reduce phase.

Table 5.1 summarizes all the results in this chapter. Before describing the algorithms,

Cases	Theorems	Communication cost	Approximation ratio
The **lower bounds** for the *A2A mapping schema problem*			
Different-sized inputs	5.1	$\frac{s^2}{q}$	
Equal-sized inputs	5.4	$m\lfloor\frac{m-1}{q-1}\rfloor$	
The **lower bounds** for the *X2Y mapping schema problem*			
Different-sized inputs	5.20	$\frac{2\cdot sum_x\cdot sum_y}{q}$	
Optimal algorithms for the *A2A mapping schema problem* (* equal-sized inputs)			
Algorithm for reducer capacity $q = 2$	5.9	$m(m-1)$	optimal
Algorithm for reducer capacity $q = 3$	5.9	$m\lfloor\frac{m-1}{2}\rfloor$	optimal
The *AU* method: When q is a prime number	5.9	$m\lfloor\frac{m-1}{q-1}\rfloor$	optimal
Non-optimal algorithms for the *A2A mapping schema problem* and their **upper bounds**			
Bin-packing-based algorithm, not including an input of size $> \frac{q}{2}$	5.3	$\frac{4s^2}{q}$	$\frac{1}{4}$
Algorithm 7	5.12	$\frac{q}{2k}\lceil\frac{sk}{q(k-1)}\rceil(\lceil\frac{sk}{q(k-1)}\rceil - 1)$	$1/k-1$
Algorithm 8: The first extension of the *AU method*	5.14	$qp(p+1)+z'$	$q/(q+1)$
Algorithm 9: The second extension of the *AU method*	5.18	$q^2\times(q(q+1))^{l-1}$	$(q^l-1)/q(q-1)(q+1)^{l-1}$
Bin-packing-based algorithm considering an input of size $> \frac{q}{2}$	5.19	$(m-1)\cdot q+\frac{4s^2}{q}$	$\frac{s^2}{mq^2}$
A non-optimal algorithm for the *X2Y mapping schema problem* and their **upper bounds**			
Bin-packing-based algorithm, $q = 2b$	5.21	$\frac{4\cdot sum_x\cdot sum_y}{b}$	$\frac{1}{4}$
Approximation ratio. The ratio between the optimal communication cost and the communication cost obtained from an algorithm. Notations: s: sum of all the input sizes. q: the reducer capacity. m: the number of inputs. sum_x: sum of input sizes of the list X. sum_y: sum of input sizes of the list Y. p: the nearest prime number to q. $l > 2$. $k > 1$.			

Table 5.1: The bounds for algorithms for the *A2A* and the *X2Y* mapping schema problems.

we look at lower bounds for the above parameters as they are expressed in terms of the reducer capacity q and sum of sizes of all inputs s.

Theorem 5.1 (Lower bounds on the communication cost and number of reducers) *For a list of inputs and a given reducer capacity q, the communication cost and the number of reducers, for the A2A mapping schema problem, are at least $\frac{s^2}{q}$ and $\frac{s^2}{q^2}$, respectively, where s is the sum of all the input sizes.*

The proof of the theorem is given in Appendix C.1.

5.1.1 Bin-packing-based approximation

Our general strategy for building approximation algorithms is as follows: we use a known bin-packing algorithm to place the given m inputs to bins of size $\frac{q}{k}$. Assume that we need

Algorithms	Inputs
Non-optimal algorithms for the *A2A mapping schema problem*	
Bin-packing-based algorithm	Any number of inputs of any size
Algorithm 7	Any number of inputs of size at most $\frac{q}{k}$, $k > 3$
Algorithm 8: The first extension of the *AU method*	$p^2 + p \cdot l + l, p + l = q, l > 2$
Algorithm 9: The second extension of the *AU method*	$q^l, l > 2$ and q is a prime number
A non-optimal algorithm for the *X2Y mapping schema problem*	
Bin-packing-based algorithm, $> \frac{q}{2}$	Any number of inputs of any size
Notations: w_i and w_j: the two largest size inputs of a list. p: the nearest prime number to q. w_k: the largest input of a list X. w'_k: the largest input of a list Y.	

Table 5.2: Reducer capacity and input constraints for different algorithms for the mapping schema problems.

x bins to place m inputs. Now, each of these bins is considered as a single input of size $\frac{q}{k}$ for our problem of finding an optimal mapping schema. Of course, the assumption is that all inputs are of size at most $\frac{q}{k}$.

First-Fit Decreasing (FFD) and Best-Fit Decreasing (BFD) [33] are most notable bin-packing algorithms. FFD or BFD bin-packing algorithm ensures that all the bins (except only one bin) are at least half-full. There also exists a pseudo polynomial bin-packing algorithm, suggested by Karger and Scott [71], that can place the m inputs in as few bins as possible of certain size.

For an example, let us discuss in more detail the case $k = 2$. In this case, since the reducer capacity is q, any two bins can be assigned to a single reducer. Hence, the approximation algorithm uses at most $\frac{x(x-1)}{2}$ reducers, where x is the number of bin; see Figure 5.1[1] for an example. For this strategy a lower bound on communication cost depends also on k as follows:

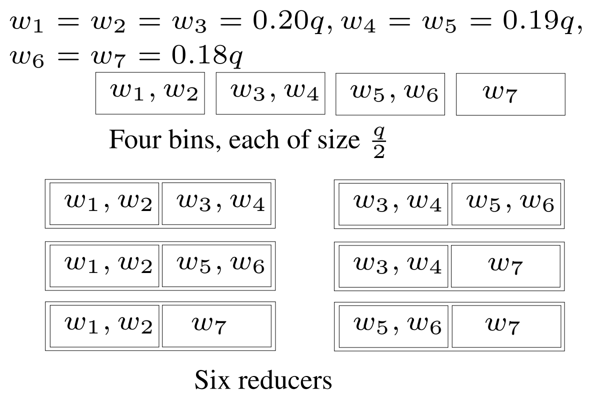


Figure 5.1: Bin-packing-based approximation algorithm.

[1]Note that this is not an optimal mapping schema for the given inputs.

Theorem 5.2 (Lower bound on the communication cost) *Let $q > 1$ be the reducer capacity, and let $\frac{q}{k}$, $k > 1$, is the bin size. Let the sum of the given inputs is s. The communication cost, for the A2A mapping schema problem, is at least $s\lfloor \frac{\frac{sk}{q}-1}{k-1} \rfloor$.*

The proof of the theorem is given in Appendix C.1. This communication cost in the above theorem, as expected, is larger than the one in Theorem 5.1, where no restriction in a specific strategy was taken into account.

Example for $k = 2$. Let us apply our strategy to the case where $k = 2$, *i.e.*, we have the algorithm: (*i*) we do bin-packing to put the inputs in bins of size $\frac{q}{2}$; and (*ii*) we provide a mapping schema for assigning each pair of bins to at least one reducer. Such a schema is easy and has discussed in the literature (*e.g.*, [105]).

FFD and BFD bin-packing algorithms provide an $\frac{11}{9} \cdot \textsc{Opt}$ approximation ratio [70], *i.e.*, if any optimal bin-packing algorithm needs $\textsc{Opt}$ bins to place (m) inputs in the bins of a given size $\frac{q}{2}$, then FFD and BFD bin-packing algorithms always use at most $\frac{11}{9} \cdot \textsc{Opt}$ bins of an identical size (to place the given m inputs). Since we require at most $\frac{x(x-1)}{2}$ reducers for a solution to the *A2A mapping schema problem*, the algorithm requires at most $\frac{(\frac{11}{9} \cdot \textsc{Opt})^2}{2}$ reducers. Note that, here in this case, $\textsc{Opt}$ does not indicate the optimal number of reducers to assign m inputs that satisfy the *A2A mapping schema problem*; $\textsc{Opt}$ indicates the optimal number of bins of size $\frac{q}{2}$ that are required to place m inputs.

The following theorem gives the upper bounds that this approximation algorithm achieves on the communication cost and the number of reducers.

Theorem 5.3 (Upper bounds on communication cost and number of reducers for $k = 2$ using bin-packing) *The above algorithm using a bin size $b = \frac{q}{2}$ where q is the reducer capacity achieves the following upper bounds: the number of reducers and the communication cost, for the A2A mapping schema problem, are at most $\frac{8s^2}{q^2}$ and at most $4\frac{s^2}{q}$, respectively, where s is the sum of all the input sizes.*

The proof of the theorem is given in Appendix C.1.

5.2 Optimal Algorithms for Equal-Sized Inputs

As we explained, looking at inputs of same size makes sense because we imagine the inputs are being bin-packed into bins of size $\frac{q}{k}$, for $k \geq 2$, and that once this is done, we can treat the bins themselves as things of unit size to be sent to the reducers. Thus, in this section, we will shift the notation so that all inputs are of unit size, and q is some small integer, *e.g.*, 3.

In this section, we provide optimal algorithms for $q = 2$ (in Section 5.2.1) and $q = 3$ (in Section 5.2.2). Afrati and Ullman [18] provided an optimal algorithm for the *A2A mapping schema problem* where q is a prime number and the number of inputs is $m = q^2$. We extend this algorithm for $m = q^2 + q + 1$ inputs (in Section 5.2.3), and this extension also meets the lower bound on the communication cost. We will generalize these three algorithms in the Sections 5.3 and 5.4.

In this setting, by minimizing the number of reducers, we minimize communication, since each reducer is more-or-less filled to capacity. So, we define: $r(m, q)$ to be the minimum number of reducers of capacity q that can solve the all-pairs problem for m inputs. The following theorem sets a lower bound on $r(m, q)$ and the communication cost for this setting.

Theorem 5.4 (Lower bounds on the communication cost and number of reducers) *For a given reducer capacity $q > 1$ and a list of m inputs of size one, the communication cost and the number of reducers ($r(m, q)$), for the A2A mapping schema problem, are at least $m\lfloor \frac{m-1}{q-1} \rfloor$ and at least $\lfloor \frac{m}{q} \rfloor \lfloor \frac{m-1}{q-1} \rfloor$, respectively.*

The proof of the theorem is given in Appendix C.2.

5.2.1 Reducer capacity $q = 2$

Here, we offer a recursive algorithm and show that this algorithm does not only obtain the bound $r(m, 2) \leq \frac{m(m-1)}{2}$, but it does so in a way that divides the reducers into $m - 1$ "teams" of $\frac{m}{2}$ reducers, where each team has exactly one occurrence of each input. We will use these properties of the output of this algorithm to build an algorithm for $q = 3$ in the next subsection.

The recursive algorithm. We are given a list A of m inputs. The intention is to have all pairs of inputs from list A partitioned into $m - 1$ teams with each team containing exactly $\frac{m}{2}$ pairs and each input appearing exactly once within a team. Hence, we will use $\frac{m(m-1)}{2}$ reducers for assigning pairs of each input.

We split A into two sublists A_1 and A_2 of size $\frac{m}{2}$ each. Suppose, we have the $\frac{m}{2} - 1$ teams for a list of size $\frac{m}{2}$. We will take the $\frac{m}{2} - 1$ teams of A_1, the $\frac{m}{2} - 1$ teams of A_2 and "mix them up" in a rather elaborate way to form the $m - 1$ teams for A:

Let the teams for A_1 and A_2 be $\{g_1, g_2, g_3, \ldots, g_{\frac{m}{2}}\}$ and $\{h_1, h_2, h_3, \ldots, h_{\frac{m}{2}}\}$ respectively. We will form two kind of teams, teams of kind I and teams of kind II as follows:

Teams of kind I. We will form $\frac{m}{2}$ teams of kind I by taking one input from A_1 and one input from A_2. For example, the first team for

A is $\{(g_1, h_1), (g_2, h_2), (g_3, h_3), \dots, (g_{\frac{m}{2}}, h_{\frac{m}{2}})\}$, the second team for A is $\{(g_1, h_2), (g_2, h_3), (g_3, h_4), \dots, (g_{\frac{m}{2}}, h_1)\}$, and so on.

Teams of kind II. We will form the remaining $\frac{m}{2} - 1$ teams having $\frac{m}{2}$ reducers in each. In teams of kind I each pair (reducer) contains only inputs from one of the lists A_1 or A_2. Now we produce pairs, with each pair having both inputs from A_1 or A_2. In order to do that, we divide recursively divide A_1 into two sublists and perform the operation what we performed in the team of kind I. The same procedure is recursively implemented on A_2.

Example 5.5 *For $m = 8$, we form 7 teams. First we form teams of kind I. We divide 8 inputs into two lists A_1 and A_2. After that, we take one input from A_1 and one input from A_2, and create 4 teams, see Figure 5.2a. Now, we recursively follow the same rule on each sublist, A_1 and A_2, and create 3 remaining teams of kind II, see Figure 5.2b.*

1,5	1,6	1,7	1,8
2,6	2,7	2,8	2,5
3,7	3,8	3,5	3,6
4,8	4,5	4,6	4,7
Team 1	Team 2	Team 3	Team 4

(a) Teams of kind I

1,3	1,4	1,2
2,4	2,3	3,4
5,7	5,8	5,6
6,8	6,7	7,8
Team 5	Team 6	Team 7

(b) Teams of kind II

Figure 5.2: The teams for $m = 8$ and $q = 2$.

Actually in Figure 5.3, the teams for this example are shown in non-bold face fonts (two in each triplet in Figure 5.3, notice that they are from 1-8) in teams 1 through 7 in Figure 5.3.

The following theorem is easy to prove.

Theorem 5.6 *1. In each team an input appears only once.*

2. *In each team all inputs appear.*
3. *There are $m - 1$ teams which is the minimum possible.*
4. *This is an optimal mapping scheme that assigns inputs to reducers.*[2]

This works if the number of inputs is a power of two. We can use known techniques to make it work with good approximation in general.

[2]Interested readers may see the proof of Algorithm 7A in Appendix C.5.1 for the proof of the statements (1) and (2). An input gets paired with the remaining $m-1$ inputs, and by the statement (2) an input can appear exactly once in a team. Hence, we need $m - 1$ teams to assign all the pairs of an input, resulting statement (3) is true. Since a team hold $\frac{m}{2}$ reducers that provide $\frac{m}{2}$ pairs and there are total $\frac{m(m-1)}{2}$ pairs of inputs, we are using $\frac{m(m-1)}{2}$ reducers. Thus. this is an optimal mapping schema.

5.2.2 Reducer capacity $q = 3$

Here, we present an algorithm that constructs an optimal mapping schema for $q = 3$. Our recursive algorithm starts by taking the mapping schema constructed in previous subsection for $q = 2$. We showed there that for $q = 2$, we can not only obtain the bound $r(m, 2) \leq \frac{m(m-1)}{2}$, but that we can do so in a way that divides the reducers into $m - 1$ teams of $\frac{m}{2}$ reducers in each team, where each team has exactly one occurrence of each input.

Now, we split m inputs into two disjoint sets: set A and set B. Suppose $m = 2n - 1$. Set A has n inputs and set B has $n-1$ inputs. We start with the n inputs in set A, and create $n - 1$ teams of $\frac{n}{2}$ reducers, each reducer getting two of the n inputs in A, by following the algorithm given in Section 5.2.1. Next, we add to all reducers in one team another input from set B. *I.e.*, in a certain team we add to all $\frac{n}{2}$ reducers of this team a certain input from set B, and thus, we form a triplet for each reducer.

Since there are $n - 1$ teams, we can handle another $n - 1$ inputs. This is the start of a solution for $q = 3$ and $m = 2n - 1$ inputs. To complete the solution, we add the reducers for solving the problem for the $n - 1$ inputs of the set B. That leads to the following recurrence

$$\begin{aligned} r(m, 3) &= \frac{n(n-1)}{2} + r(n-1, 3), \text{ where } m = 2n - 1 \\ r(3, 3) &= 1 \end{aligned}$$

We solve the recurrence for m a power of 2, and it exactly matches the lower bound of $r(m, 3) = \frac{m(m-1)}{6}$. Moreover, notice that we can prove that this case is optimal either by proving that $r(m, 3) = m(m-1)/6$ (as we did above) or by observing that every pair of inputs meets exactly in one reducer. This is easy to prove. Hence the following theorem:

Theorem 5.7 *This algorithm constructs an optimal mapping schema for $q = 3$.*

Example 5.8 *An example is shown in Figure 5.3. We explained how this figure is constructed for $q = 2$ (the non-bold entries). Now we use the algorithm just presented here to construct the 35 ($= 15 \times \frac{14}{6}$) reducers. We explain below in detail how we construct these 35 reducers.*

We are given 15 inputs ($I = \{1, 2, \ldots, 15\}$). We create two sets, namely A of $y = 8$ inputs and B of $x = 7$ inputs, and arrange $(y - 1) \times \lceil \frac{y}{2} \rceil = 28$ reducers in the form of 7 teams *of* 4 *reducers in each team. These 7 teams assign each input of the set A with all other inputs of the set A and all the inputs in the set B as follows. We pair every two inputs of the set A and assign them to exactly one of 28 reducers as we explained in Section 5.2.1.*

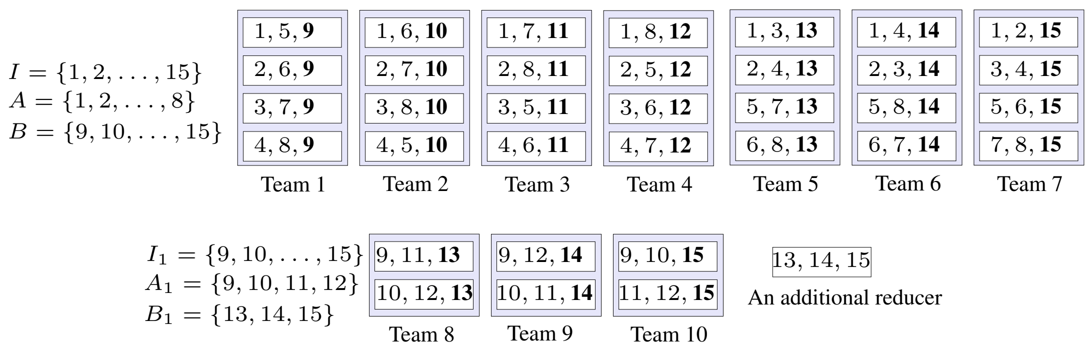


Figure 5.3: An example of a mapping schema for $q = 3$ and $m = 15$.

Once every pair of $y = 8$ inputs of the set A is assigned to exactly one of 28 reducers, then we assign the i^{th} input of the set B to all the four reducers of $(i - 8)^{th}$ team. Thus, e.g., input 10 is assigned to the four reducers of Team 2.

Now these 28 reducers have seen that each pair of inputs from set A meet in at least one reducer and each pair of inputs, one from A and one from B meet in at least one reducer. Thus, it remains to build more reducers so that each pair of inputs (both) from set B meet. According to the recursion we explained, we break set B into sets A_1 and B_1, of size 4 and 3 respectively, and we apply our method again. In particular, we create two sets, $A_1 = \{9, 10, 11, 12\}$ of $y_1 = 4$ inputs and $B_1 = \{13, 14, 15\}$ of $x_1 = 3$. Then, we arrange $(y_1 - 1) \times \lceil \frac{y_1}{2} \rceil = 6$ reducers in the form of 3 teams *of* 2 *reducers in each team. We assign each pair of inputs of the set A_1 to these 6 reducers, and then i^{th} input of the set B_1 to all the two reducers of a team, see Team 8 to Team 10.*

The last team is constructed so that all inputs in B_1 meet at the same reducers (since B_1 has only 3 elements and 3 is the size of a reducer, one reducer suffices for this to happen).

Open Problem. Now the interesting observation is that if we can argue that the resulting reducers can be divided into $\frac{m-1}{2}$ teams of $\frac{m}{3}$ reducers each (with each team having one occurrence of each input), then we can extend the idea to $q = 4$, and perhaps higher.

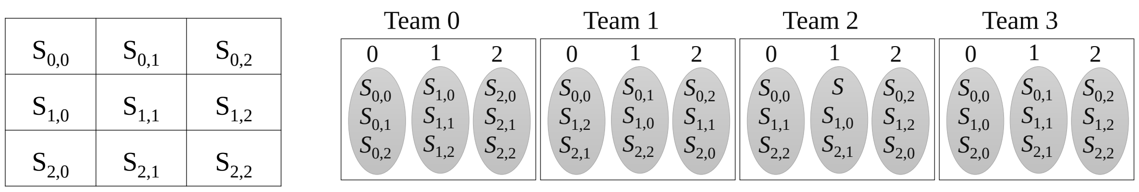
5.2.3 When q or $q - 1$ is a prime number

An algorithm to provide a mapping schema for the reducer capacity q, where q is a prime number, and $m = q^2$ inputs is suggested by Afrati and Ullman in [18]. This method meets the lower bounds on the communication cost. We call this algorithm the *AU method*.

For the sake of completeness, we provide an overview of the *AU method*. Interested readers may refer to [18]. We divide the m inputs into q^2 equal-sized *subsets* (each with $\frac{m}{q^2}$ inputs) that are arranged in a $Q = q \times q$ square. The subsets in row i and column j are represented by $S_{i,j}$, where $0 \leq i < q$ and $0 \leq j < q$.

We now organize $q(q+1)$ reducers in the form of $q+1$ *teams* of q *players* (or reducers) in each team. Note that sum of sizes of the inputs in each row and column of the Q square is exactly q.

The teams are arranged from 0 to q, and the reducers are arranged from 0 to $q-1$. We first arrange inputs to the team q. Since the sum of the sizes in each column of the P square is q, we place one column of the P square to one reducer of the team q. Now we place the inputs to the remaining teams. We use modulo operation for the assignment of each subset to each team. The subset $S_{i,j}$ is assigned to a reducer r of each team t, $0 \leq t < q$, such that $(i+tj) modulo\, q = r$. An example for $q = 3$ and $m = 9$ is given in Figure 5.4.



$S_{0,0}$ $S_{0,1}$ $S_{0,2}$
$S_{1,0}$ $S_{1,1}$ $S_{1,2}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$
Team 0
0 1 2
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$
$S_{1,0}$ $S_{1,1}$ $S_{1,2}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$
Team 1
0 1 2
$S_{0,0}$ $S_{1,2}$ $S_{2,1}$
$S_{0,1}$ $S_{1,0}$ $S_{2,2}$
$S_{0,2}$ $S_{1,1}$ $S_{2,0}$
Team 2
0 1 2
$S_{0,0}$ $S_{1,1}$ $S_{2,2}$
S $S_{1,0}$ $S_{2,1}$
$S_{0,2}$ $S_{1,2}$ $S_{2,0}$
Team 3
0 1 2
$S_{0,0}$ $S_{1,0}$ $S_{2,0}$
$S_{0,1}$ $S_{1,1}$ $S_{2,1}$
$S_{0,2}$ $S_{1,2}$ $S_{2,2}$

Figure 5.4: The *AU method* for the reducer capacity $p = 3$ and $m = 9$.

Total required reducers. The *AU method* uses $q(q+1)$ reducers, which are organized in the form of $q+1$ teams of q reducers in each team, and the communication cost is $q^2(q+1)$.
A simple extension of the *AU method*. Now, we can extend the *AU method* as follows: we can add $q+1$ additional inputs, add one to each reducer and add one more reducer that has the $q+1$ new inputs. That gives us reducers of size $q = q+1$ and $m = q^2+q+1$, or $r(q^2+q+1, q+1) = q(q+1)+1 = q^2+q+1$. If you substitute $m = q^2+q+1$ and $p = p+1$, you can check that this also meets the bound of $r = \frac{m(m-1)}{q(q-1)}$. In Figure 5.5, we show a mapping schema for this extension to the *AU method* for $q = 4$ and $m = 14$.

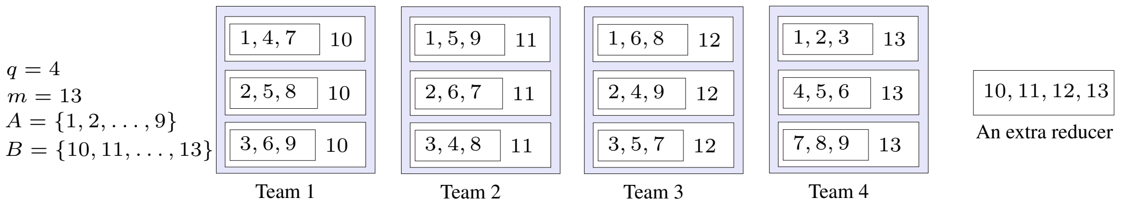


Figure 5.5: An optimum mapping schema for $q = 4$ and $m = 14$ by extending the *AU method*.

In conclusion, in this section we have shown the following:

Theorem 5.9 *We can construct optimal mapping schemas for the following cases:*

1. $q = 2$.
2. $q = 3$.
3. *q being a prime number and $m = q^2$.*
4. *$q - 1$ being a prime number and $m = (q-1)^2 + q$.*

Open Problem: Can we generalize the last idea to get optimal schemas for more cases?

Approximation Algorithms for the *A2A Mapping Schemas Problem*. We can use the optimal mapping schemas of Section 5.2 to construct good approximation of mappings schemas in many cases. The general techniques, we will use in this section move along the following dimensions/ideas:

- Assuming that there are no inputs of size greater than $\frac{q}{k}$, construct bins of size $\frac{q}{k}$, and treat each of the bins as a single input of size 1 and assume the reducer capacity is k. Then apply one of the optimal techniques of Section 5.2 to construct a mapping schema. These algorithms are presented in Sections 5.3 and 5.5.
- Getting inspiration from the methods developed (or only presented – in the case of the *AU method*) in Section 5.2.3, we extend the ideas to construct good approximation algorithms for inputs that are all of equal size (see Sections 5.4.1 and 5.4.2).

Thus, in Sections 5.3, 5.4, and 5.5, we will give several such techniques and show that some of them construct mapping schemas close to the optimal. To that end, we have already shown a schema based on bin-packing algorithms in Section 5.1.1.

5.3 Generalizing the Technique for the Reducer Capacity $q > 3$ and Inputs of Size $\leq q/k$, $k > 3$

In this section, we will generalize the algorithm for $q = 3$ given in Section 5.2.2 and present an algorithm (Algorithm 7) for inputs of size less than or equal to $\frac{q}{k}$ and $k > 3$. For simplicity, we assume that k divides q evenly throughout this section.

5.3.1 Algorithm 7A

We divide Algorithm 7 into two parts based on the value of k as even or odd. Algorithm 7A considers that k is an odd number. Pseudocode of Algorithm 7A is given in Appendix C.3. Algorithm 7A works as follows:

First places all the given inputs, say m', to some bins, say m, each of size $\frac{q}{k}$, $k > 3$ is an odd number. Thus, a reducer can hold an odd number of bins. After placing all the m' inputs to m bins, we can treat each of the m bins as a single input of size one and the reducer capacity to be k. Now, it is easy to turn the problem to a case similar to the case of $q = 3$. Hence, we divide the m bins into two sets A and B, and follow a similar approach as given in Section 5.2.2.

Aside. Equivalently, we can consider q to be odd and the inputs to be of unit size. In what follows, we will continue to use q, which is an odd number, as the reducer capacity and assume all inputs (that are actually bins containing inputs) are of unit size.

Example 5.10 *If $q = 30$ and $k = 5$, then we can pack given inputs to some bins of size 6. Hence, a reducer can hold 5 bins. Equivalently, we may consider each of the bins as a single input of size 1 and $q = 5$.*

For understanding of Algorithm 7A, an example for $q = 5$ is presented in Figure 5.6, where we obtain $m = 23$ bins (that are considered as 23 unit-sized inputs) after implementing a bin-packing algorithm to given inputs.

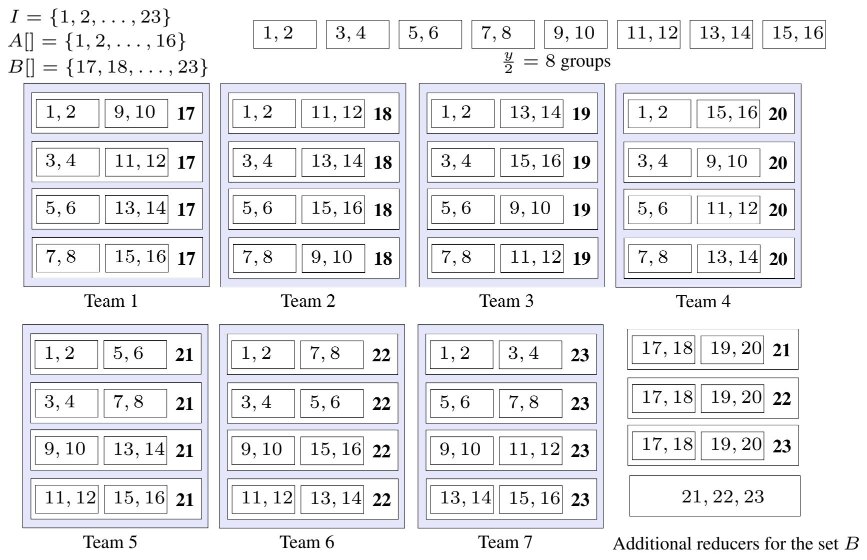


Figure 5.6: Algorithm 7 – an example of a mapping schema for $q = 5$ and 23 bins.

Algorithm 7A consists of six steps as follows:

1. *Implement a bin-packing algorithm*: Implement a bin-packing algorithm to place all the given m' inputs to bins of size $\frac{q}{k}$, where $k > 3$ and the size of all the inputs is less than or equal to $\frac{q}{k}$. Let m bins are obtained, and now each of the bins is considered as a single input.
2. *Division of bins (or inputs) to two sets, A and B*: Divide m inputs into two sets: A of size $y = \lfloor \frac{q}{2} \rfloor (\lfloor \frac{2m}{q+1} \rfloor + 1)$ and B of size $x = m - y$.
3. *Grouping of inputs of the set A*: Group the y inputs into $u = \lceil \frac{y}{q - \lceil q/2 \rceil} \rceil$ disjoint groups, where each group holds $\lceil \frac{q-1}{2} \rceil$ inputs. (We consider each of the u ($= \lceil \frac{y}{q - \lceil q/2 \rceil} \rceil$) disjoint groups as a single input that we call the *derived input*. By making u disjoint groups[3] (or derived inputs) of y inputs of the set A, we turn the case of any odd value of q to a case

[3]We suppose that u is a power of 2. In case u is not a power of 2 and $u > q$, we add dummy inputs each of size $\lceil \frac{q-1}{2} \rceil$ so that u becomes a power of 2. Consider that we require d dummy inputs. If groups of inputs of the set B each of size $\lceil \frac{q-1}{2} \rceil$ are less than equal to d dummy inputs, then we use inputs of the set B in place of dummy inputs, and the set B will be empty.

where a reducer can hold only three inputs, the first two inputs are pairs of the derived inputs and the third input is from the set B.)

4. *Assigning groups (inputs of the set A) to some reducers*: Organize $(u-1) \times \lceil \frac{u}{2} \rceil$ reducers in the form of $u-1$ teams of $\lceil \frac{u}{2} \rceil$ reducers in each team. Assign every two groups to one of $(u-1) \times \lceil \frac{u}{2} \rceil$ reducers. To do so, we will prove the following Lemma 5.11.

   Lemma 5.11 *Let q be the reducer capacity. Let the size of an input is $\lceil \frac{q-1}{2} \rceil$. Each pair of $u = 2^i$, $i > 0$, inputs can be assigned to $2^i - 1$ teams of 2^{i-1} reducers in each team.*[4]

5. *Assigning inputs of the set B) to the reducers*: Once every pair of the derived inputs is assigned, then assign i^{th} input of the set B to all the reducers of i^{th} team.
6. *Use previous steps on the inputs of the set B*: Apply (the above mentioned) steps 1-4 on the set B until there is a solution to the *A2A mapping schema problem* for the x inputs.

Algorithm correctness. The algorithm correctness appears in Appendix C.5.1.

Theorem 5.12 (The communication cost obtained using Algorithm 7A or 7B) *For a given reducer capacity $q > 1$, $k > 3$, and a set of m inputs whose sum of sizes is s, the communication cost, for the A2A mapping schema problem, is at most $\frac{q}{2k} \lceil \frac{sk}{q(k-1)} \rceil (\lceil \frac{sk}{q(k-1)} \rceil - 1)$.*

The proof of the theorem is given in Appendix C.5.

Approximation factor. The optimal communication cost (from Theorem 5.2) is $s\lfloor(\frac{sk}{q} - 1)/k - 1\rfloor \approx \frac{s^2}{q} \cdot \frac{k}{k-1}$ and the communication cost of the algorithm (from Theorem 5.12) is $\frac{q}{2k} \lceil \frac{sk}{q(k-1)} \rceil (\lceil \frac{sk}{q(k-1)} \rceil - 1) \approx s^2k/q(k-1)^2$. Thus, the ratio between the optimal communication and the communication of our mapping schema is approximately $\frac{1}{k-1}$.

5.3.2 Algorithm 7B

For the sake of completeness, we include the pseudocode of the algorithm for handling the case when k is an even number. We call it Algorithm 7B and pseudocode is given in Appendix C.4. In this algorithm, we are given m' inputs of size less than or equal to $\frac{q}{k}$ and $k \geq 4$ is an even number.

Similar to Algorithm 7A, Algorithm 7B first places all the m' inputs to m bins, each of size $\frac{q}{k}$, $k > 2$ is an even number. Thus, a reducer can hold an even number of bins. After placing all the m' inputs to m bins, we can treat each of the m bins as a single input of size one and the reducer capacity to be k. Now, we easily turn this problem to a case similar to the case of $q = 2$. Hence, we divide the m bins into two set A and B, and follow a similar approach as given in Section 5.2.1.

[4]The proof appears in Appendix C.5.

Example 5.13 *If* $q = 30$ *and* $k = 6$*, then we can pack given inputs to some bins of size* 5*. Hence, a reducer can hold 6 bins. Equivalently, we may consider each of the bins as a single input of size 1 and* $q = 6$*.*

Note. Algorithms 7A and 7B are based on a fact that how do we pack inputs in a *well* manner to bins of even or odd size. To understand this point, consider $q = 30$ and $m' = 46$. For simplicity, we assume that all the inputs are of size three. Now, consider $k = 5$, so we will use 23 bins each of size 6 and apply Algorithm 7A. On the other, consider $k = 6$, so we will use 46 bins each of size 5 and apply Algorithm 7B.

5.4 Generalizing the *AU method*

In this section, we extend the *AU method* (Section 5.2.3) to handle more than q^2 inputs, when q is a prime number, Algorithms 8 and 9. Recall that the *AU method* can assign each pair of q^2 inputs to reducers of capacity q. We provide two extensions: (*i*) take $m = p^2 + p \cdot l + l$ identical-sized inputs and assign these inputs to reducers of capacity $p + l = q$, where p is the nearest prime number to q, in Section 5.4.1, and (*ii*) take $m = q^l$ inputs, where $l > 2$, and assign inputs to reducers of capacity q, in Section 5.4.2. Pseudocodes of Algorithms 8 and 9 are given in Appendix C.6 and Appendix C.7, respectively.

5.4.1 When we consider the nearest prime to q

We provide an extension to the *AU method* that handles $m = p^2 + p \cdot l + l$ identical-sized inputs and assigns them to reducers of capacity $p + l = q$, where p is the nearest prime number to q. We call it the *first extension to the AU method* (Algorithm 8).

Algorithm 8: The *First Extension of the AU method.* We extend the *AU method* by increasing the reducer capacity and the number of inputs, see Algorithm 8. Consider that the *AU method* assigns p^2 identical-sized inputs to reducers of capacity p, where p is a prime number. We add $l(p + 1)$ inputs and increase the reducer capacity to $p + l$ $(= q)$.

In other words, m identical-sized inputs and the reducer capacity q are given. We select a prime number, say p, that is near most to q such that $p + l = q$ and $p^2 + l(p + 1) \leq m$. Also, we divide the m inputs into two disjoint sets A and B, where A holds at most p^2 inputs and B holds at most $l(p + 1)$ inputs.

Algorithm 8 consists of six steps, as follows:

1. Divide the given m inputs into two disjoint sets A of $y = p^2$ inputs and B of $x = m - y$ inputs, where p is the nearest prime number to q such that $p+l = q$ and $p^2+l(p+1) \leq m$.
2. Perform the *AU method* on the inputs of the set A by placing y inputs to $p + 1$ teams of p bins in each team, where the size of each bin is p.

3. Organize $p(p+1)$ reducers in the form of $p+1$ teams of p reducers in each teams, and assign j^{th} bin of i^{th} team of bins to j^{th} reducer of i^{th} team of reducers.
4. Group the x inputs of the set B into $u = \lceil \frac{x}{q-p} \rceil$ disjoint groups.
5. Assign i^{th} group to all the reducers of i^{th} team.
6. Use Algorithm 7A or Algorithm 7B to make each pair of inputs of the set B, depending on the case of the value of q, which is either an odd or an even number, respectively.

Note that when we perform the above mentioned step 3, we assign each pair of inputs of the set A to $p(p+1)$ reducers, and such an assignment uses p capacity of each reducer. Now, each of $p(p+1)$ reducers has $q-p$ remaining capacity that is used to assign i^{th} group of inputs of the set B. Thus, all the inputs of the set A are assigned with all the m inputs.
Algorithm correctness. The algorithm correctness appears in Appendix C.6.1.

Theorem 5.14 (The communication cost obtained using Algorithm 8) *Algorithm 8 requires at most $p(p+1)+z$ reducers, where $z = \frac{2l^2(p+1)^2}{q^2}$, and results in at most $qp(p+1)+z'$ communication cost, where $z' = \frac{2l^2(p+1)^2}{q}$, q is the reducer capacity, and p is the nearest prime number to q.*

The proof of the theorem is given in Appendix C.6. When $l = q-p$ equals to one, we have provided an extension of the *AU method* in Section 5.2.3, and in this case, we have an optimum mapping schema for q and $m = q^2+q+1$ inputs.
Approximation factor. The optimal communication cost using the *AU method* is $q^2(q+1)$. Thus, the difference between the communication of our mapping schema ($q^2(q+1)+z'$, when assuming p is equal to q) and the optimal communication is z'. We can see two cases, as follows:

1. When q is large. Consider that q is greater than square or cube of the maximum difference between any two prime numbers. In this case, z' will be very small, and we will get almost optimal ratio.
2. When q is very small. In this case, then z' plays a role as follows: here, the number of inputs in the set B will be at most $(p+1)l < q^2$. Thus, the ratio becomes $q/(q+1)$.

5.4.2 For input size $m = q^l$

We also provide another extension to the *AU method* that handles $m = q^l$ identical-sized inputs and assigns them to reducers of capacity q, where q is a prime number and $l > 2$. We call it the *second extension to the AU method* (Algorithm 9, refer to Appendix C.7).
Algorithm 9: The *Second Extension of the AU method.* The second extension to the *AU method* (Algorithm 9) handles a case when $m = q^l$, where $l > 2$ and q is a prime number. We present Algorithm 9 for $m = q^l$, $l > 2$, inputs and the reducer capacity q, where q is a

prime number. Nevertheless, m inputs that are less than but close to q^l can also be handled by Algorithm 9 by adding dummy inputs such that $m = q^l$, $l > 2$.

Algorithm 9 consists of two phases, as follows:

The first phase: creation of a bottom up tree. Here, we present a simple example for the bottom-up tree's creation for $q = 3$ and $m = 3^4$; see Figure 5.7.

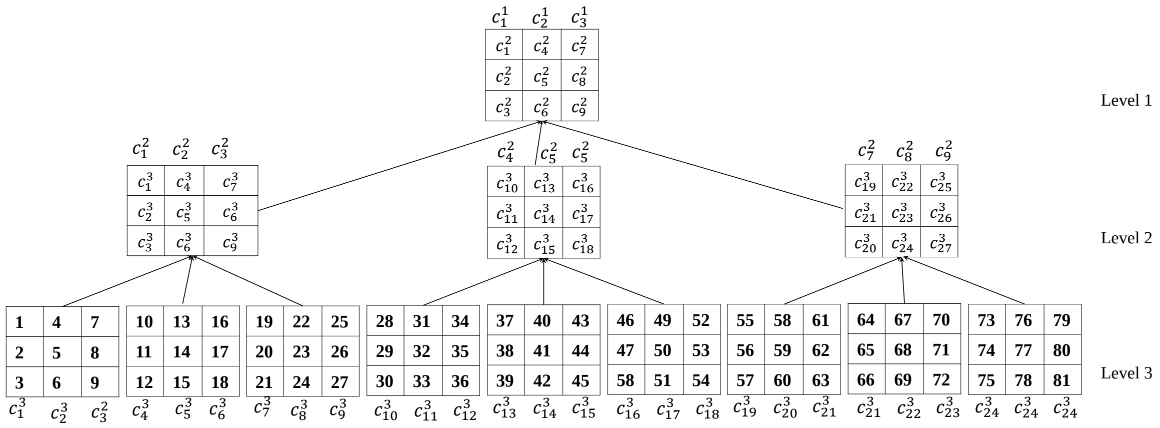


Figure 5.7: The *second extension of the AU method* (Algorithm 9): Phase 1 – Creation of the bottom-up tree.

Example 5.15 (Bottom-up tree creation) *A bottom-up tree for $m = q^l = 3^4$ identical-sized inputs and $q = 3$ is given in Figure 5.7. Here, we explain how we constructed it.*

The height of the bottom up tree is $l - 1$, and the last $(l - 1)^{th}$ level has m inputs in the form of $\frac{m}{q^2}$ matrices of size $q \times q$. Note that we have $\frac{m}{q}$ columns at the last level, which holds m inputs; and these $\frac{m}{q}$ columns are called the input columns. *We create the tree in bottom-up fashion, where $(l - 2)^{th}$ level has $\frac{m}{q^3}$ matrices, whose each cell value refers to a* input column *of $(l - 1)^{th}$ level. We use a notation to refer a column of i^{th} level by c^i_j, where j is column index. Note that each column, c^i_j, at level i holds q columns ($c^{i+1}_{(j-1)q+1}, c^{i+1}_{(j-1)q+2}, \ldots c^{i+1}_{jq}$) of $(i+1)^{th}$ level. In general, there are $\frac{m}{q^{l-i+2}}$ matrices at level i, whose each cell value, c^i_j, refers to a column, c^{i+1}_j, of $(i+1)^{th}$ level.*

Following that the bottom-up tree for $m = 3^4$ identical-sized inputs and $q = 3$ has height 3. The last level ($(l-1)^{th} = 3^{rd}$) has 81 inputs in the form of $\frac{m}{q^2} = 9$ matrices of size 3×3. Note that we have $\frac{m}{q} = 24$ columns at 3^{rd} level; called the input columns. *The $l - 2 = 2^{ed}$ level has $\frac{m}{q^3} = 3$ matrices, whose each column, c^2_j, refers to $q = 3$ columns ($c^3_{(j-1)q+1}, c^3_{(j-1)q+2}, \ldots c^3_{jq}$) of 3^{rd} level. Further, the root node is at level 1, whose each column, c^1_j, refers to $q = 3$ columns ($c^2_{(j-1)q+1}, c^2_{(j-1)q+2}, \ldots c^2_{jq}$) of 2^{ed} level.*

The second phase: creation of an assignment tree. The assignment tree is created in top-down fashion. Our objective is to assign each pair of inputs to a reducer, where inputs are arranged in the *input columns* of the bottom-up tree. If we can assign each pair of *input columns* (of the bottom-up tree) in the form of $(q \times q)$-sized matrices, then the implementation of the *AU method* on each such matrices results in an assignment of every pair of inputs to reducers. Hence, we try to make pairs of all the *input columns* by creating a tree called the *assignment tree*.

Here, we present a simple assignment tree for $m = 3^4$ and $q = 3$ (see Figure 5.8).

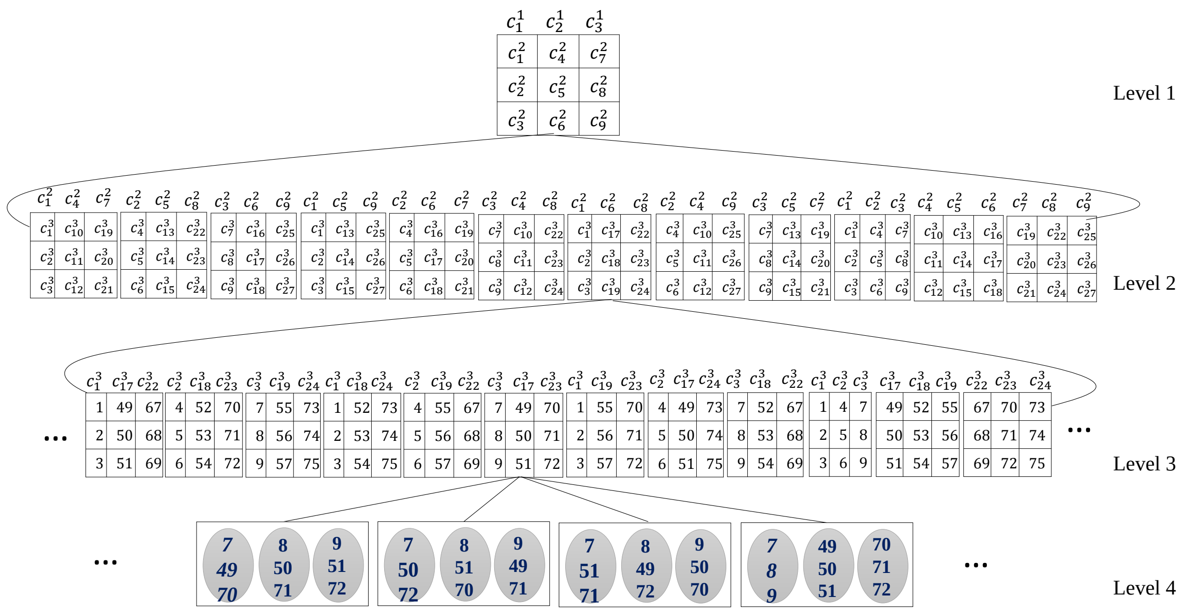


Figure 5.8: The *second extension of the AU method* (Algorithm 9): Phase 2 – Creation of the assignment tree.

Example 5.16 (Assignment tree creation) *The root node of the bottom-up tree becomes the root node the assignment tree. Recall that the root node of the bottom-up tree is a $q \times q$ matrix. First, consider the root node to understand the working of the* AU method *to create the assignment tree. Consider that each cell value of the root node matrix is of size one, and we have $(q + 1)$ teams of q bins (of size q) in each team. Our objective to use the* AU method *on the root node matrix is to assign each pair of cell values ($\langle c^2_x, c^2_y \rangle$) in $q(q + 1)$ bins that results in an assignment of every pair of cell values $\langle c^2_x, c^2_y \rangle$ at a bin.*

Now, we create matrices by using these bins (the bins created by the AU method*'s implementation on the root node) that are holding the indices of columns of the second level (c^2_x) of the bottom-up tree. We take each bin and its q indices $c^2_j, c^2_{j+1}, \dots c^2_{j+q}$. We replace each c^2_j with q columns as: $c^3_{(j-1)q+1}, c^3_{(j-1)q+2}, \dots c^3_{jq}$ that results in $q(q + 1)$ matrices of size $q \times q$, and these $q(q + 1)$ matrices become child nodes of the root node. Now, we consider each such matrix separately and perform a similar operation as we did for the root node.*

In this manner, the AU method *creates* $(q(q+1))^{i-1}$ *child nodes (that are matrices of size* $q \times q$*) at* i^{th} *level of the assignment tree, and they create* $(q(q+1))^{i}$ *child nodes (matrices of size* $q \times q$*) at* $(i+1)^{th}$ *level of the assignment tree.*

Recall that there are $\frac{m}{q}$ input column*s at* $(l-1)^{th}$ *level of the bottom-up tree that hold the original* m *inputs. The implementation of the* AU method *on each node (*$q \times q$*-sized) matrix of* $(l-2)^{th}$ *level of the assignment tree assigns each pair of* input columns *at* $(l-1)^{th}$ *level of the assignment tree. Further the* AU method*'s implementation on each matrix of* $(l-1)^{th}$ *level assigns every pairs of the original inputs to* $q^l \times (q+1)^{l-1}$ *reducers at* l^{th} *level, which have reducers in the form of* $(q(q+1))^{l-1}$ *teams of* q *reducers in each team.*

For $m = 3^4$ *identical-sized inputs and* $q = 3$*, we take the root node of the bottom-up tree (Figure 5.7) that becomes the root node of the assignment tree. We implement the* AU method *on the root node and assign each pair of cell values (*c_j^2*,* $1 \leq j \leq 9$*) to a bin of size* q*. Each cell value of the bins (*c_j^2*) is then placed by* $q = 3$ *columns* $c^3_{(j-1)q+1}, c^3_{(j-1)q+2}, \ldots c^3_{jq}$ *that results in an assignment of each pair of columns of the second level of the bottom-up tree. For clarity, we are not showing bins. For the next* 3^{rd} *level, we again implement the* AU method *on all 12 matrices at* 2^{nd} *level and get 144 matrices at the third level. The matrices at* 3^{rd} *level are pairs of each* input columns *(of the bottom-up tree). The* AU method*'s implementation on each matrix of* 3^{rd} *level assigns each pair of original inputs to reducers. For clarity, we are only showing all the matrixes and teams at levels 3 and 4, respectively.*

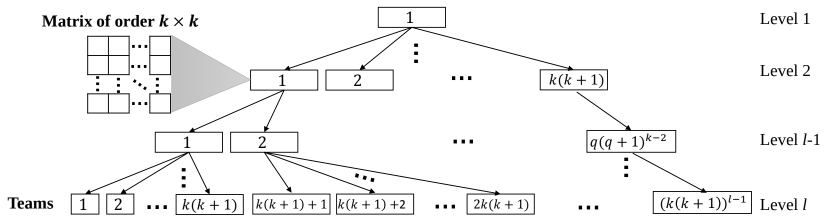


Figure 5.9: An assignment tree created using Algorithm 9.

The assignment tree uses the root node of the bottom up tree, and we implement the *AU method* on the root node that results in $q(q+1)$ child nodes at level two. Each child node is a $q \times q$ matrix, and the columns of all the $q(q+1)$ matrices provide all-pairs of the cell values of the root node matrix. At level i, the assignment tree has $(q(q+1))^{i-1}$ nodes, see Figure 5.9. The height of the assignment tree is l, where $(l-1)^{th}$ level has all-pairs of *input columns* and l^{th} level has a solution to the *A2A mapping schema problem* for m inputs.
Algorithm correctness. Algorithm 9 satisfies the following Lemma 5.17:

Lemma 5.17 *The height of the assignment tree is* l*, and* l^{th} *level of the assignment tree assigns each pairs of inputs to reducers.*

Theorem 5.18 (The communication cost obtained using Algorithm 9) *Algorithm 9 requires at most $q \times (q(q+1))^{l-1}$ reducers and results in at most $q^2 \times (q(q+1))^{l-1}$ communication cost.*

The proof of the theorem is given in Appendix C.7.

Approximation factor. The optimal communication is $\frac{m(m-1)}{q-1}$ (see Theorem 5.4). Replacing m with q^l we get $q^l(q^l-1)/(q-1)$. Thus, the ratio between the optimal communication and the communication of our mapping schema is $(q^l-1)/q(q-1)(q+1)^{l-1}$. We can see two cases:

1. When q is large. Then we drop the constant 1 and the ratio is approximately equal to $\frac{1}{q}$.
2. When q is very small compared to q^l. Then the ratio is $q^l/q(q-1)(q+1)^{l-1}$.
   For $q = 5$, the inverse of the ratio is approximately $(6/5)^{l-1}$. This is already acceptable for practical applications if we think that the size of data is 5^l, thus l may as well be $l = 9$, in which case this ratio is approximately 4.3. For $q = 2$ and $q = 3$ we already have optimal mappings schemas. Our conjecture is that there are optimal schemas for $q = 4$ and $q = 5$ even by using the techniques developed and presented here.

5.5 A Hybrid Algorithm for the *A2A Mapping Schema Problem*

In the previous sections, we provide algorithms for different-sized and almost equal-sized inputs. The hybrid approach considers both different-sized and almost equal-sized inputs together. The objective of the hybrid approach is to place inputs to two different-sized bins, and then consider each of the bins as a single input.

Specifically, the hybrid approach uses the previously given algorithms (bin-packing-based approximation algorithm) and Algorithms 7A, 7B, 8, 9. We divide the given m inputs into two disjoint sets according to their input size, and then use the bin-packing-based approximation algorithm and Algorithms 7A, 7B, 8, or 9 depending on the size of inputs.

Hybrid Algorithm. We divide m inputs into two sets A that holds the input i of size $\frac{q}{3} < w_i \leq \frac{q}{2}$, and B holds all the inputs of sizes less than or equal to $\frac{q}{3}$. The algorithm consists of four steps, as follows:

1. Use the bin-packing-based approximation algorithm to place all the inputs of:
   (a) the set A to bins of size $\frac{q}{2}$, and each such bin is considered as a single input of size $\frac{q}{2}$ that we call the *big input*. Consider that x big inputs are obtained.
   (b) the set B twice, first to bins of size $\frac{q}{2}$, where each bin is considered as a single input of size $\frac{q}{2}$ that we call the *medium input*, and second, to bins of size $\frac{q}{3}$, where each bin

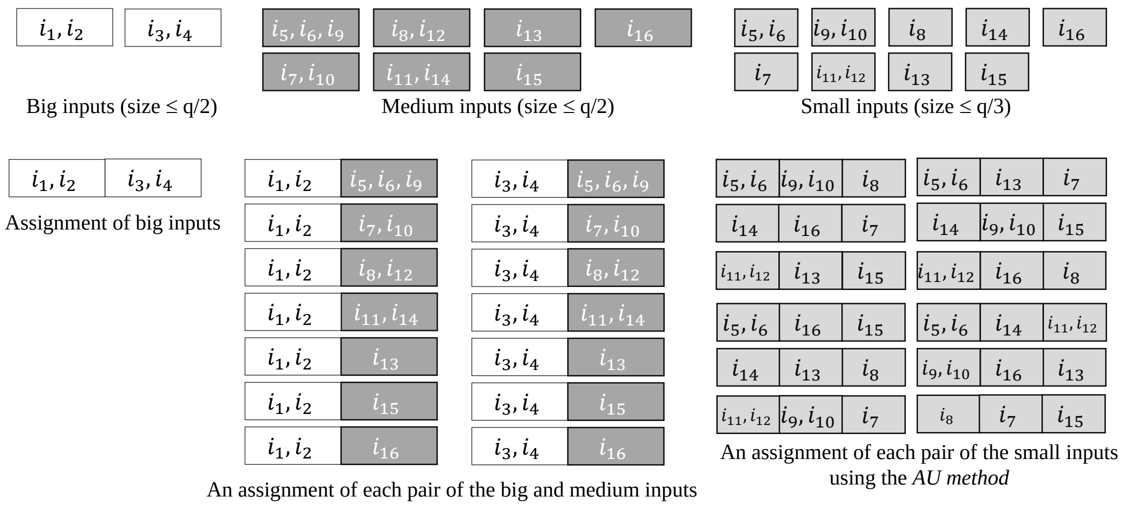

Big inputs (size ≤ q/2)
Medium inputs (size ≤ q/2)
Small inputs (size ≤ q/3)
Assignment of big inputs
An assignment of each pair of the big and medium inputs
An assignment of each pair of the small inputs using the *AU method*

Figure 5.10: An example to show the working of the hybrid algorithm. We are given 15 inputs, where inputs i_1 to i_4 are of sizes greater than $\frac{q}{3}$, and all the other inputs are of sizes less than or equal to $\frac{q}{3}$.

is also considered as a single input of size $\frac{q}{3}$ that we call the *small input*. Consider that y medium and z small inputs are obtained.

2. Use $\frac{x(x-1)}{2}$ reducers to assign each pair of big inputs.
3. Use $x \times y$ reducers to assign each big input with each medium input.
4. Use the *AU method*, Algorithm 7A, 8, or 9 on the z small inputs, depending on the case, to assign each pair of small inputs.

We present an example to illustrate the hybrid algorithm in Figure 5.10. Note that the use of $\frac{x(x-1)}{2}$ reducers assigns each pair of original inputs whose size between $\frac{q}{3}$ and $\frac{q}{2}$. Also by using $x \times y$ reducers, we assign each big input (or original inputs whose size is between $\frac{q}{3}$ and $\frac{q}{2}$) with each original input whose size is less than $\frac{q}{3}$. Further, the *AU method*, Algorithm 7A, 8, or Algorithm 9 assigns each pair of original inputs whose size is less than or equal to $\frac{q}{3}$.

Algorithm correctness. The algorithm correctness shows that every pair of inputs is assigned to reducers. Specifically, the algorithm correctness shows that each pair of the big inputs is assigned to reducers, each of the big inputs is assigned to reducers with each of the medium inputs, and each pair of the small inputs is assigned to reducers.

5.6 Approximation Algorithms for the *A2A Mapping Schema Problem* with an Input $> q/2$

In this section, we consider the case of an input of size w_i, $\frac{q}{2} < w_i < q$; we call such an input as a *big input*. Note that if there are two big inputs, then they cannot be assigned to a single reducer, and hence, there is no solution to the *A2A mapping schema problem*. We assume m inputs of different sizes are given. There is a big input and all the remaining $m-1$ inputs, which we call the *small inputs*, have at most size $q - w_i$. We consider the following three cases in this section:

1. The big input has size w_i, where $\frac{q}{2} < w_i \leq \frac{2q}{3}$,
2. The big input has size w_i, where $\frac{2q}{3} < w_i \leq \frac{3q}{4}$,
3. The big input has size w_i, where $\frac{3q}{4} < w_i < q$.

The communication cost is dominated by the big input. We consider three different cases of the big input to provide efficient algorithms in terms of the communication cost, where the first two cases can assign inputs to almost an optimal number of reducers, which results in almost minimum communication cost. We use the previously given algorithms to provide a solution to the *A2A mapping schema problem* for the case of a big input.

A *simple solution* is to use FFD or BFD bin-packing algorithm to place the small inputs to bins of size $q-w_i$. Now, we consider each of the bins as a single input of size $q-w_i$. Let x bins are used. We assign each of the x bins to one reducer with a copy of the big input. Further, we assign the small inputs to bins of size $\frac{q}{2}$, and consider each of such bins as a single input of size $\frac{q}{2}$. Now, we can assign each pair of bins (each of size $\frac{q}{2}$) to reducers. In this manner, each pair of inputs is assigned to reducers.

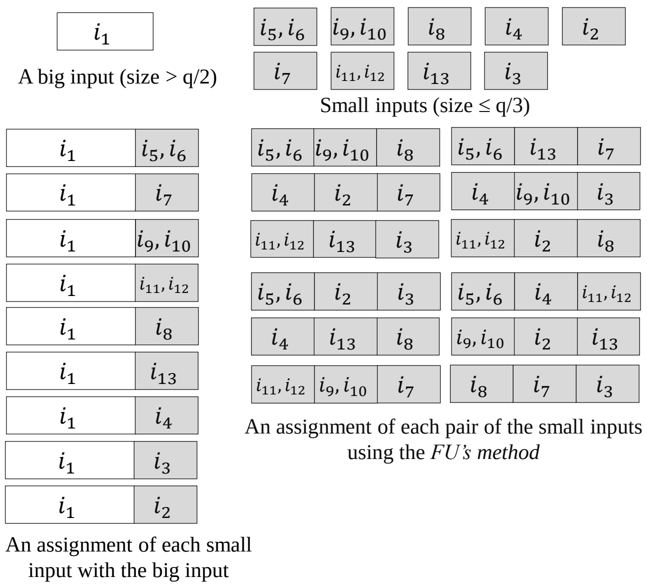
The big input of size $\frac{q}{2} < w_i \leq \frac{2q}{3}$. In this case, we assume that the small inputs have at most $\frac{q}{3}$ size. We use FFD or BFD bin-packing algorithm, the *AU method* (Section 5.2.3), and Algorithms 8, 9 (Section 5.4). We proceed as follows:

1. First assign the big input with the small inputs.
   (a) Use a bin-packing algorithm to place the small inputs to bins of size $\frac{q}{3}$. Now, we consider each of the bins as a single input of size $\frac{q}{3}$.
   (b) Consider that x bins are used. Assign each of the bins to one reducer with a copy of the big input.
2. Depending on the number of bins, we use the *AU method*, and Algorithms 8, 9 to assign each pair of the small inputs to reducers.

An example is given in Figure 5.11, where we place the small inputs to 9 bins of size $\frac{q}{3}$ and assign each of the bins to one reducer with a copy of the big input. Further, we implement the *AU method* on 9 bins to assign each pair of the small inputs.

The big input of size $\frac{2q}{3} < w_i \leq \frac{3q}{4}$. In this case, we assume that the small inputs have at most $\frac{q}{4}$ size. We use a bin-packing algorithm and Algorithms 7B (Sections 5.3). We proceed as follows:

1. First assign the big input with the small inputs.
   (a) Use a bin-packing algorithm to place the small inputs to bins of size $\frac{q}{4}$.
   (b) Consider that x bins are used. Assign each of the bins to one reducer with a copy of



i_1
A big input (size > q/2)
i_5, i_6
i_9, i_{10}
i_8
i_4
i_2
i_7
i_{11}, i_{12}
i_{13}
i_3
Small inputs (size ≤ q/3)
An assignment of each small input with the big input
An assignment of each pair of the small inputs using the *FU's method*

Figure 5.11: An example to show an assignment of a big input of size $\frac{q}{2} < w_i \leq \frac{2q}{3}$ with all the remaining inputs of sizes less than or equal to $\frac{q}{3}$.

the big input.

2. Depending on the number of bins, we use Algorithm 7B to assign each pair of small inputs.

The big input of size $\frac{3q}{4} < w_i < q$. In this case, we assume that the small inputs have at most $q - w_i$ size. In this case, we use a bin-packing algorithm and place the small inputs to bins of size $q - w_i$. We then place each of the bins to one reducer with a copy of the big input. Note that, we have not assigned each pair of small inputs. In order to assign each pair of small inputs, we use the bin-packing-based approximation algorithm (Section 5.1.1) or Algorithms 7A- 9 depending on size of the small inputs.

Theorem 5.19 (Upper bounds from algorithms) *For a list of m inputs where a big input, i, of size $\frac{q}{2} < w_i < q$ and for a given reducer capacity q, $q < s' < s$, an input is replicated to at most $m-1$ reducers for the A2A mapping schema problem, and the number of reducers and the communication cost are at most $m-1+\frac{8s^2}{q^2}$ and $(m-1)q+\frac{4s^2}{q}$, respectively, where s' is the sum of all the input sizes except the size of the big input and s is the sum of all the input sizes.*

The proof of the theorem is given in Appendix C.8.

Approximation factor. The optimal communication cost (from Theorem 5.1) is s^2/q and the communication cost of the algorithm (from Theorem 5.19) is $(m-1)q + 4s^2/q$. Thus, the ratio between the optimal communication and the communication of our mapping schema is approximately $\frac{s^2}{mq^2}$.

5.7 An Approximation Algorithm for the *X2Y Mapping Schema Problem*

We propose an approximation algorithm for the *X2Y mapping schema problem* that is based on bin-packing algorithms. Two lists, X of m inputs and Y of n inputs, are given. We assume that the sum of input sizes of the lists X, denoted by sum_x, and Y, denoted by sum_y, is greater than q. We analyze the algorithm on criteria (number of reducers and the communication cost) given in Section 5.1. We look at the lower bounds in Theorem 5.20, and Theorem 5.21 gives an upper bound from the algorithm. The bounds are given in Table 5.1.

Theorem 5.20 (Lower bounds on the communication cost and number of reducers) *For a list X of m inputs, a list Y of n inputs, and a given reducer capacity q, the communication cost and the number of reducers, for the X2Y mapping schema problem, are at least $\frac{2\cdot sum_x\cdot sum_y}{q}$ and $\frac{2\cdot sum_x\cdot sum_y}{q^2}$, respectively.*

The proof of the theorem is given in Appendix C.9.

Bin-packing-based approximation algorithm for the *X2Y mapping schema problem.* A solution to the *X2Y mapping schema problem* for different-sized inputs can be achieved using bin-packing algorithms. Let two lists X of m inputs and Y of n inputs are given. The algorithm will not work when a list holds an input of size w_i and the another list holds an input of size greater than $q-w_i$, because these inputs cannot be assigned to a single reducer in common. Let the size of the largest input, i, of the list X is w_i; hence, all the inputs of the list Y have at most size $q-w_i$. We place inputs of the list X to bins of size w_i, and let x bins are used to place m inputs. Also, we place inputs of the list Y to bins of size $q-w_i$, and let y bins are used to place n inputs. Now, we consider each of the bins as a single input, and a solution to the *X2Y mapping schema problem* is obtained by assigning each of the x bins with each of the y bins to reducers. In this manner, we require $x\cdot y$ reducers.

Theorem 5.21 (Upper bounds from the algorithm) *For a bin size b, a given reducer capacity $q=2b$, and with each input of lists X and Y being of size at most b, the number of reducers and the communication cost, for the X2Y mapping schema problem, are at most $\frac{4\cdot sum_x\cdot sum_y}{b^2}$, and at most $\frac{4\cdot sum_x\cdot sum_y}{b}$, respectively, where sum_x is the sum of input sizes of the list X, and sum_y is the sum of input sizes of the list Y.*

The proof of the theorem is given in Appendix C.9.

Approximation factor. The optimal communication is $\frac{2\cdot sum_x\cdot sum_y}{q}$. Thus, the ratio between the optimal communication and the communication of our mapping schema is $\frac{1}{4}$.

Chapter 6

Meta-MapReduce

In the previous chapter, we investigated impacts on the communication cost when the locations of input data and MapReduce computations are identical. However, ensuring an identical location of data and mappers-reducers cannot always be guaranteed. It may be possible that a user has a single local machine and wants to enlist a public cloud to help data processing. Consequently, in both the cases, it is required to move data to the location of mappers-reducers. Interested readers may refer to examples of MapReduce computations where the locations of data and mappers-reducers are different in our review paper [44].

In order to motivate and demonstrate the impact of different locations of data and mappers-reducers, we consider a real example of Amazon Elastic MapReduce[1]. Amazon Elastic MapReduce (EMR) processes data that is stored in Amazon Simple Storage Service (S3)[2], where the locations of EMR and S3 are not identical. Hence, it is required to move data from S3 to the location of EMR. However, moving the whole dataset from S3 to EMR is not efficient if only small specific part of it is needed for the final output.

In this chapter, we are interested in minimizing the data transferred in order to avoid communication and memory overhead, as well as to protect data privacy as much as possible. In MapReduce, we transfer inputs to the site of mappers-reducers from the site of the user, and then, several copies of inputs from the map phase to the reduce phase in each iteration, regardless of their involvement in the final output. If few inputs are required to compute the final output, then it is not communication efficient to move all the inputs to the site of mappers-reducers, and then, the copies of same inputs to the reduce phase. There are some works that consider the location of data [94, 96] in a restrictive manner and some works [15, 41, 86] that consider data movement from the map phase to the reduce phase.

Motivating Example: Equijoin of two relations $X(A, B)$ and $Y(B, C)$. *Problem statement*: The join of relations $X(A, B)$ and $Y(B, C)$, where the joining attribute is B,

[1]http://aws.amazon.com/elasticmapreduce/
[2]http://aws.amazon.com/s3/

provides output tuples $\langle a, b, c\rangle$, where (a, b) is in A and (b, c) is in C. In the equijoin of $X(A, B)$ and $Y(B, C)$, all tuples of both the relations with an identical value of the attribute B should appear together at the same reducer for providing the final output tuples.

Communication cost analysis: We now investigate the impact of different locations of the relations and mappers-reducers on the communication cost. In Figure 6.1, the communication cost for joining of the relations X and Y — where X and Y are located at two different clouds and equijoin is performed on a third cloud — is the sum of the sizes of all three tuples of each relation that are required to move from the location of the user to the location of mappers, and then, from the map phase to the reduce phase. Consider that each tuple is of unit size, and hence, the total communication cost is 12 for obtaining the final output.

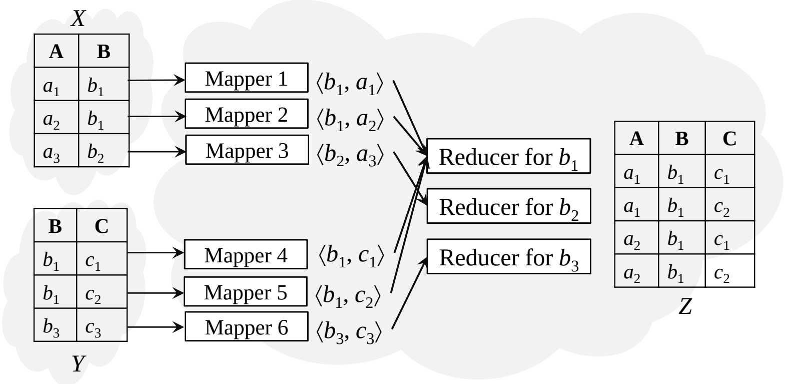


Figure 6.1: Equijoin of relations $X(A, B)$ and $Y(B, C)$.

However, if there are a few tuples having an identical B-value in both the relations, then it is useless to move the whole relations from the user's location to the location of mappers, and then, tuples from the map phase to the reduce phase. In Figure 6.1, two tuples of X and two tuples of Y have a common B value (*i.e.*, b_1). Hence, it is not efficient to send tuples having values b_2 and b_3, and by not sending tuples with B values b_2 and b_3, we can reduce the communication cost.

6.1 The System Setting

The system setting is an extension of the settings presented in Section 4.2, where we consider for the first time the locations of data and mappers-reducers. The setting is suitable for a variety of problems where *at least* two inputs are required to produce an output. In order to produce an output, we use the definition of the mapping schema, given in Section 4.2.

The Model. The model is simple but powerful and assumes the following:

1. Existence of systems such as Spark, Pregel, or modern Hadoop.
2. A preliminary step at the user site who owns the dataset for finding metadata[3] that has smaller memory size than the original data.
3. Approximation algorithms (given in Chapter 5), at the cloud or the global reducer in case of Hierarchical MapReduce [82]. The approximation algorithms assign outputs of the map phase to reducers and regards the reducer capacity. Particularly, in our case, approximation algorithms will assign metadata to reducers in such a manner that the size of actual data at a reducer will not exceed than the reducer capacity and all the inputs that are required to produce outputs must be assign at one reducer in common.

It should be noted that we are enhancing MapReduce and not creating entirely a new framework for large-scale data processing; thus, Meta-MapReduce is *implementable* in the state-of-the art MapReduce systems such as Spark, Pregel, or modern Hadoop.

6.2 Meta-MapReduce: Description

The idea behind the proposed technique is to process metadata at mappers and reducers, and process the original required data at required iterations of a MapReduce job at reducers. In this manner, we suggest to process metadata at mappers and reducers at all the iterations of a MapReduce job. Therefore, the proposed technique is called *Meta-MapReduce*[4]

We need to redefine the communication cost, which was defined in Section 4.1, to take into account the size of the metadata, the total amount of the (required) original data, and different locations of data and computations.

The communication cost for metadata and data. In the context of Meta-MapReduce, the communication cost is the sum of the following:

Metadata cost The total amount of metadata that is required to move from the location of users to the location of mappers (if the locations of data and mappers are different) and from the map phase to the reduce phase in each iteration of MapReduce job.

Data cost The total amount of required original data that is needed to move to reducers at required iterations of a MapReduce job.

In Meta-MapReduce, users send metadata to the site of mappers, instead of original data, see Figure 6.2. Now, mappers and reducers work on metadata, and at required iterations of a MapReduce job, reducers *call* required original data from the site of users (according to assigned $\langle key, value \rangle$ pairs) and provide the desired result. The detailed

[3]The term metadata is used in a different manner, and it represents a small subset, which varies according to tasks, of the dataset. The selection of metadata depends on the problem.

[4]The proposed algorithmic technique is similar to Bloomjoin [83, 25]. However, due to distributed nature of data in a MapReduce job, it is not trivial to apply Bloom filters in MapReduce [75].

execution of Meta-MapReduce is given below:

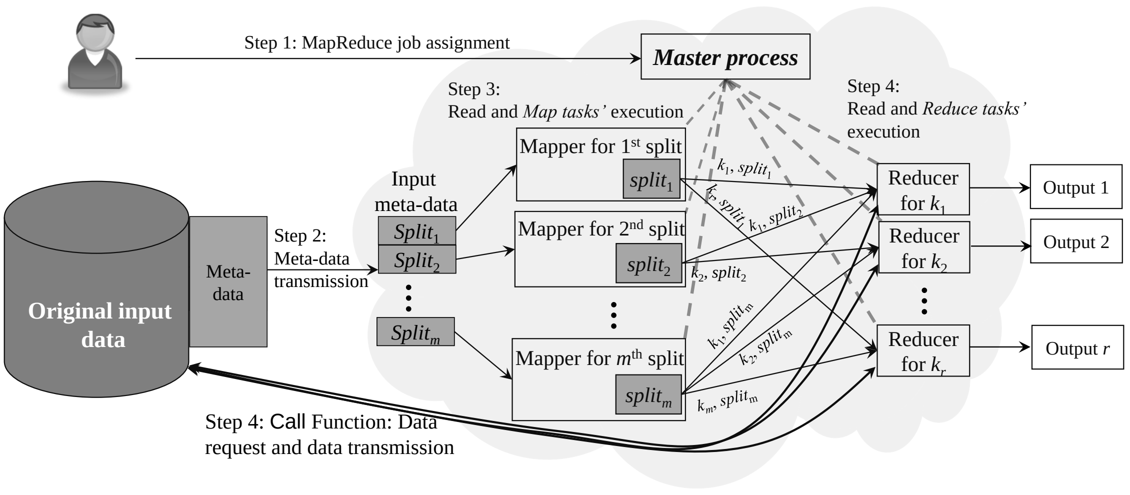


Figure 6.2: Meta-MapReduce algorithmic approach.

STEP 1 Users create a *master process* that creates *map tasks* and *reduce tasks* at different compute nodes. A compute node that processes the *map task* is called a *map worker*, and a compute node that processes the *reducer task* is called a *reduce worker*.

STEP 2 Users send metadata, which varies according to an assigned MapReduce job, to the site of mappers. Also, the user creates an index, which varies according to the assigned job, on the entire database.

For example, in the case of equijoin (see Figure 6.1), a user sends metadata for each of the tuples of the relations $X(A, B)$ and $Y(B, C)$ to the site of mappers. In this example, metadata for a tuple i ($\langle a_i, b_i\rangle$, where a_i and b_i are values of the attributes A and B, respectively) of the relation X includes the size of all non-joining values (*i.e.*, $|a_i|$[5]) and the value of b_i. Similarly, metadata for a tuple i ($\langle b_i, c_i\rangle$, where b_i and c_i are values of the attributes B and C, respectively) of the relation Y includes the size of all non-joining values (*i.e.*, $|c_i|$) with b_i (remember that the size of b_i is much smaller than the size of a_i or c_i). In addition, the user creates an index on the attribute B of both the relations X and Y.

STEP 3 In the map phase, a mapper processes an assigned input and provides some number of $\langle key, value\rangle$ pairs, which are known as *intermediate outputs*, a $value$ is the total *size* of the corresponding input data (which is included in metadata). The *master process* is then informed of the location of intermediate outputs.

For example, in case of equijoin, a mapper takes a single tuple i (*e.g.*, $\langle |a_i|, b_i\rangle$) and generates some $\langle b_i, value\rangle$ pairs, where b_i is a key and a $value$ is the *size* of tuple i (*i.e.*, $|a_i|$). Note that in the original equijoin example, a $value$ is the whole data associated

[5]The notation $|a_i|$ refers to the size of an input a_i.

with the tuple i (*i.e.*, a_i).

STEP 4 The *master process* assigns *reduce tasks* (by following a mapping schema) and provides information of intermediate outputs, which serve as inputs to *reduce tasks*. A reducer is then assigned all the $\langle key, value\rangle$ pairs having an identical key by following a mapping schema for an assigned job. Now, reducers perform the computation and `call`[6] only required data if there is only one iteration of a MapReduce job. On the other hand, if a MapReduce job involves more than one iteration, then reducers `call` original required data at required iterations of the job (we will discuss multi-rounds MapReduce jobs using Meta-MapReduce in Section 6.3.3).

For example, in case of equijoin, a reducer receives all the $\langle b_i, value\rangle$ pairs from both the relations X and Y, where a $value$ is the size of tuple associated with key b_i. Inputs (*i.e.*, intermediate outputs of the map phase) are assigned to reducers by following a mapping schema for equijoin such that a reducer does not assign more original inputs than its capacity, and after that reducers invoke the `call` operation. Note that a reducer that receives at least one tuple with key b_i from both the relations X and Y produces outputs and requires original input data from the user's site. However, if the reducer receives tuples with key b_i from a single relation only, the reducer does not request for the original input tuple, since these tuples do not participate in the final output.

Following Meta-MapReduce, we now compute the communication cost involved in equijoin example (see Figure 6.1). Recall that without using Meta-MapReduce, a solution to equijoin problem (in Figure 6.1) requires 12 units communication cost. However, using Meta-MapReduce for performing equijoin, there is no need to send the tuple $\langle a_3, b_2\rangle$ of the relation X and the tuples $\langle b_3, c_3\rangle$ of the relation Y to the location of computation. Moreover, we send metadata of all the tuples to the site of mappers, and intermediate outputs containing metadata are transferred to the reduce phase, where reducers `call` only desired tuples having b_1 value from the user's site. Consequently, a solution to the problem of equijoin has only 4 units cost plus a constant cost for moving metadata using Meta-MapReduce, instead of 12 units communication cost.

Theorem 6.1 (The communication cost for join of two relations) *Using Meta-MapReduce, the communication cost for the problem of join of two relations is at most $2nc + h(c + w)$ bits, where n is the number of tuples in each relation, c is the maximum size of a value of the joining attribute, h is the number of tuples that actually join, and w is the maximum required memory for a tuple.*

The proof of the theorem is given in Appendix D.

[6]The `call` operation will be explained in Section 6.2.1.

Problems	Section	Theorem	Communication cost	
			using Meta-MapReduce	using MapReduce
Join of two relations	6.2	6.1	$2nc + h(c + w)$	$4nw$
Skewed Values of the Joining Attribute	6.2.2	6.2	$2nc + rh(c + w)$	$2nw(1 + r)$
Join of two relations by hashing the joining attribute	6.3.2	6.3	$6n \cdot log\ m + h(c + w)$	$4nw$
Join of k relations by hashing the joining attributes	6.3.3	6.4	$3knp \cdot log\ m + h(c + w)$	$2knw$

n: the number of tuples in each relation, c: the maximum size of a value of the joining attribute, r: the replication rate, h: the number of tuples that actually join, w is the maximum required memory for a tuple, p: the maximum number of dominating attributes in a relation, and m: the maximal number of tuples in all given relations.

Table 6.1: The communication cost for joining of relations using Meta-MapReduce.

6.2.1 The `Call` function

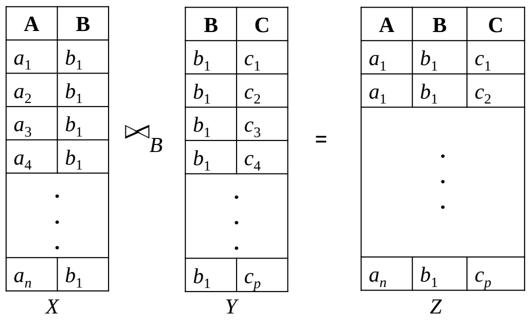
In this section, we will describe the `call` function that is invoked by reducers to have the original required inputs from the user's site to produce outputs.

All the reducers that produce outputs require the original inputs from the site of users. Reducers can know whether they produce outputs or not, after receiving intermediate outputs from the map phase, and then, inform the corresponding mappers from where they have fetched these intermediate outputs (for simplicity, we can say all reducers that will produce outputs send 1 to all the corresponding mappers to request the original inputs, otherwise send 0). Mappers collect requests for the original inputs from all the reducers and fetch the original inputs, if required, from the user's site by accessing the index file. Remember that in Meta-MapReduce, the user creates an index on the entire database according to an assigned job, refer to STEP 2 in Section 6.2. This index helps to access required data that reducers want without doing a scan operation. Note that the `call` function can be easily implemented on recent implementations of MapReduce, *e.g.*, Pregel and Spark.

For example, we can consider our running example of equijoin. In case of equijoin, a reducer that receives at least one tuple with key b_i from both the relations $X(A, B)$ and $Y(B, C)$ requires the original input from the user's site, and hence, the reducer sends 1 to the corresponding mappers. However, if the reducer receives tuples with key b_i from a single relation only, the reducer sends 0. Consider that the reducer receives $\langle b_i, |a_i|\rangle$ of the relation X and $\langle b_i, |c_i|\rangle$ of the relation Y. The reducer sends 1 to corresponding mappers that produced $\langle b_i, |a_i|\rangle$ and $\langle b_i, |c_i|\rangle$ pairs. On receiving requests for the original inputs from the reducer, the mappers access the index file to fetch a_i, b_i, and c_i, and then, the mapper provides a_i, b_i, and c_i to the reducer.

6.2.2 Meta-MapReduce for skewed values of the joining attribute

Consider two relations $X(A, B)$ and $Y(B, C)$, where the joining attribute is B and the size of all the B values is very small as compared to the size of values of the attributes A and C. One or both of the relations X and Y may have a large number of tuples with an identical B-value. A value of the joining attribute B that occurs many times is known as a *heavy hitter*. In skew join of $X(A, B)$ and $Y(B, C)$, all the tuples of both the relations with an identical heavy hitter should appear together to provide the output tuples.



A	B
a_1	b_1
a_2	b_1
a_3	b_1
a_4	b_1
⋮	
a_n	b_1

X $\bowtie_B$

B	C
b_1	c_1
b_1	c_2
b_1	c_3
b_1	c_4
⋮	
b_1	c_p

Y =

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
	⋮	
a_n	b_1	c_p

Z

Figure 6.3: Skew join example for a heavy hitter, b_1.

In Figure 6.3, b_1 is a heavy hitter; hence, it is required that all the tuples of $X(A, B)$ and $Y(B, C)$ with the heavy hitter, b_1, should appear together to provide the output tuples, $\langle a, b_1, c\rangle$ ($a \in A, b_1 \in B, c \in C$), which depend on exactly two inputs. However, due to a single reducer — for joining all tuples with a heavy hitter — there is no parallelism at the reduce phase, and a single reducer takes a long time to produce all the output tuples of the heavy hitter.

We can restrict reducers in a way that they can hold many tuples, but not all the tuples with the heavy-hitter-value. In this case, we can reduce the time and use more reducers, which results in a higher level of parallelism at the reduce phase. But, there is a higher communication cost, since each tuple with the heavy hitter must be sent to more than one reducer. We can solve the problem of skew join using Meta-MapReduce.

Theorem 6.2 (The communication cost for skew join) *Using Meta-MapReduce, the communication cost for the problem of skew join of two relations is at most* $2nc+rh(c+w)$ *bits, where* n *is the number of tuples in each relation,* c *is the maximum size of a value of the joining attribute,* r *is the replication rate,* h *is the number of distinct tuples that actually join, and* w *is the maximum required memory for a tuple.*

The proof of the theorem is given in Appendix D.

6.3 Extensions of Meta-MapReduce

We have presented Meta-MapReduce framework for different locations of data and mappers-reducers. However, some extensions are required to use Meta-MapReduce for geo-distributed data processing, for handling large size of values of joining attributes, and for handling multi-rounds computations. In this section, we will provide three extensions of Meta-MapReduce.

6.3.1 Incorporating Meta-MapReduce in geo-distributed MapReduce

There are some extensions of the standard MapReduce for processing geo-distributed data at their locations. Details about geo-distributed MapReduce-based data processing frameworks are given in [44]. G-MR [69], G-Hadoop [108], and Hierarchical MapReduce (HMR) [82] are three implementations for geo-distributed data processing using MapReduce. These implementations assume that a cluster processes data using MapReduce and provides its outputs to one of the clusters that provides final outputs (by executing a MapReduce job on the received outputs of all the clusters). However, the transmission of outputs of all the clusters to a single cluster for producing the final output is not efficient, if all the outputs of a cluster do not participate in the final output.

We can apply Meta-MapReduce idea to systems such as G-MR, G-Hadoop, and HMR. Note that we *do not* change basic functionality of these implementations. We take our running example of equijoin (see Figure 6.4, where we have three clusters, possibly on three continents, the first cluster has two relations $U(A, B)$ and $V(B, C)$, the second cluster has two relations $W(D, B)$ and $X(B, E)$, and the third cluster has two relations $Y(F, B)$ and $Z(B, G)$) and assume that data exist at the site of mappers in each cluster. In the final output, reducers perform the join operation over all the six relations, which share an identical B-value.

The following three steps are required for obtaining final outputs using an execution of Meta-MapReduce over G-MR, G-Hadoop, and HMR.

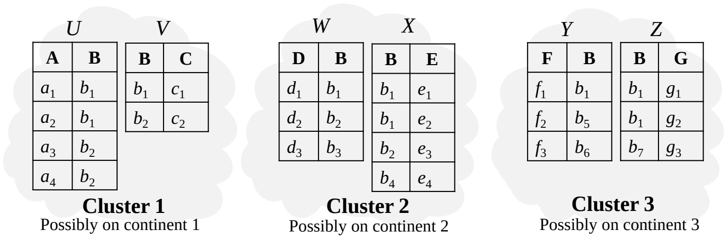
STEP 1 Mappers at each cluster process input data according to an assigned job and provide $\langle key, value \rangle$ pairs, where a $value$ is the size of an assigned input.

For example, in Figure 6.4, a mapper at Cluster 1 provides outputs of the form of $\langle b_i, |a_i| \rangle$ or $\langle b_i, |c_i| \rangle$.

STEP 2 Reducers at each cluster provide partial outputs by following an assigned mapping schema, and partial outputs, which contain only metadata, are transferred to one of the clusters, which will provide final outputs.

For example, in case of equijoin, reducers at each cluster provide partial output tuples as $\langle |a_i|, b_i, |c_i|\rangle$ at Cluster 1, $\langle |d_i|, b_i, |e_i|\rangle$ at Cluster 2, and $\langle |f_i|, b_i, |g_i|\rangle$ at Cluster 3 (by following a mapping schema for equijoin). Partial outputs of Cluster 1 and Cluster 3 have to be transferred to one of the clusters, say Cluster 2, for obtaining the final output.

STEP 3 A designated cluster for providing the final output processes all the outputs of the clusters by implementing the assigned job using Meta-MapReduce. Reducers that provide the final output `call` the original input data from all the clusters.
For example, in equijoin, after receiving outputs of Cluster 1 and Cluster 3, Cluster 2 implements two iterations for joining tuples. In the first iteration, outputs of Clusters 1 and 2 are joined (by following a mapping schema for equijoin), and in the second iteration, outputs of Clusters 3 and the output of the previous iteration are joined at reducers. A reducer in the second iteration provides the final output as $\langle |a_i|, b_1, |c_i|, |d_i|, |e_i|, |f_i|, |g_i|\rangle$ and `call`s all the original values of $|a_i|$, $|c_i|$, $|d_i|$, $|e_i|$, $|f_i|$, and $|g_i|$ for providing the desired output, as suggested in Section 6.2.1.



A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_2

B	C
b_1	c_1
b_2	c_2

D	B
d_1	b_1
d_2	b_2
d_3	b_3

B	E
b_1	e_1
b_1	e_2
b_2	e_3
b_4	e_4

F	B
f_1	b_1
f_2	b_5
f_3	b_6

B	G
b_1	g_1
b_1	g_2
b_7	g_3

Figure 6.4: Three clusters, each with two relations.

Communication cost analysis. In Figure 6.4, we are performing equijoin in three clusters, and assuming that data is available at the site of mappers in each cluster. In addition, we consider that each value takes two units size; hence, any tuple, for example, $\langle a_i, b_i\rangle$, has size 4 units.

First, each of the clusters performs an equijoin within the cluster using Meta-MapReduce. Note that using Meta-MapReduce, there is no need to send any tuple from the map phase to the reduce phase within the cluster, while G-MR, G-Hadoop, and HMR do data transfer from the map phase to the reduce phase, and hence, results in 76 units of communication cost. Moreover, in GMR, G-Hadoop, and HMR, the transmission of two tuples ($\langle a_3, b_2\rangle$, $\langle a_4, b_2\rangle$) of U, one tuple ($\langle b_2, c_2\rangle$) of V, two tuples ($\langle d_2, b_2\rangle$, $\langle d_3, b_3\rangle$) of W, two tuples ($\langle b_2, e_3\rangle$, $\langle b_4, e_4\rangle$) of X, two tuples ($\langle f_2, b_5\rangle$, $\langle f_3, b_6\rangle$) of Y, and one tuple ($\langle b_7, g_3\rangle$) of Z from the map phase to the reduce phase is useless, since they do not participate in the final output.

After computing outputs within the cluster, metadata of outputs (*i.e.*, size of tuples associated with key b_1 and key b_2) is transmitted to Cluster 2. Here, it is important to note that tuples with value b_1 provide final outputs. Using Meta-MapReduce, we will not send the complete tuples with value b_2, hence, we also decrease the communication cost; while G-MR, G-Hadoop, and HMR send all the outputs of the first and third clusters to the second cluster. After receiving outputs from the first and the third clusters, the second cluster performs two iterations as mentioned previously, and in the second iteration, a reducer for key b_1 provides the final output. Following that the communication cost is only 36 units.

On the other hand, transmission of outputs with data from the first cluster and the third cluster to the second cluster and performing two iterations result in 132 units communication cost. Therefore, G-MR, G-Hadoop, and HMR require 208 units communication cost, while Meta-MapReduce provides the final results using 36 units communication cost.

6.3.2 Large size of joining values

We have considered that sizes of joining values are very small as compared to sizes of all the other non-joining values. For example, in Figure 6.1, sizes of all the values of the attribute B are very small as compared to all the values of the attributes A and C. However, considering very small size of values of the joining attribute is not realistic. All the values of the joining attribute may also require a considerable amount of memory, which may be equal or greater than the sizes of non-joining values. In this case, it is not useful to send all the values of the joining attribute with metadata of non-joining attributes. Thus, we enhance Meta-MapReduce for handling a case of large size of joining values.

We consider our running example of join of two relations $X(A, B)$ and $Y(B, C)$, where the size of each of b_i is large enough such that the value of b_i cannot be used as metadata. We use hash function to gain a short identifier (that is unique with high probability) for each b_i. We denote $H(b_i)$ to be the hash value of the original value of b_i. Here, Meta-MapReduce works as follows:

STEP 1 For all the values of the joining attribute (B), use a hash function such that an identical b_i in both of the relations has a unique hash value with a high probability, and b_i and b_j, $i \neq j$, receive two different hash values with a high probability.

STEP 2 For all the other non-joining attributes' values (values corresponding to the attributes A and C), find metadata that includes size of each of the values.

STEP 3 Perform the task using Meta-MapReduce, as follows: (*i*) Users send hash values of joining attributes and metadata of the non-joining attributes. For example, a user sends hash value of b_i ($H(b_i)$) and the corresponding metadata (*i.e.*, size) of values a_i or c_i to the site of mappers. (*ii*) A mapper processes an assigned tuples and provides intermediate outputs, where a key is $H(b_i)$ and a $value$ is $|a_i|$ or $|c_i|$. (*iii*) Reducers `call` all the *values* corresponding to a key (hash value), and if a reducer receives metadata of a_i and c_i, then the reducer calls the original input data and provides the final output.

Note that there may be a possibility that two different values of the joining attribute have an identical hash value; hence, these two values are assigned to a reducer. However, the reducer will know these two different values, when it will `call` the corresponding data. The reducer notifies the master process, and a new hash function is used.

Theorem 6.3 (The communication cost when joining attributes are large) *Using Meta-MapReduce for the problem of join where values of joining attributes are large, the communication cost for the problem of join of two relations is at most $6n \cdot log\ m + h(c+w)$ bits, where n is the number of tuples in each relation, m is the maximal number of tuples in two relations, h is the number of tuples that actually join, and w is the maximum required memory for a tuple.*

The proof of the theorem is given in Appendix D.

6.3.3 Multi-rounds computations

We show how Meta-MapReduce can be incorporated in a multi-rounds MapReduce job, where values of joining attributes are also large as the value of non-joining attributes. In order to explain, the working of Meta-MapReduce in a multi-iterative MapReduce job, we consider an example of join of four relations $U(A, B, C, D)$, $V(A, B, D, E)$, $W(D, E, F)$, and $X(F, G, H)$, and perform the join operation using a cascade of two-way joins..

STEP 1 Find *dominating attributes* in all the relations. An attribute that occurs in more than one relation is called a dominating attribute [17].
For example, in our running example, attributes A, B, D, E, and F are dominating attributes.

STEP 2 Implement a hash function over all the values of dominating attributes so that all the identical values of dominating attributes receive an identical hash value with a high probability, and all the different values of dominating attributes receive different hash values with a high probability.
For example, identical values of a_i, b_i, d_i, e_i, and f_i receive an identical hash value, and any two values a_i and a_j, such that $i \neq j$, probably receive different hash values (a similar case exists for different values of attributes B, D, E, F).

STEP 3 For all the other non-dominating joining attributes' (an attribute that occurs in only one of the relations) values, we find metadata that includes size of each of the values.

STEP 4 Now perform 2-way cascade join using Meta-MapReduce and follow a mapping schema according to a problem for assigning inputs (*i.e.*, outputs of the map phase) to reducers.

For example, in equijoin example, we may join relations as follows: first, join relations U and V, and then join the relation W to the outputs of the join of relations U and V. Finally, we join the relation X to outputs of the join of relations U, V, and W. Thus, we join the four relations using three iterations of Meta-MapReduce, and in the final iteration, reducers `call` the original required data.

Example: Following our running example, in the first iteration, a mapper produces $\langle H(a_i), [H(b_i), |c_i|, H(d_i)]\rangle$ after processing a tuple of the relation U and $\langle H(a_i), [H(b_i), H(d_i), H(e_i)]\rangle$ after processing a tuple of the relation V (where $H(a_i)$ is a key). A reducer corresponding to $H(a_i)$ provides $\langle H(a_i), H(b_j), |c_k|, H(d_l), H(e_z)\rangle$ as outputs.

In the second iteration, a mapper produces $\langle H(d_i), [H(a_i), H(b_j), |c_k|, H(e_z)]\rangle$ and $\langle H(d_i), [H(e_i), H(f_i)]\rangle$ after processing outputs of the first iteration and the relation W, respectively. Reducers in the second iteration provide output tuples by joining tuples that have an identical $H(d_i)$. In the third iterations, a mapper produces $\langle H(f_i), [H(a_i), H(b_i), |c_i|, H(d_i), H(e_i)]\rangle$ or $\langle H(f_i), [|g_i|, |h_i|]\rangle$, and reducers perform the final join operations. A reducer, for key $H(f_i)$, receives $|g_i|$ and $|h_i|$ from the relation X and output tuples of the second iteration, provides the final output by calling original input data from the location of user.

Theorem 6.4 (The communication cost for k relations and when joining attributes are large) *Using Meta-MapReduce for the problem of join where values of joining attributes are large, the communication cost for the problem of join of k relations, each of the relations with n tuples, is at most $3knp \cdot log\ m + h(c + w)$ bits, where n is the number of tuples in each relation, m is the maximal number of tuples in k relations, p is the maximum number of dominating attributes in a relation, h is the number of tuples that actually join, and w is the maximum required memory for a tuple.*

The proof of the theorem is given in Appendix D.

Chapter 7

Interval Join

In the previous Chapters 4, 5, and 6, we focused on inputs having different sizes, and accordingly, put a constraint in terms of the reducer capacity. Now, we will focus on identical-sized inputs. Thereby, all the reducers may hold an identical number of inputs. A model to investigate the communication cost in the case of identical-sized inputs is proposed by Afrati et al. [15]. In this model, the term *reducer size* is defined as the maximum number of inputs that can be assigned to a reducer.

In this chapter, we will use the model proposed by Afrati et al. [15] and focus on the problem of interval join of overlapping intervals that is a type of the *X2Y mapping schema problem*. We will study three cases, where we consider three types of intervals: (*i*) unit-length and equally spaced, (*ii*) variable-length and equally spaced, and (*iii*) equally-spaced with specific distribution of the various lengths.

MapReduce-based 2-way and multiway interval join algorithms of overlapping intervals *without regarding the reducer size* are presented in [31]. However, the analysis of a lower bound on replication of individual intervals is not presented; neither is an analysis of the replication rate of the algorithms offered therein.

7.1 Preliminaries

7.1.1 An example

Employees involved in the phases of a project. We show an example to illustrate temporal relations (a relation that stores data involving timestamps), intervals, and the need for interval join of overlapping intervals. Consider two (temporal) relations (*i*) $Project(Phase, Duration)$ that includes several phases of a project with their durations, and (*ii*) $Employee(EmpId, Name, Duration)$ that shows data of employees according to their involvement in the project's phases and their durations; see Figure 7.1. Here, the duration of a phase or the duration of an employee's involvement in a phase is given by an interval.

Phase	*Duration*
Requirement Analysis (RA)	1-Mar – 1-May
Design (D)	1-Apr – 1-June
Coding (C)	1-May – 1-Aug

Project

EmpId	*Name*	*Duration*
e_1	U	1-Apr – 1-June
e_2	V	1-June – 1-Aug
e_3	W	1-Apr – 1-July
e_4	X	1-Mar – 1-June
e_5	Y	1-Mar – 1-Aug

Employee

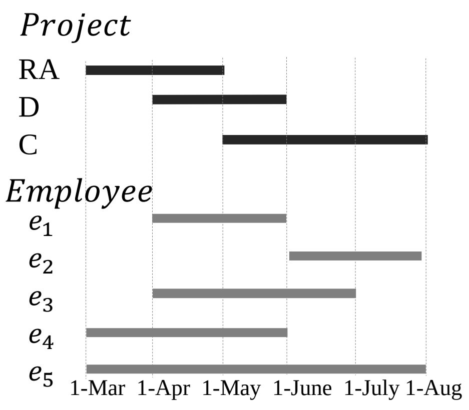


Figure 7.1: Two temporal relations ($Project(Phase, Duration)$ and $Employee(EmpId, Name, Duration)$) and their representation on a time diagram.

It is interesting to find all the employees that worked in more than one phase of the project. Formally, a query: find the name of all employees who worked in more than one phase of the project; requires us to join the relations to find all overlapping intervals of the relations. The answer to this query for our example will be a list of four employees such as employees U, W, X, and Y.

7.1.2 Formal problem statement

We consider the problem of interval join of overlapping intervals, where two relations X and Y are given. Each relation contains binary tuples that represent intervals, *i.e.*, each tuple corresponds to an interval and contains the starting-point and ending-point of this interval. Each pair of intervals $\langle x_i, y_j \rangle$, where $x_i \in X$ and $y_j \in Y$, $\forall i, j$, such that intervals x_i and y_j share at least one common time, corresponds to an output.

7.1.3 The setting

A (time) interval, i, is represented by a pair of times $[T_s^i, T_e^i]$, $T_s^i < T_e^i$, where T_s^i is the *starting-point* of the interval i and T_e^i is the *ending-point* of the interval i. The duration from T_s^i to T_e^i is the *length* of the interval i. Two intervals, say interval i with $[T_s^i, T_e^i]$ and interval j with $[T_s^j, T_e^j]$, may be related based on several operations between them, *e.g.*, `after` ($T_s^i < T_e^i < T_s^j < T_e^j$), `before` ($T_s^j < T_e^j < T_s^i < T_e^i$), and `overlap` ($T_s^i \leq T_s^j < T_e^i \leq T_e^j$, $T_s^j \leq T_s^i < T_e^j \leq T_e^i$, $T_s^i < T_e^i = T_s^j < T_e^j$, or $T_s^i = T_s^j < T_e^i = T_e^j$). Interested readers can refer to Figure 1 of [31].

We are interested in the overlap operation among intervals. Two intervals are called *overlapping intervals* if the intersection of the intervals is nonempty, one interval contains the other, or intervals are superimposed.

Mapping Schema. A MapReduce job can be described by a *mapping schema* [15]. A mapping schema assigns each input (*i.e.*, an interval in this problem) to a number of reducers (via the formation of key-value pairs) so that

1. No reducer is assigned more than q inputs (intervals in this case).
2. For each output (*i.e.*, pair of overlapping intervals, in this problem), there must exist a reducer that receives the corresponding pair of inputs (*i.e.*, overlapping intervals) that participate in the computation of this output,

Here, the point (*i*) puts a constraint on each reducer, defined as the *reducer size*. The point (*ii*) provides a mapping between inputs and outputs so that the algorithm produces the desired results. The *replication rate* of a mapping schema is the average number of key-value pairs for each interval and is a significant performance parameter in a MapReduce job. We analyze here lower and upper bounds on the replication rate for the problem of overlapping intervals.

Parameters for bound analysis. We now define two parameters to analyze the problem of interval join of overlapping intervals, as follows:

1. *Replication rate*. Intervals are needed to be replicated at different numbers of reducers. We therefore need to consider the number of reducers to which each individual input is sent, and the *replication rate* [15] is a parameter that finds the average replication of intervals. Formally, the replication rate is the average number of key-value pairs created for an interval.
2. *The communication cost*. The communication cost is the sum of all the bits that are required to transfer from the map phase to the reduce phase.

Table 7.1 summarizes all the results in this chapter.

Cases	Solutions	Theorems	Replication rate
The **lower bounds**			
Unit-length and equally spaced intervals		7.1	$\frac{2n}{qk}$
Variable-length and equally spaced intervals		7.6	$\frac{2l_{min}}{qs}$
The **upper bounds**			
Unit-length and equally spaced intervals	Algorithm 10	7.13	$\frac{4n}{(q-n/k)k}$
Variable length and equally spaced (big-small) intervals	Algorithm 10(a)	7.13	$\frac{4l_{min}}{(q-l_{min}/s)s}$

Table 7.1: The bounds for interval joins of overlapping intervals.

7.2 Unit-Length and Equally Spaced Intervals

We start with a special case of unit-length and equally spaced intervals. In reality, we are not expecting the data to be so regular, but looking at this case help us to derive a lower bound on the replication rate for any general algorithm that handles arbitrary collections of intervals.

Two relations X and Y, each of n unit-length intervals are given. We assume that all the intervals have their starting-points in $[0, k)$, *i.e.*, there is no interval that starts before 0 or at k. Thus, the space between every two successive intervals is $\frac{k}{n} < 1 \ll k$. In other words, the first interval starts at time 0, the second interval starts at time $\frac{k}{n}$, the third interval starts at time $\frac{2k}{n}$, and the last n^{th} interval starts at time $k - \frac{k}{n}$; see Figure 7.2.

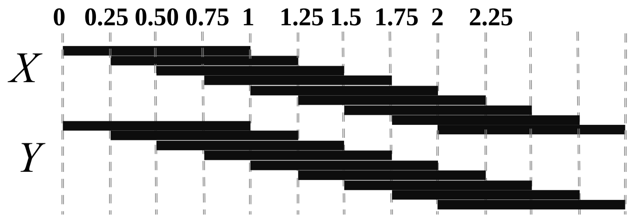


Figure 7.2: An example of unit-length and equally spaced intervals, where $n = 9$ and $k = 2.25$.

The output we want to produce is the set of all pairs of intervals such that one interval overlaps with the other interval in the pair. The problem is not really interesting if all these intervals exist on the input. The real assumption is that some fraction of them exist, and the reducer size q is selected so that the expected number of inputs that actually arrive at a given reducer is within the desired limit, *e.g.*, no more than what can be processed in main memory.

Recall that a *solution* to the problem of interval join of overlapping unit-length and equally spaced intervals is a mapping schema that assigns each pair of overlapping intervals, one from X and one from Y, to at least one reducer in common, without exceeding q inputs at any one reduce.

Since every two consecutive intervals have an equal space $(\frac{k}{n})$, an interval $x_i \in X$, which is not at one of the ends,[1] overlaps with at least $2\lfloor 1/\frac{k}{n} \rfloor + 1 = 2\lfloor \frac{n}{k} \rfloor + 1$ intervals of Y, because:

1. $\lfloor \frac{n}{k} \rfloor$ intervals of the relation Y have their ending-points between the starting-point and the ending-point of x_i.
2. $\lfloor \frac{n}{k} \rfloor$ intervals of the relation Y have their starting-points between the starting-point and the ending-point of x_i.

[1]Specifically, x_i does not have starting-point before 1 and after $k - 1$.

3. There is an interval $y_i \in Y$ that has the same end-points as x_i.

In this section, we will show a lower bound on the replication rate and the communication cost for interval join of overlapping unit-length and equally spaced intervals. Then, we provide an algorithm, its correctness, and an upper bound on the replication rate obtained by the algorithm.

Assumption. Now for the remaining chapter, we assume for simplicity that k divides n evenly.

Theorem 7.1 (Minimum replication rate) *Let there be two relations: X and Y, each of them containing n unit-length and equally spaced intervals in the range $[0, k)$, and let q be the reducer size. The replication rate for joining each interval of the relation X with all its overlapping intervals of the relation Y, is at least $\frac{2n}{qk}$.*

The proof of the above theorem appears in Appendix E.1.

Theorem 7.2 (Minimum communication cost) *Let there be two relations: X and Y, each of them containing n unit-length and equally spaced intervals in the range $[0, k)$, and let q be the reducer size. The communication cost for joining of each interval of the relation X with all its overlapping intervals of the relation Y is at least $\frac{4n^2}{qk}$.*

The proof of the above theorem appears in Appendix E.1.

Algorithm 10. An algorithm for 2-way interval join is given in [31], without regarding the reducer size and any analysis of bounds on replication of an interval. We include the concept of the reducer size and modify the algorithm, given in [31], for real scenarios. Two relations X and Y, each of them containing n unit-length and equally spaced intervals, are the inputs to the algorithm. Recall that it is expected that not all possible intervals are present.

We divide the time-range from 0 to k into equal-sized blocks of length $w = \frac{q-c}{4c}$, where $c = \frac{n}{k}$. Consider that we have P blocks by partitioning of the time-range. We now arrange P reducers, one for each block. We consider a block p_i, $1 \leq i \leq P$, and assign all the intervals of the relation X that exist in the block p_i to the i^{th} reducer. In addition, we assign all the intervals of the relation Y that have their starting or ending-point in the block p_i to the i^{th} reducer.

Theorem 7.3 (Algorithm correctness) *Let there be two relations: X and Y, each of them containing n unit-length and equally spaced intervals in the range $[0, k)$. Let w be the length of a block, and let $q = 4wc + c$ is the reducer size, where $c = \frac{n}{k}$. Algorithm 10 assigns each pair of overlapping intervals to at least one reducer in common.*

The proof of the above theorem appears in Appendix E.1.

From Algorithm 10, it is clear that an interval of the relation X is assigned to reducers corresponding to blocks that the interval i crosses, and an interval of the relation Y is assigned to at most two reducers. Now, we will present an upper bound on the replication rate and the communication cost for interval join of overlapping unit-length and equally spaced intervals.

Theorem 7.4 (Maximum replication rate) *Let there be two relations: X and Y, each of them containing n unit-length and equally spaced intervals in the range $[0, k)$. Let w be the length of a block, and let $q = 4wc + c < 2n$ is the reducer size, where $c = \frac{n}{k}$. The replication of an interval, for joining each interval of the relation X with all its overlapping intervals of the relation Y, is (i) at most 2 for $w \geq 1$ and (ii) at most $\frac{4c}{q-c}$ for $w < 1$.*

The proof of the above theorem appears in Appendix E.1.

Theorem 7.5 (Maximum communication cost) *Let there be two relations: X and Y, each of them containing n unit-length and equally spaced intervals in the range $[0, k)$. Let w be the length of a block, and let $q = 4wc + c < 2n$ is the reducer size, where $c = \frac{n}{k}$. The communication cost for joining of each interval of the relation X with all its overlapping intervals of the relation Y is (i) at most $4n$ for $w \geq 1$ and (ii) at most $\frac{8nc}{q-c}$ for $w < 1$.*

The proof of the above theorem appears in Appendix E.1.

7.3 Variable-Length and Equally Spaced Intervals

Now, we focus on a realistic scenario where two relations X and Y, each of them containing n intervals, are given such that all intervals can have non-identical length but adjacent intervals have equal spacing. We assume that the first interval starts at time 0, and the space ($s < 1$) between the beginnings of intervals is the same, but their endpoints will have quite different spacing (see Figure 7.3), where a relation X has 6 intervals, and a relation Y has also 6 intervals. A *solution* to the problem of interval join of overlapping variable-length and equally spaced intervals is a mapping schema such that each pair of overlapping intervals is sent to at least one reducer in common without exceeding q inputs at any one reducer.

We consider two types of intervals, as follows:

1. *Big and small intervals*: one of the relation, say X, is holding intervals of length l and the other relation, say Y, is holding intervals of length $l' \gg l$; we call intervals of the relations X and Y as *small intervals* and *big intervals*, respectively.
2. *Different-length intervals*: all the intervals of both the relations are of different-length.

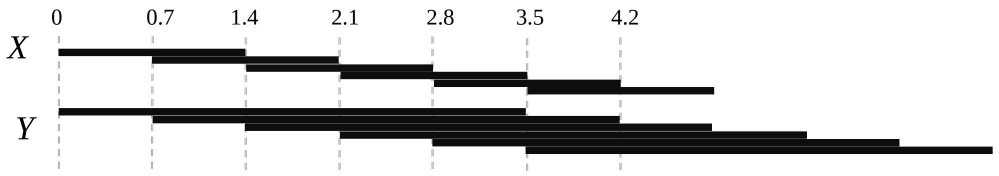


Figure 7.3: An example of big and small length but equally spaced intervals, where $n = 6$ and $s = 0.7$.

Throughout this section, we will use the following notations: l_{max}: the maximum length of an interval, l_{min}: the minimum length of an interval, and w: length of a block.

7.3.1 Big and small intervals

In this section, we consider a special case of variable-length and equally spaced intervals, where all the intervals of two relations X and Y have length l_{min} and l_{max}, respectively, such that $l_{min} \ll l_{max}$; see Figure 7.3. We call the intervals of the relations X and Y as *small intervals* and *big intervals*, respectively.

Since every two successive intervals have an equal space, s, an interval $x_i \in X$ of length l_{min} can overlap with at least $2\lfloor \frac{l_{min}}{s} \rfloor + 1$ intervals of the relation Y, because

1. $\lfloor \frac{l_{min}}{s} \rfloor$ intervals of the relation Y have their ending-points between the starting and the ending-points of x_i.
2. $\lfloor \frac{l_{min}}{s} \rfloor$ intervals of the relation Y have their starting-points between the starting and the ending-points of x_i.
3. There is an interval $y_i \in Y$ has an identical starting-point as x_i.

Now, we provide a lower bound on the replication rate for interval join of overlapping big- and small-length but equally spaced intervals.

Assumption. Now for the remaining chapter, we assume for simplicity that s divides l_{min} evenly.

Theorem 7.6 (Minimum replication rate) *Let there be two relations: X containing n small and equally spaced intervals and Y containing n big and equally spaced intervals, and let q be the reducer size. Let s be the spacing between every two successive intervals, and let l_{min} be the length of the smallest interval. The replication rate for joining of each interval of the relation X with all its overlapping intervals of the relation Y is at least $\frac{2l_{min}}{qs}$.*

The proof of the above theorem appears in Appendix E.2.

Theorem 7.7 (Minimum communication cost) *Let there be two relations: X containing n small and equally spaced intervals and Y containing n big and equally spaced intervals, and let q be the reducer size. Let s be the spacing between every two successive intervals,* *and let l_{min} be the length of the smallest interval. The communication cost for joining of each interval of the relation X with all its overlapping intervals of the relation Y is at least $\frac{4nl_{min}}{qs}$.*

The proof of the above theorem appears in Appendix E.2.

Algorithm 10(a). Algorithm 10(a) for interval join of overlapping intervals of a relation X of small and equally spaced intervals and a relation Y of big and equally spaced intervals works in a way similar to Algorithm 10. However, Algorithm 10(a) creates P blocks of the time-range (from 0 to ns), each of length $w = \frac{q-c}{4c}$, where $c = \frac{l_{min}}{s}$. After that, we follow the same procedure as followed in Algorithm 10.

Note that in Algorithm 10(a), small intervals are assigned to several reducers corresponding to their blocks that they cross, and large intervals are assigned to only two reducers corresponding to their stating- and ending-points' blocks.

Theorem 7.8 (Algorithm correctness) *Let there be two relations: X containing n small and equally spaced intervals and Y containing n big and equally spaced intervals. Let w be the length of a block, let s be the spacing between every two successive intervals, and let l_{min} be the length of the smallest interval. Let $q = 4wc + c$ is the reducer size, where $c = \frac{l_{min}}{s}$. Algorithm 10(a) assigns each pair of overlapping intervals to at least one reducer in common.*

We can prove the above theorem in a way similar to Theorem 7.3.

Theorem 7.9 (Maximum replication rate) *Let there be two relations: X containing n small and equally spaced intervals and Y containing n big and equally spaced intervals. Let s be the spacing between every two successive intervals, let w be the length of a block, and let l_{min} be the length of the smallest interval. Let $q = 4wc + c$ is the reducer size, where $c = \frac{l_{min}}{s}$. The replication rate for joining of each interval of the relation X with all its overlapping intervals of the relation Y is (i) at most 2 for $w \geq l_{min}$ and (ii) at most $\frac{4c}{q-c}$ for $w < l_{min}$.*

The proof of the above theorem appears in Appendix E.2.

Theorem 7.10 (Maximum communication cost) *Let there be two relations: X containing n small and equally spaced intervals and Y containing n big and equally spaced intervals. Let s be the spacing between every two successive intervals, let w be the length of a block, and let l_{min} be the length of the smallest interval. Let $q = 4wc + c$ is the reducer size, where $c = \frac{l_{min}}{s}$. The communication cost for joining of each interval of the relation X with all its overlapping intervals of the relation Y is (i) at most $4n$ for $w \geq l_{min}$ and (ii) at most $\frac{8nc}{q-c}$ for $w < l_{min}$.*

The proof of the above theorem appears in Appendix E.2.

7.3.2 Different-length intervals

We consider a case of different-length and equally spaced intervals. Let there be two relations: X and Y, each of them containing n different-length intervals, and s be the spacing between every two successive intervals; see Figure 7.4. For this case, the lower bound on the replication rate for joining of each interval of the relation X with all its overlapping intervals of the relation Y is same as given in Theorem 7.6.

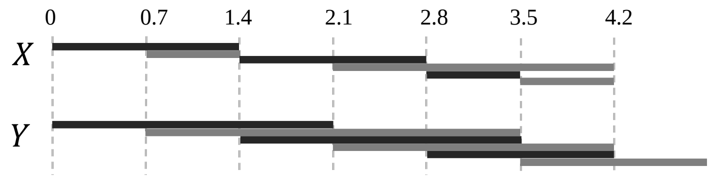


Figure 7.4: An example of different-length but equally spaced intervals, where $n = 6$ and $s = 0.7$.

Algorithm 10(b). Algorithm 10(b) for interval join of overlapping different-length and equally spaced intervals, which belong to two relations X and Y, each of them containing n intervals, works identically to Algorithms 10 and 10(a). Let l_{max} be the length of the largest interval. Algorithm 10(b) divides the time-range from 0 to ns into P blocks, each of length $w = \frac{q-c}{4c}$, where $c = \lceil \frac{l_{max}}{s} \rceil$. After that, we follow the same procedure as followed in Algorithm 10.

Theorem 7.11 (Algorithm correctness) *Let there be two relations: X and Y, each of them containing n different-length and equally spaced intervals. Let w be the length of a block, let s be the spacing between every two successive intervals, and let l_{max} be the length of the largest interval. Let $q = 4wc + c$ is the reducer size, where $c = \lceil \frac{l_{max}}{s} \rceil$. Algorithm 10(b) assigns each pair of overlapping intervals to at least one reducer in common.*

We can prove the above theorem in a way similar to Theorem 7.3.

7.3.3 An upper bound for the general case

In this section, we show an algorithm and an upper bound on the replication rate for the problem of interval join of variable-length but equally spaced intervals. We use the following notations:

1. T: the length of time in which all intervals exist, *i.e.*, all intervals begin at some time greater than or equal to 0 and end by time T.

2. n: the number of intervals in each of the two relations, X and Y.
3. S: the total length of all the intervals in one relation.
4. w: the length of time corresponding to one reducer, *i.e.*, we divide T into $\frac{T}{w}$ equal-length segments, each of length w.

Algorithm 10(c). Algorithm 10(c) works in a manner similar to Algorithms 10, 10(a), and 10(b) do. But this algorithm does more than Algorithms 10, 10(a), and 10(b). It finds all intervals that intersect, regardless of whether they overlap, are superimposed, or any other relation. We divide the time-range into $\frac{T}{w}$ equal-sized blocks and arrange $\frac{T}{w}$ reducers, one for each block. After that, we follow the same procedure as followed in Algorithm 10.

Theorem 7.12 (Algorithm correctness) *Let there be two relations: X and Y, each of them containing n intervals. Let S be the total length of all the intervals in one relation, let w be the length of a block, let T be the length of time in which all intervals exist, and let $q = \frac{3nw+S}{T}$ is the reducer size. Algorithm10(c) assigns each pair of overlapping intervals to at least one reducer in common.*

The proof of the above theorem appears in Appendix E.2.1.

Theorem 7.13 (Replication rate) *Let there be two relations: X and Y, each of them containing n intervals. Let S be the total length of all the intervals in one relation, let w be the length of a block, let T be the length of time in which all intervals exist, and let $q = \frac{3nw+S}{T}$ is the reducer size. The replication rate for joining each interval of the relation X with all its overlapping intervals of the relation Y is at most $\frac{3}{qT-S}\frac{S}{2}$.*

The proof of the above theorem appears in Appendix E.2.1.

Chapter 8

Computing Marginals of a Data Cube

In this chapter, we continue to explore the communication cost in the case of identical-sized inputs, as we did in Chapter 7, and focus on the problem of computing marginals of a data cube. A data cube is a tool for analyzing high dimensional data. For example, consider Figure 8.1 of a 3-dimensional data cube having dimensions such as $Time$, $User$, and $City$. This cube stores the total number of users accessing social media at a particular time across different cities. The number of dimensions can be increased by adding dimensions such as Day, $Month$, $Year$, and $Site\_name$. A user may easily solve a query such as the total number of users from New York and London accessing the Website at 8am, 9am, and 10am. Related work on data cube is presented in Appendix F.1.[1]

8.1 Preliminaries

8.1.1 Marginals

Consider an n-dimensional data cube and the computation of its marginals by MapReduce. A *marginal* of a data cube is the aggregation of the data in all those tuples that have fixed values in a subset of the dimensions of the cube. We assume this aggregation is the sum, but the exact nature of the aggregation is unimportant in what follows. If the value in a dimension is fixed, then the fixed value represents the dimension. If the dimension is aggregated, then there is a * for that dimension. The number of dimensions over which we aggregate is the *order* of the marginal.

Example 8.1 *Suppose* $n = 5$*, and the data cube is a relation DataCube(D1,D2,D3,D4,D5,V). Here, D1 through D5 are the dimensions, and* V *is*

[1] I am thankful to Prof. Foto Afrati, Prof. Jeffrey Ullman, and Prof. Jonathan Ullman who helped me a lot to write the content of this chapter.

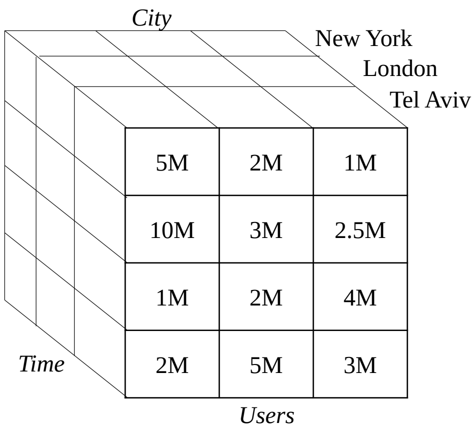


Figure 8.1: An example of a 3-dimensional data cube.

the value that is aggregated.

```
SELECT SUM(V)
FROM DataCube
WHERE D1 = 10 AND D3 = 20 AND D4 = 30;
```

will sum the data values in all those tuples that have value 10 in the first dimension, 20 in the third dimension, 30 in the fourth dimension, and any values in the second and fifth dimension of a five-dimensional data cube. We can represent this marginal by the list $[10, *, 20, 30, *]$*, and it is a second-order marginal.*

We make the simplifying assumption that in each dimension there are d different values. In practice, we do not expect to find that each dimension really has the same number of values. For example, if one dimension represents Amazon customers, there would be millions of values in this dimension. If another dimension represents the date on which a purchase was made, there would “only” be thousands of different values.

8.1.2 Mapping schema

We define a mapping schema in order to compute all the k^{th}-order marginals, as follows:

A mapping schema assigns a set of inputs (the tuples or points of the data cube, here) to a number of reducers so that

1. No reducer is associated with more than q inputs, and
2. For every output (the values of the marginals), there is some reducer that is associated with all the inputs that output needs for its computation.

Here, the point (1) puts a constraint on each reducer, defined as the reducer size. The point (2) provides a mapping between inputs and outputs so that the algorithm produces the desired results.

8.1.3 Naïve solution: computing one marginal per reducer

Consider the problem of computing all the marginals of a data cube in the above model. If we are not careful, the problem becomes trivial. The marginal that aggregates over all dimensions is an output that requires all d^n inputs of the data cube. Thus, $q = d^n$ is necessary to compute all the marginals. But that means we need a single reducer as large as the entire data cube, if we are to compute all marginals in one round. As a result, it only makes sense to consider the problem of computing a limited set of marginals in one round.

The k^{th}-order marginals are those that fix $n - k$ dimensions and aggregate over the remaining k dimensions. To compute a k^{th}-order marginal, we need $q \geq d^k$, since such a marginal aggregates over d^k tuples of the cube. Thus, we could compute all the k^{th}-order marginals with $q = d^k$, using one reducer for each marginal. We can represent the problem in terms of a mapping schema such that there are d^n inputs, $d^{n-k}\binom{n}{k}$ outputs, each representing one of the marginals, and $q = d^k$. Each output is connected to the d^k inputs over which it aggregates. Each input contributes to $\binom{n}{k}$ marginals – those marginals that fix $n - k$ out of the n dimensions in a way that agrees with the tuple in question. That is, for $q = d^k$, we can compute all the k^{th}-order marginals with a replication rate r equal to $\binom{n}{k}$.

For $q = d^k$, there is nothing better we can do. However, when q is larger, we have a number of options, and the purpose of this chapter is to explore these options.

8.2 Computing Many Marginals at One Reducer

We study the tradeoff between reducer size and the replication rate for computing the k^{th}-order marginals.

8.2.1 Covering marginals

We want to compute all k^{th}-order marginals, but we are willing to use reducers of size $q = d^m$ for some $m > k$. If we fix any $n - m$ of the n dimensions of the data cube, we can send to one reducer the d^m tuples of the cube that agree with those fixed values. We then can compute all the marginals that have $n - k$ fixed values, as long as those values agree with the $n - m$ fixed values that we chose originally.

Example 8.2 *Let $n = 7$, $k = 2$, and $m = 3$. Suppose we fix the first $n - m = 4$ dimensions, say using values a_1, a_2, a_3, and a_4. Then we can cover the d marginals $a_1a_2a_3a_4x**$ for any of the values x that may appear in the fifth dimension. We can also cover all marginals $a_1a_2a_3a_4{*}y{*}$ and $a_1a_2a_3a_4{*}{*}z$, where y and z are any of the possible* *values for the sixth and seventh dimensions, respectively. Thus, we can cover a total of $3d$ second-order marginals at this one reducer. That turns out to be the largest number of marginals we can cover with one reducer of size $q = d^3$.*

8.2.2 From marginals to sets of dimensions

To understand why the problem is more complex than it might appear at first glance, let us continue thinking about the simple case of Example 8.2. We need to cover all second-order marginals, not just those that fix the first four dimensions. If we had one team of d^4 reducers to cover each four of the seven dimensions, then we would surely cover all second-order marginals. But we do not need all $\binom{7}{2} = 21$ such teams. Rather, it is sufficient to pick a collection of sets of four of the seven dimensions, such that every set of five of the seven dimensions contains one of those sets of size four.

In what follows, we find it easier to think about the sets of dimensions that are aggregated, rather than those that are fixed. So we can express the situation above as follows. Collections of second-order marginals are represented by pairs of dimensions – the two dimensions such that each marginal in the collection aggregates over those two dimensions. These pairs of dimensions must be *covered* by sets of three dimensions – the three dimensions aggregated over by one third-order marginal. Our goal, which we will realize in Example 8.3 below, is to find a smallest set of tripletons such that every pair chosen from seven elements is contained in one of those tripletons.

In general, we are faced with the problem of covering all sets of k out of n elements by the smallest possible number of sets of size $m > k$. Such a solution leads to a way to compute all k^{th}-order marginals using as few reducers of size d^m as possible. We will refer to the sets of k dimensions as *marginals*, even though they really represent teams of reducers that compute large collections of marginals with the same fixed dimensions. We will call the larger sets of size m *handles*. The implied MapReduce algorithm takes each handle and creates from it a team of reducers that are associated, in all possible ways, with fixed values in all dimensions except for those dimensions in the handle. Each created reducer receives all inputs that match its associated values in the fixed dimensions.

Example 8.3 *Call the seven dimensions $ABCDEFG$. Then here is a set of seven handles (sets of size three), such that every marginal of size two is contained in one of them:*

$$ABC,\ ADE,\ AFG,\ BDF,\ BEG,\ CDG,\ CEF$$

To see why these seven handles suffice, consider three cases, depending on how many of A, B, and C are in the pair of dimensions to be covered.

Case 0: If none of A, B or C is in the marginal, then the marginal consists of two of D, E, F, and G. Note that all six such pairs are contained in one of the last six of the handles.
Case 1: If one of A, B, or C is present, then the other member of the marginal is one of D, E, F, or G. If A is present, then the second and third handles, ADE and AFG together pair A with each of the latter four dimensions, so the marginal is covered. If B is present, a similar argument involving the fourth and fifth of the handles suffices, and if C is present, we argue from the last two handles.
Case 2: If the marginal has two of A, B, and C, then the first handle covers the marginal.

Incidentally, we cannot do better than Example 8.3. Since no handle of size three can cover more than three marginals of size two, and there are $\binom{7}{2} = 21$ marginals, clearly seven handles are needed.

As a strategy for evaluating all second-order marginals of a seven-dimensional cube, let us see how the reducer size and replication rate compare with the baseline of using one reducer per marginal. Recall that if we use one reducer per marginal, we have $q = d^2$ and $r = \binom{7}{5} = 21$. For the present method, we have $q = d^3$ and $r = 7$. That is, each tuple is sent to the seven reducers that have the matching values in dimensions $DEFG$, $BCFG$, and so on, each set of attributes on which we match corresponding to the complement of one of the seven handles mentioned in Example 8.3.

8.2.3 Covering numbers

We define $C(n, m, k)$ to be the minimum number of sets of size m out of n elements such that every set of k out of the same n elements is contained in one of the sets of size m. For instance, Example 8.3 showed that $C(7, 3, 2) = 7$. $C(n, m, k)$ is called the *covering number* in [22]. The numbers $C(n, m, k)$ guide our design of algorithms to compute k^{th}-order marginals. There is an important relationship between covering numbers and replication rate, that justifies our focus on constructive upper bounds for $C(n, m, k)$.

Theorem 8.4 *If $q = d^m$, then we can solve the problem of computing all k^{th}-order marginals of an n-dimensional data cube with $r = C(n, m, k)$.*

The proof of the theorem is given in Appendix F.2.

8.2.4 First-order marginals

The case $k = 1$ is quite easy to analyze. We are asking how many sets of size m are needed to cover each singleton set, where the elements are chosen from a set of size n. It is easy to see that we can group the n elements into $\lceil n/m \rceil$ sets so that each of the n elements is in at least one of the sets, and there is no way to cover all the singletons with fewer than this number of sets of size m. That is, $C(n, m, 1) = \lceil n/m \rceil$. For example, If $n = 7$ and $m = 2$, then the seven dimensions $ABCDEFG$ can be covered by four sets of size 2, such as AB, CD, EF, and FG.

8.2.5 2nd-order marginals covered by 3rd-order handles

The next simplest case is $C(n, 3, 2)$, *i.e.*, covering second-order marginals by third-order marginals. First we will look at the lower bound on the number of handles that are required to cover all the second-order marginals.

Theorem 8.5 (Lower bound on the number of handles) *For an n-dimensional data cube, where the marginals are of size two, the minimum number of handles is $\lfloor \frac{n(n-1)}{6} \rfloor$.*

The proof of the theorem is given in Appendix F.2.

We now present an algorithm for constructing 3rd-order handles so that all the 2nd-order marginals of a data cube of $n = 3^i$, $i > 0$, dimensions are covered. We will show that the upper bound on the number of handles obtained by the algorithm meets the lower bound on the number of handles to cover all the marginals.

Algorithm 1: A data cube of $n = 3^i, i > 0$, dimensions in an input to Algorithm 1. Algorithm 1 constructs a set of handles of size three that can cover all the marginals of size two. Algorithm 1 uses the following recurrence:

$$C(n, 3, 2) \leq (n/3)^2 + 3 \times C(n/3, 3, 2)$$

$$\begin{array}{cc} \langle D_1, D_4, D_7 \rangle & \langle D_1, D_5, D_9 \rangle \\ \multicolumn{2}{c}{\langle D_1, D_6, D_8 \rangle} \\ \langle D_2, D_4, D_9 \rangle & \langle D_2, D_5, D_8 \rangle \\ \multicolumn{2}{c}{\langle D_2, D_6, D_7 \rangle} \\ \langle D_3, D_4, D_8 \rangle & \langle D_3, D_5, D_7 \rangle \\ \multicolumn{2}{c}{\langle D_3, D_6, D_9 \rangle} \end{array}$$

(a)

$$\begin{array}{cc} \langle D_1, D_2, D_3 \rangle & \langle D_4, D_5, D_6 \rangle \\ \multicolumn{2}{c}{\langle D_7, D_8, D_9 \rangle} \end{array}$$

(b)

Figure 8.2: The handles of size three for a 9-dimensional data cube.

Theorem 8.6 (Upper bound on the number of handles) *For an n-dimensional data cube, where $n = 3^i$, $i > 0$, and the marginals are of size two, the number of handles is bounded above by*

$$C(n, 3, 2) \leq \frac{n^2}{6}, n = 3^i, i > 0.$$

```
Algorithm 1: Constructing handles for $C(3^i, 3, 2)$, $i > 0$
Input: A data cube of $n = 3^i$, $i > 0$, dimensions
Output: A set of handles of size three so that all the marginals of size two are covered
1 Divide the $n = 3^i$, $i > 0$, dimensions into three groups, where each group holds
    $p = 3^{i-1}$ dimensions. In particular, the first group holds the first $p$ dimensions
    $(D_1, D_2, \ldots, D_p)$, the second group holds the next $p$ dimensions
    $(D_{p+1}, D_{p+2}, \ldots, D_{2p})$, and the last group holds the remaining $p$ dimensions
    $(D_{2p+1}, D_{2p+2}, \ldots, D_{3p})$.
  For example, for $n = 9$, we have three groups as follows: $\langle D_1, D_2, D_3 \rangle$ are in the first
    group, $\langle D_4, D_5, D_6 \rangle$ are in the second group, and $\langle D_4, D_5, D_6 \rangle$ are in the third group.
2 Find three dimensions out of the $n$ dimensions. Select one dimension from each group
    using the following rule: $(i + j + k) = 0 \bmod p$, where the $i^{th}$ dimension belongs to
    the first group, the $j^{th}$ dimension belongs to the second group, and the $k^{th}$ dimension
    belongs to the third group. By this step, we construct $p^2$ handles.
  See Figure 8.2a, where we selected three dimensions, one from each group, in the case
    of a 9-dimensional data cube.
3 Apply the algorithm to each group.
  See Figure 8.2b, where we apply Algorithm 1 in each group.
```

The proof of the theorem is given in Appendix F.2. The result of the above theorem is a known result in block design [20]. Note that all optimal values of $C(n, 3, 2)$ up to $n = 13$ are given in [22].

Corollary 8.7 *If $q = d^3$ and n is a power of 3, then we can compute all second-order marginals with a replication rate of $n^2/6 - n/6$.*

8.2.6 A slower recursion for 2nd-order marginals

There is an alternative recursion for constructing handles that offers solutions for $C(n, 3, 2)$. This recursion is not as good asymptotically as that of Section 8.2.5; it uses approximately $n^2/4$ rather than $n^2/6$ handles. However, this recursion gives solutions for any n, not just those that are powers of 3. Algorithm 2 provides a solution to any value of n and uses the following recurrence:

$$C(n, 3, 2) \leq n - 2 + C(n - 2, 3, 2)$$

Theorem 8.8 (Upper bound on the number of handles) *For an n-dimensional data cube, where the marginals are of size two, the number of handles is bounded above by*

$$C(n, 3, 2) \leq \frac{(n-1)^2}{4}$$

```
Algorithm 2: Constructing handles for $C(n,3,2)$
    Input: A data cube of $n$ dimensions
    Output: A set of handles of size three so that all the marginals of size two are covered
1   Create $n-2$ handles such that each of these handles has the last two dimensions and
      one of the first $n-2$ dimensions.
    For example, for $n=7$, we construct 5 handles by following this step; see Figure 8.3a.
2   Recursively create handles for the first $n-2$ dimensions.
    For example, for $n=7$, we recursively construct 4 handles; see Figure 8.3b.
```

(a)	(b)
$\langle D_1, D_6, D_7\rangle$	
$\langle D_2, D_6, D_7\rangle$	$\langle D_1, D_4, D_5\rangle$
$\langle D_3, D_6, D_7\rangle$	$\langle D_2, D_4, D_5\rangle$
$\langle D_4, D_6, D_7\rangle$	$\langle D_3, D_4, D_5\rangle$
$\langle D_5, D_6, D_7\rangle$	$\langle D_1, D_2, D_3\rangle$

Figure 8.3: Constructing handles of size 3 for a 7-dimensional data cube.

where $n \geq 5$ and n is odd[2].

The proof of the theorem is given in Appendix F.2.

8.2.7 Covering 2nd-order marginals with larger handles

We want to cover dimensions by sets of size larger than three; *i.e.*, we wish to cover second-order marginals by handles of size $m \geq 4$. We can generalize the technique of Section 8.2.6. Algorithm 3 provides a solution to any value of m and uses the following recurrence:

$$C(n,m,2) \leq n-(m-1)+C(n-(m-1),m,2)$$

```
Algorithm 3: Constructing handles for $C(n,m,2)$, $m \geq 4$
    Input: A data cube of $n$ dimensions
    Output: A set of handles of size $m$ so that all the marginals of size two are covered
1   Create $n-(m-1)$ handles such that each of these handles has the last $m-1$
      dimensions and one of the first $n-(m-1)$ dimensions.
    For example, for $n=8$, we construct 5 handles by following this step; see Figure 8.4a.
2   Recursively create handles for the first $n-(m-1)$ dimensions.
    For example, for $n=8$, we recursively construct 3 handles; see Figure 8.4b.
```

[2]When n is even, the number of handles is bounded above by $C(n,3,2) \leq \lceil \frac{(n-1)^2}{4} \rceil$, where $n \geq 8$.

$\langle D_1, D_6, D_7, D_8 \rangle$
$\langle D_2, D_6, D_7, D_8 \rangle$
$\langle D_3, D_6, D_7, D_8 \rangle$
$\langle D_4, D_6, D_7, D_8 \rangle$
$\langle D_5, D_6, D_7, D_8 \rangle$

(a)

$\langle D_1, D_3, D_4, D_5 \rangle$
$\langle D_2, D_3, D_4, D_5 \rangle$
$\langle D_1, D_2, D_3, D_4 \rangle$

(b)

Figure 8.4: Constructing handles of size 4 for a 8-dimensional data cube.

Theorem 8.9 (Upper bound on the number of handles) *For an n-dimensional data cube, where the marginals are of size two, the number of handles is bounded above by*

$$C(n, m, 2) \leq \frac{n^2}{2(m-1)}$$

where $n \geq 5$.

The proof sketch of the above theorem appears in Appendix F.2.

8.3 The General Case

Finally, we present Algorithm 4 for $C(n, m, k)$ that works for all n and for all $m > k$. However, it does not approach the lower bound, but it is significantly better than using one handle per marginal. This method generalizes Algorithm 2. Algorithm 4 uses the following recurrence:

$$C(n) \leq \binom{n-m+k-1}{k-1} + C(n-m+k-1, m, k)$$

```
Algorithm 4: Constructing handles for $C(n, m, k)$
  Input: A data cube of $n$ dimensions
  Output: A set of handles of size $m$ so that all the marginals of size $k < m$ are covered
1 Create $\binom{n-m+k-1}{k-1}$ handles such that each of these handles has the last $m-k+1$
    dimensions and any $k-1$ dimensions out of the first $n-(m-k+1)$ dimensions.
2 Recursively create handles for the first $n-(m-k+1)$ dimensions.
```

We claim that every marginal of size k is covered by one of these handles. If the marginal has at least one dimension out of the last $m-k+1$ dimensions, then it has at most $k-1$ dimensions out of the first $n-(m-k+1)$ dimensions. Therefore, it is covered by the handles from step (1). And if the marginal has no last dimensions, then it is surely covered by a handle from step (2).

Theorem 8.10 (Upper bound on the number of handles) $C(n) \leq \binom{n}{k}/(m-k+1)$ *for n equal to 1 plus an integer multiple of $m-k+1$.*

Proof. The proof is by induction on n.
Basis case. We know $C(m) = 1$, and $\binom{m}{k}/(m - k + 1) \geq 1$ for any $1 \leq k < m$.
Inductive step. By the recurrence relation,

$$C(n) \leq \binom{n - m + k - 1}{k - 1} + C(n - m + k - 1, m, k)$$

We know that

$$\binom{n - m + k - 1}{k - 1} + \frac{\binom{n-m+k-1}{k}}{(m - k + 1)}$$

is an upper bound on $C(n, m, k)$. We therefore need to show that

$$\frac{\binom{n}{k}}{m - k + 1} \geq \binom{n - m + k - 1}{k - 1} + \frac{\binom{n-m+k-1}{k}}{m - k + 1}$$

Equivalently,

$$\binom{n}{k} \geq (m - k + 1)\binom{n - m + k - 1}{k - 1} + \binom{n - m + k - 1}{k} \tag{8.1}$$

The left side of the above Equation 8.1 is all ways to pick k things out of n. The right side counts a subset of these ways, specifically those ways that pick either:

1. Exactly one of the first $m - k + 1$ elements and $k - 1$ of the remaining elements, or
2. None of the first $m - k + 1$ elements and k from the remaining elements.

Thus, Equation 8.1 holds, and $C(n, m, k) \leq \binom{n}{k}/(m - k + 1)$ is proved. ■

The bound of Theorems 8.10 and 8.4 gives us an upper bound on the replication rate:

Corollary 8.11 *We can compute all k^{th}-order marginals using reducers of size $q = d^m$, for $m > k$, with a replication rate of $r \leq \binom{n}{k}/(m - k + 1)$.*

We also demonstrate that for a given reducer size q, the largest number of marginals of a given order k that we can cover with a single reducer occurs when the reducer gets all tuples needed for a marginal of some higher order m. The proof extends the ideas found in [28, 68] regarding isoperimetric inequalities for the hypercube. This part may be found in our paper [16], and we acknowledge Prof. Jonathan Ullman for this work.

Part III

Replication Aspects in Secure and Privacy-Preserving MapReduce

Chapter 9

Security and Privacy Aspects in MapReduce

Security and privacy of data and MapReduce computations are essential concerns when a MapReduce computation is executed in public clouds or in hybrid clouds. In order to execute a MapReduce job on public and hybrid clouds, authentication of mappers-reducers, confidentiality of data-computations, integrity of data-computations, and correctness-freshness of the outputs are required. Satisfying these requirements shield the operation from several types of attacks on data and MapReduce computations.

Security of MapReduce ensures a *legitimate* functionality of the framework. A secure MapReduce framework deals with the following attacks: attacks on authentication (impersonation and replay attacks), attacks on confidentiality (eavesdropping and man-in-the-middle attacks), data tampering (modification of input data, intermediate outputs, and the final outputs), hardware tampering, software tampering (modification of mappers and reducers), denial-of-service, interception-release of data as well as computations, and communication analysis.

Privacy aspects assume *legitimate* functionality of the framework, and thus, are built on top of security. On top the correctly functioning framework, privacy in the context of MapReduce is an ability of each participating party (data providers, cloud providers, and users) to prevent other, possibly adversarial parties from observing data, codes, computations, and outputs. In order to ensure privacy, a MapReduce algorithm in public clouds hides data storage as well as computation to public clouds and adversarial users. Most of the security and privacy requirements and corresponding cloud layers are depicted in Figure 9.1, the figure shows a complete picture of the considered cloud structure, with different participating parties: data providers on the left, cloud provider in the middle and users on the right, and their specific security requirements. The figure also depicts various cloud levels and their relation to the security and privacy mechanisms, while assuming the cloud provides privacy-preserving computations.

In this chapter and in the next chapter, our focus is on the privacy-preserving computations using MapReduce in the clouds. Thus, we do not provide details of security in MapReduce, which may be found in our review paper [39], and details of security-privacy aspects in the cloud, which may be found in [21, 102, 118].

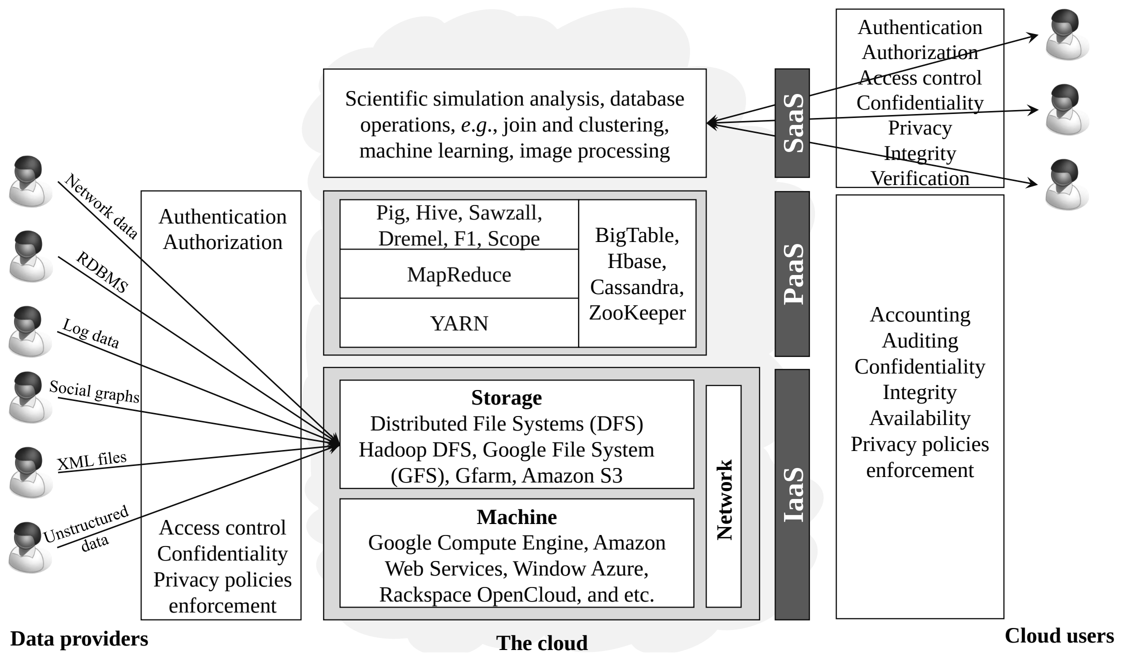

Network data
RDBMS
Log data
Social graphs
XML files
Unstructured data
Authentication
Authorization
Access control
Confidentiality
Privacy policies enforcement
Scientific simulation analysis, database operations, e.g., join and clustering, machine learning, image processing
Pig, Hive, Sawzall, Dremel, F1, Scope
MapReduce
YARN
BigTable, Hbase, Cassandra, ZooKeeper
Storage
Distributed File Systems (DFS) Hadoop DFS, Google File System (GFS), Gfarm, Amazon S3
Machine
Google Compute Engine, Amazon Web Services, Window Azure, Rackspace OpenCloud, and etc.
Network
SaaS
PaaS
IaaS
Authentication
Authorization
Access control
Confidentiality
Privacy
Integrity
Verification
Accounting
Auditing
Confidentiality
Integrity
Availability
Privacy policies enforcement
Data providers
The cloud
Cloud users

Figure 9.1: Security and privacy requirements in MapReduce environment in the cloud.

9.1 Security and Privacy Challenges in MapReduce

Massive parallel processing style of MapReduce is substantially different from the classical computation in the cloud leading to distinct design challenges for security and privacy, which we highlight here, as: size of input data and its storage, highly distributed nature of MapReduce computations, dataflow (between data storage and computing nodes, and between public clouds), the black-box nature of public clouds, hybrid clouds, scalability, fault tolerance, transparency, and untrusted data access. In addition, cloud computing and the deployment of MapReduce on public clouds present a new set of challenges in privacy of data such as data privacy protection from adversarial cloud providers, protection of data from adversarial users, and multiusers on a single public cloud.

9.2 Privacy Requirements in MapReduce

MapReduce inherently decouples data providers, cloud providers, and users that execute queries over data. Referring to the cloud structure depicted in Figure 9.1, data providers upload data to the cloud provider, and cloud users perform queries on data. However, despite separation between different entities, ensuring privacy in those settings is still challenging. Here, we provide requirements of privacy in MapReduce framework, deployed on the hybrid cloud or the public cloud.

Protection of data providers. In a setting where data is uploaded to the cloud by various data providers, each data provider might have a different privacy requirements. The cloud provider has to ensure that those privacy requirements are met even in the presence of adversarial users. Moreover, different data providers might require a different privacy level for various data sets. The privacy framework should allow adaptation of privacy levels for those requirements.

Untrusted cloud providers. As an adversarial cloud provider can perform any computation on data for revealing data, modifying data, and producing wrong outputs, data has to be protected from cloud providers. In addition to protect the data from cloud providers, privacy framework has to be able to protect the performed computations as well. As an example, consider a user querying for specific information. Even if the data results are not released to the cloud provider, it is possible to learn the intent of the user from observing performed computations.

Utilization and privacy tradeoff. A data provider can encode/encrypt/secret-shared her data in a way that no information can be learnt from it. However, this will also prevent the user from performing any computation on the data, and thus, decreases utilization of MapReduce. As such, MapReduce privacy framework has to provide maximum possible utilization, while still preserving data privacy according to data providers' requirements.

Efficiency. In most of the public clouds, users are tariffed for usage and storage. Hence, the privacy framework has to be efficient in terms of CPU and memory consumption, and in the amount of storage required. If the privacy framework provides high overhead, it could be more cost-effective to perform computations on the private cloud, where physical security solves privacy issues.

9.3 Adversarial Models for MapReduce Privacy

We explain adversarial models that are applied in privacy settings in the context of MapReduce deployment in the clouds.

Honest-but-Curious adversary. This type of adversary mostly applies to cloud providers.

Curious cloud providers can breach the privacy of data and MapReduce computations very easily, since the whole cluster is under the control of cloud providers, which have all types of privileged access to data and computing nodes. It is important to note that in reality curious cloud providers are not necessarily adversaries by choice, but rather might be compliant by court law, regulations, and governmental requests.[1]
Malicious adversary. This type of adversary applies to a user that tries to learn, modify, or delete information from the data by issuing various queries. In general, cloud providers are not assumed to be malicious, as assuring privacy with malicious cloud providers requires a high level of privacy measures that considerably reduce the utilization of the framework.
Knowledgeable adversary. A knowledgeable adversary applies to both a cloud provider and a user, who is trying to learn, modify, or delete information. Knowledgeable adversary is assumed to have a complete knowledge of MapReduce framework, the cloud structure, and is able to use any algorithm or cryptography drawback.
Network and node adversary. A cloud provider working as a network and node adversary has all the privileged access to computing nodes and the entire cloud infrastructure. A real-world example of such adversary is a cloud provider employee that breaches sensitive information most clearly shown by Edward Snowden case. It is impossible to hide any MapReduce computation or data from this type of adversary [99].

[1]http://www.zdnet.com/article/microsoft-admits-patriot-act-can-access-eu-based-cloud-data/

Chapter 10

Privacy-Preserving Computations using MapReduce

Data outsourcing allows data owners to keep their data in the public clouds. However, as we saw in the previous chapter, a public cloud does not ensure the privacy of data or computations. Hence, in this chapter, we investigate and present techniques for executing MapReduce computations in the public cloud, while preserving the privacy.

Specifically, we propose a technique to outsource a database based on replication of information of the form of secret-shares, (created using Shamir's secret-sharing [101] (SSS) scheme) to the public clouds, and then, provide privacy-preserving algorithms for performing count, search and fetch, equijoin, and range queries using MapReduce. A user can execute her queries using accumulating-automata (AA) [46][1] on these secret-shares without revealing queries/data to the cloud. Consequently, in our proposed algorithms, the public cloud cannot learn the database or computations.

All the proposed algorithms eliminate the role of the database owner, which only creates and distributes secret-shares only once, as compared to the existing solutions [55, 56, 81, 77, 32]. In addition, the proposed algorithms minimize the role of the user, which has to perform a simple operation (especially, polynomial interpolation using Lagrange polynomials [35]) for reconstructing the result, for query processing.

10.1 Motivating Examples

We present two examples (search and equijoin) to show the need for security and privacy of data and query execution using MapReduce in the public cloud.

[1]Our MapReduce-based count operation (Section 10.4) adapts the basic working of AA; hence, we provide the basic working of AA in that section. The remaining advanced operations of AA are detailed in [46].

Secure and privacy-preserving search. *Problem statement*: Consider a hospital database that can have different users, *e.g.*, doctors, nurses, insurance companies, and database administrators. As there are different users that may search in the database, on one hand, it is required that only an authenticated and authorized user will find the desired result. On the other hand, maintaining a database in the hospital is not a trivial and cheap task. Hence, it is beneficial to outsource the database to the public clouds.

The public clouds, however, do not ensure the privacy of data and computations; any user or the cloud can breach the privacy of data and computations. Therefore, it is necessary to keep a database in the cloud in a privacy-preserving manner so that only authenticated and authorized users can access and know the database.

Secure and privacy-preserving equijoin of two relations $X(A, B)$ and $Y(B, C)$. *Problem statement*: The join of relations $X(A, B)$ and $Y(B, C)$, where the joining attribute is B, provides output tuples $\langle a, b, c\rangle$, where (a, b) is in X and (b, c) is in Y. In the equijoin of $X(A, B)$ and $Y(B, C)$, all tuples of both the relations with an identical value of the attribute B should appear together for providing the final output tuples.

Consider that the relations X and Y belong to two organizations, *e.g.*, a company and a hospital, while a third user wants to perform the equijoin. However, both the organizations want to provide results, while maintaining the privacy of their databases, *i.e.*, without revealing the whole database to the other organization and the user. Hence, it is required to perform the equijoin in a secure and privacy-preserving manner.

10.2 System Settings

We consider, for the first time, data and MapReduce-based computation outsourcing of the form of secret-shares to c *non-communicating* clouds. The meaning of non-communicating clouds is that they do not exchange data with each other, only exchange data with the user or the database owner.

The system architecture. The architecture is simple but powerful and assumes the following:

STEP 1. A data owner outsources her databases of the form of secret-shares to c (non-communicating) clouds only once; see STEP 1 in Figure 10.1. We use c clouds to provide privacy-preserving computations using SSS. Note that a single *non-trustworthy* cloud cannot provide privacy-preserving computations using secret-sharing.

STEP 2. A preliminary step is carried out at the user-side wishing to perform a MapReduce computation. The user sends a query of the form of secret-shares to all c clouds to find the desired result of the form of secret-shares; see STEP 2 in Figure 10.1. The query must be sent to at least $c' < c$ number of clouds, where c' is the threshold of SSS.

STEP 3. The clouds deploy a *master process* that executes the computation by assigning the *map tasks* and the *reduce tasks*; see STEP 3 in Figure 10.1. The user interacts only with the master process in the cloud, and the master process provides the addresses of the outputs to the user. It must be noted that the communication between the user and the clouds is presumed to be the same as the communication between the user and the master process.

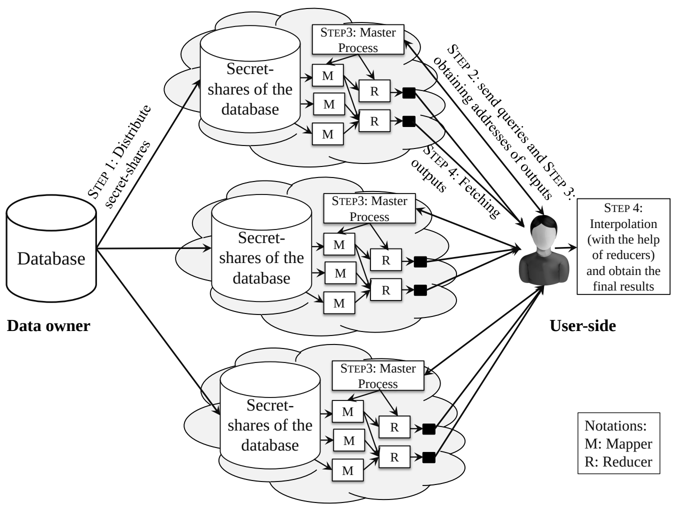

STEP3: Master Process
Secret-shares of the database
M
R
STEP 1: Distribute secret-shares
STEP 2: send queries and STEP 3: obtaining addresses of outputs
STEP 4: Fetching outputs
Database
Data owner
User-side
STEP 4: Interpolation (with the help of reducers) and obtain the final results
Notations: M: Mapper R: Reducer

Figure 10.1: The system architecture.

STEP 4. The user fetches the outputs from the clouds and performs the interpolation (with the help of reducers) for obtaining the secret-values; see STEP 4 in Figure 10.1.

In this system setting, users wish to execute their MapReduce computations without revealing the computation to the clouds, while the database owner wishes to store its database and perform queries' execution in public clouds without compromising the privacy.

Note. Physical machines of a single cloud provider can be compromised as well, possibly leaking information (through the network) they received when participating in the MapReduce; thus, secret sharing will make the leaked information meaningless, as long as the number of leaked machines is less than the threshold or the compromised machines are controlled by different (non-collaborating) adversaries.

Adversarial settings. We assume, on one hand, that an adversary cannot launch any attack against the data owner, who is trustworthy. Also, the adversary cannot access the secret-sharing algorithm and machines at the database owner side.

On the other hand, an adversary can access public clouds and data stored therein. Hence, the adversary can also access input, intermediate, and output data of a MapReduce job. A user who wants to perform a computation on the data stored in public clouds may also behave as an adversary. Moreover, the cloud itself can behave as an adversary, since it has complete privileges to all the machines and storage. Both the user and the cloud can launch any attack to compromise the privacy of data or computations.

We consider an honest-but-curious adversary, which is considered in the standard settings for security in the public cloud [112, 30, 93]. The honest-but curious adversary performs assigned computations correctly, but tries to breach the privacy of data or MapReduce computations, by analyzing data, computations, or data flow. However, such an adversary does not modify or delete information from the data.

We assume that an adversary can know less than $c' < c$ clouds locations that store databases and execute queries. Recall that c' is the threshold of SSS. In addition, the adversary cannot eavesdrop all the c' or c channels (between the database owner and the clouds, and between the user and the clouds). Hence, we do not impose private communication channels.

Under such an adversarial setting, we provide a guaranteed solution so that an adversary cannot learn the data or computations. It is important to mention that an adversary can break our protocols by colluding c' clouds, which is the threshold for which the secret sharing scheme is designed for.

Parameters for analysis. We analyze our privacy-preserving algorithms on the following parameters:

Communication cost: is the sum of all the bits that are required to transfer between a user and a cloud.

Computational cost: is the sum of all the bits over which a cloud or a user works.

Number of rounds: shows how many times a user communicates with a cloud for obtaining the results.

Table 10.1 summarizes all the results of this chapter and a comparison with the existing algorithms.

10.3 Creation and Distribution of Secret-Shares of a Relation

We consider an example of a relation, $Employee$, see Figure 10.2. A data owner creates secret-shares of this relation and sends to c clouds. In this section, we show how to create secret-shares of a value, following an approach given in [46].

A secure way for creating *secret-shares*. Assume that a database only contains English words. Since the English alphabet consists of 26 letters, each letter can be represented by a unary vector with 26 bits. Hence, the letter 'A' is represented as $(1_1, 0_2, 0_3, \ldots, 0_{26})$, where the subscript represents the position of the letter; since 'A' is the first letter, the

Algorithms	Communication cost	Computational cost		# rounds	Matching	Based on
		User	Cloud			
Count operation						
EPiC [26]	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	1	Online	E
Our solution 10.4	$\mathcal{O}(1)$	$\mathcal{O}(1)$	nw	1	Online	SS
Search and single tuple fetch operation						
Niv [32]	$\mathcal{O}(nmw)$	$\mathcal{O}(1)$	$\mathcal{O}(nmw)$	$log_2 n$	Online	SS
PRISM [27]	$\mathcal{O}((nm)^{\frac{1}{2}}w)$	$\mathcal{O}((nm)^{\frac{1}{2}}w)$	$\mathcal{O}(nmw)$	q		E
Our solution 10.5.1	$\mathcal{O}(mw)$	$\mathcal{O}(mw)$	$\mathcal{O}(mw)$	1	Online	SS
Search and multi-tuples fetch operation						
rPIR [77]	$\mathcal{O}(nm)$	$\mathcal{O}(1)$	$\mathcal{O}(nmw)$	1	No	SS
PIRMAP [88]	$\mathcal{O}(nmw)$	$\mathcal{O}(mw)$	$\mathcal{O}(nmw)$	1	No	E
Goldberg [81]	$\mathcal{O}(n+m)$	$\mathcal{O}(m)$	$\mathcal{O}(nm)$	2	Offline	SS
Emekci et al. [56]	$\mathcal{O}(\ell m)$	$\mathcal{O}(\ell m)$	$\mathcal{O}(n)$	2	Offline	vSS
Our solution: knowing addresses 10.5.2	$\mathcal{O}\big((log_\ell n + log_2 \ell)\ell\big)$	$\mathcal{O}\big((log_\ell n + log_2 \ell)\ell\big)$	$\mathcal{O}\big((log_\ell n + log_2 \ell)\ell nw\big)$	$\lfloor log_\ell n \rfloor + \lfloor log_2 \ell \rfloor + 1$	Online	SS
Our solution: fetching tuples 10.5.2	$\mathcal{O}((n+m)\ell w)$	$\mathcal{O}((n + \ell m)w)$	$\mathcal{O}(\ell nmw)$	1	Online	SS
Equijoin						
Our solution 10.6	$2nwk + 2k\ell^2 mw$	$2nw + 2k\ell^2 mw$	$2\ell^2 kmw$	$2k$	Online	SS

Notations: Online: perform string matching in the cloud. Offline: perform string matching at the user-side. E: encryption-decryption based. SS: Secret-sharing based. vSS: a variant of SS. n: # tuples, m: # attributes, ℓ: # occurrences of a pattern ($\ell \leq n$), w: bit-length of a pattern.

Table 10.1: Comparison of different algorithms with our algorithms.

Employee Id	First name	Last name	Date of birth	Salary	Department
E101	Adam	Smith	12/07/1975	1000	Sale
E102	John	Boro	10/30/1985	2000	Design
E103	Eve	Smith	05/07/1985	500	Sale
E104	John	Williams	04/04/1990	5000	Sale

Figure 10.2: A relation: *Employee*.

first value in the vector is one and others are zero. Similarly, 'B' is $(0_1, 1_2, 0_3, \ldots, 0_{26})$, 'J' is $(0_1, \ldots, 0_9, 1_{10}, 0_{11}, \ldots, 0_{26})$, and so on. The reason of using unary representation here is that it is very easy for verifying two identical letters. The expression $S = \sum_{i=0}^{r} u_i \times v_i$, compares two letters, where $(u_0, u_1, \cdots u_r)$ and $(v_0, v_1, \cdots, v_r)$ are two unary representations. It is clear that whenever any two letters are identical, S is equal to one; otherwise, S is equal to zero. Binary representation can also be accepted, but the comparison function is different from that used in the unary representation [49].

Now, when outsources a vector to the clouds, we use SSS and make secret-shares of every bit by selecting different polynomials of an identical degree; see Algorithm 11 in Appendix G.1. For example, we create secret-shares of the vector of 'A' $((1_1, 0_2, 0_3, \ldots, 0_{26}))$ by using 26 polynomials of an identical degree to create secret-shares of each bit, since the length of the vector is 26. Following that, we can create secret-shares for all the other letters and distribute them to different clouds.

Since we use SSS, a cloud cannot infer a secret. Moreover, it is important to emphasize that we use *different* polynomials for creating secret-shares of each letter; thereby multiple occurrences of a word in a database have different secret-shares. Therefore, a cloud is also unable to know the total number of occurrences of a word in the whole database. Following that, the two occurrences of the word `John` in our example, see Figure 10.2, have two different secret-shares.

Secret-shares of numeral values. We follow the similar approach for creating secret-shares of numeral values as used for alphabets. In particular, we create a unary vector of length 10 and put all the values 0 except only 1 according to the position of a number. For example, '1' becomes $(1_1, 0_2, \ldots, 0_{10})$. Then, we use SSS to make secret-shares of every bit in each vector by selecting different polynomials of an identical degree for each number, and send them to multiple clouds.

Aside. There is a challenge for creating secret-shares of a database; but once we did it, the rest of operations are relatively easier. Moreover, creating secret-shares of a database and its storage is less expensive than encrypting a database and its storage [97]. Also, it should be noted that standard techniques based on Berlekamp-Welch algorithm [110], where additional secret shares are used to encode the data can be directly applied here, enabling us to cope with a malicious adversary, with no change in the communication pattern.

In the next section, we will present four privacy-preserving algorithms for performing four fundamental operations on a database of the form of secret-share, as: count the occurrences of a pattern, fetch all the tuples containing a pattern, equijoin of two relations, and execution of range queries. All these algorithms are based on string matching of a value of a relation with a pattern, where the value and the pattern are of the form of secret-shares. The string matching operation on secret-shares will be done with the help of accumulating-automata (AA) [46]. All these algorithms execute operations obliviously in the cloud so that the cloud can never know which operations are executing on which tuples of a relation, while the user has to perform a simple operation to reconstruct the result. Throughout the section, we denote a pattern by p.

10.4 Count Query

We present a privacy-preserving algorithm for counting the number of occurrences of p in the cloud; see Algorithm 12 in Appendix G.2. We use our running example to count the number of people who have their first name as `John` in the relation *Employee*, see Figure 10.2. The algorithm is divided into two phases, as:

PHASE 1: Privacy-preserving counting in the clouds, Section 10.4.1

PHASE 2: Result reconstruction at the user-side, Section 10.4.2

In short, we apply a string matching algorithm, which is done using AA that compares each value of a relation with p. If a value and p match, it will result in 1; otherwise, we have 0. We apply the same algorithm on each value and collect the outputs. The sum of all the outputs provide the number of occurrences of p. Note that all the values of a relation, a pattern, and the result, *i.e.*, 0 or 1, are of the form of secret-share.

10.4.1 Counting a pattern

Counting a pattern, `John`, in secret-shares in different clouds. We explain how to count the occurrences of `John` in a relation of the form of secret-shares; see Algorithm 12 in Appendix.

Working at the user-side. Recall that the user creates unary vectors for each letter of p. In order to hide the vectors of p, the user creates secret-shares of each vector of p, as suggested in Section 10.3, sends them to the clouds. In our running example, a user creates four unary vectors for each letter of `John`, and then, creates secret-shares of each unary vector. In addition, the user writes a code of mappers for each cloud and also creates node values of the form of secret-shares.

Working in the cloud. Now, a cloud has three things, as: (*i*) a relation of the form secret-shares, (*ii*) a searching pattern of the form of secret-shares, and (*iii*) code of mappers with node values of the form of secret-shares.

The mapper creates an automaton, which performs a string matching operation, with $x + 1$ nodes[2] where x is the length of p and initializes values of these nodes. The first node is assigned a value one ($N_1 = 1$, N_i shows the value of node i), and all the other nodes are assigned values zero ($N_i = 0, i \neq 1$). The mapper reads each encoded word one-by-one and executes $x + 1$ steps for each word for finding new values of the nodes. At the end of the computation, the value of the node N_{x+1} shows the occurrences of p.

Example. In our running example, since we are searching a pattern of length four, a mapper creates an automaton of five nodes, assigns a node value one to the first node, and

[2]Note that $x + 1$ nodes are not machine nodes. These are parts of an automaton.

zero to the other nodes. The mapper reads the first name of employees one-by-one and executes five steps, given in Table 10.2 for each word.

STEP 1: $N_1 = 1, N_5^0 = 0$ STEP 2: $N_2^{(i)} = N_1 \times v_1$ STEP 3: $N_3^{(i)} = N_2^{(i)} \times v_2$ STEP 4: $N_4^{(i)} = N_3^{(i)} \times v_3$ STEP 5: $N_5^{(i)} = N_5^{(i-1)} + N_4^{(i)} \times v_4$
The notation $N_j^{(i)}$ shows that the node j is executing a step in iteration i. The final value of the node N_5, which is sent to the user, is the number of occurrences of the pattern. In our example, there are four tuples so that these five steps will be executed exactly four times.

Table 10.2: The steps executed by a mapper for counting `John`.

Explanation of the steps for counting `John` *in non-secret-shared data.* For the purpose of simplicity and understanding, we first show how to perform a string matching operation on the unary vectors, and then, we explain for the database of the form of secret-shares.

In the first iteration $i = 1$, the mapper reads the word 'Adam,' executes STEPs 1 and 2, and obtains the value of v_1 by multiplying the vector of 'A' with the vector of '`J`,' which results in $v_1 = 0$ and $N_2^{(1)} = 0$. After that, the mapper executes STEP 3 and obtains the value of v_2 by multiplying the vector of 'd' with the vector of '`o`,' which results in $v_2 = 0$, and hence, using the value of $N_2^{(1)} = 0$, $N_3^{(1)}$ will be 0. Then, the mapper executes STEPs 4 and 5 and obtains values of v_3 and v_4 by multiplying the vector of 'a' with the vector of '`h`,' which results in $v_3 = 0$ and $N_4^{(1)} = 0$, and respectively, by multiplying the vector of 'm' with the vector of '`n`,' which results in $v_4 = 0$ and $N_5^{(1)} = 0$. The value of $N_5^{(1)} = 0$ in the first iteration shows that 'Adam' and `John` are not identical.

Next, the mapper reads the word 'John' and executes the second iteration, $i = 2$. In STEP 2, multiplication of the vector of 'J' with the vector of '`J`' results in $v_1 = 1$ and $N_2^{(2)} = 1$. Similarly, the mapper executes STEPS 3, 4, and 5, and obtains, the values as: $v_2 = 1$ and $N_3^{(2)} = 1$, $v_3 = 1$ and $N_4^{(2)} = 1$, and $v_4 = 1$ and $N_5^{(2)} = 1$. Next, the mapper reads the word 'Eve,' executes all the STEPs, and results in $v_1 = 0$ and $N_2^{(3)} = 0$, $v_2 = 0$ and $N_3^{(3)} = 0$, $v_3 = 0$ and $N_4^{(3)} = 0$, $v_4 = 0$, and the value of $N_5^{(3)}$ will be 1, which shows that until now only one employee has `John` as a first name. The mapper reads the word 'John,' executes all the STEPs and eventually results in $N_5^{(4)} = 2$, which shows that two employees have `John` as their first names.

Explanation of the steps for counting `John` *in secret-shared data.* In order to count the number of occurrences of p, the mapper performs five steps, as mentioned above, for comparing `John` with each first name. At this time, the mapper is unable to know the value of the node N_5 in each iteration and sends the final value of N_5 to the user of form of a $\langle key, value \rangle$ pair, where a key is an identity of an input split over which the operation has performed, and the corresponding $value$ is the final value of the node N_5 of the form of secret-shares. The user collects $\langle key, value \rangle$ pairs from all the clouds or a sufficient number of clouds such that the secret can be generated using those shares.

10.4.2 Result reconstruction at the user-side

When we count the occurrences of p in the unary vectors, there is no need for result reconstruction at the user-side. The final value of the node N_{x+1} shows the number of occurrences of p, where x is the length of p. In our example, the final value of the node N_5 shows the total number of occurrences of `John` in the relation.

In case of secret-shares, however, we need to reconstruct the final value of the node N_{x+1}. The user has $\langle key, value \rangle$ pairs from all the clouds. All the values corresponding to a key are assigned to a reducer that performs the interpolation and provides the final value of the node N_{x+1}. If there are more than one reducer, then after the interpolation the sum of the final values shows the total number of occurrences of p.

Aside. If a user searches `John` in a database containing names like 'John' and 'Johnson,' then our algorithm will show two occurrences of `John`. However, it is a problem associated with string matching. In order to search a pattern precisely, we may use the terminating symbol for indicating the end of the pattern. In the above example, we can use "`John `", which is the searching pattern ending with a space, for obtaining the correct answer.

Theorem 10.1 (Cost for *count* operation) *The communication cost, the computational cost at a cloud, and the computational cost at the user-side for counting the occurrences of a pattern is at most $\mathcal{O}(1)$, at most nw, and at most $\mathcal{O}(1)$, respectively, where n is the number of tuples in a relation and w is the maximum bit length.*

The proof is given in Appendix G.2.

10.5 Search and Fetch Queries

In this section, we provide a privacy-preserving algorithm for fetching all the tuples containing p. The proposed algorithms first execute Algorithm 12 for counting the number of tuples containing p, and then, fetch all the tuples after obtaining their addresses. Specifically, we provide 2-phased algorithms, where:
PHASE 1: Finding addresses of tuples containing p
PHASE 2: Fetching all the tuples containing p

We will present Algorithm 13 for fetching a tuple when a relation has only one tuple containing p, in Section 10.5.1, and Algorithm 14 for fetching $\ell > 1$ tuples when a relation has ℓ tuples containing p, in Section 10.5.2. In both the algorithms, the user follows the similar approach for creating secret-shares and counting the occurrences of p, as described in Sections 10.3 and 10.4, respectively.

10.5.1 Unary occurrence of a pattern

When only one tuple contains p, there is no need to obtain the address of the tuple, and Algorithm 13 fetches the whole tuple in a privacy-preserving manner. Here, we explain how to fetch a single tuple containing p. Algorithm 13 works as follows:

Fetching the tuple. The user sends secret-shares of p. The cloud executes a map function on a specific attribute, and the map function matches p with i^{th} value of the attribute, as we did for the count operation; however, we do not add the output of the all tuple. Consequently, the map function results in either 0 or 1 of the form of secret-shares. Note that if p matches the i^{th} value of the attribute, then the result is 1. After that the map function multiplies the result (0 or 1) by all the m values of the i^{th} tuple. In this manner, the output of the map phase is a relation of n tuples and m attributes.

When the map function finishes over all the n tuples, it adds and sends all the secret-shares of each attribute, as: $S_1||S_2||\ldots||S_m$ to the user, where S_i is the sum of the secret-shares of i^{th} attribute (since after multiplication all the tuples contain zero of the form of secret-shares except one, addition operation over each attribute results in the desired row of the form of secret-shares). The user on receiving shares from all the clouds executes a reduce function that performs the interpolation and provides the desired tuple containing p.

Theorem 10.2 *The communication cost, the computational cost at a cloud, and the computational cost at the user-side for fetching a single tuple containing a pattern is at most $\mathcal{O}(mw)$, at most $\mathcal{O}(nmw)$, and at most $\mathcal{O}(mw)$, respectively, where a relation has n tuples and m attributes and w is the maximum bit length.*

The proof of the theorem is given in Appendix G.3.

10.5.2 Multiple occurrences of a pattern

When multiple tuples contain p, we cannot fetch all those tuples obliviously without obtaining their addresses. Therefore, we first need to perform a pattern search algorithm to obtain the addresses of all the tuples containing p, and then, fetch the tuples in a privacy-preserving manner. Throughout this section, we consider that ℓ tuples contain p.

In this section, we provide two algorithms for obtaining the addresses of tuples containing p. Both the algorithms have 2-phases, as:
PHASE 1: Finding the addresses of the desired ℓ tuples
PHASE 2: Fetching all the ℓ tuples

Tradeoff. When fetching multiple tuples containing p, there is a tradeoff between the number of communication rounds and the computational cost at the user-side, and this tradeoff will be clear after the description of the first and the second algorithm. In particular, the user performs a lot of computation when she wants to know the addresses of all the tuples containing p in one round. On the other hand, obtaining the addresses in multiple rounds requires that the cloud has to perform a heavy computation, while the user has to perform the interpolation.
Naive algorithm. A simple and naive algorithm requires only two rounds of communication between a user and the cloud for executing the two-phases, one round for each phase. However, the algorithm requires more workload at the user-side.

Finding addresses. The user sends p of the form of secret-shares to the clouds, and the cloud executes a map function that performs a string matching algorithm on secret-shares of each tuple, as we did to count the occurrences of `John` in Section 10.4.1. However, we do not accumulate occurrences, and hence, sends n values corresponding to each tuple. The user implements a reduce function that performs the interpolation and creates a vector, v, of length n, where i^{th} entity has value either 0 or 1, depending on the occurrence of p in the i^{th} tuple of the relation. As a disadvantage, the user has to work on all the tuples, but the user knows addresses of all the desired tuples in a single round.

Fetching tuples. The user creates a $\ell \times n$ matrix, M, and creates secret-shares of it, by following the approach suggested in Section 10.3. All the n columns of a row of the matrix M has 0 but 1 that is dependent on the addresses of the tuples containing p. For example, in the vector v, if the second position is 1, then we create a row of the matrix M where all the n columns have 0 but the second column has 1. After that, we use n polynomials of identical degree for making secret-shares of all the values and send them to clouds.

Recall that the cloud has a relation of n tuples and m attributes. A mapper in the cloud performs matrix multiplication by multiplying the matrix M with the relation and sends the results to the user. Recall that the matrix M has 0 and 1 of the form of secret-shares, so that the multiplication results in only the desired tuple and all the other tuples are eliminated. The user finally executes a reduce function that performs the interpolation and provides the desired ℓ tuples. A similar approach is also presented in [81].

Theorem 10.3 *After obtaining the addresses of the desired tuples containing a pattern, p, the communication cost, the computational cost at a cloud, and the computational cost at* *the user-side for fetching the desired tuples is at most $\mathcal{O}((n+m)\ell w)$, $\mathcal{O}(\ell nmw)$, and at most $\mathcal{O}((n+m\ell)w)$, respectively, where a relation has n tuples and m attributes, w is the maximum bit length, and ℓ is the number of tuples containing p.*

The proof of the theorem is given in Appendix G.4.

Tree-based algorithm. In order to decrease the computational load at the user-side, we propose a search-tree-based keyword search algorithm (Algorithm 14, pseudocode is given in Appendix G.4) that consists of two phases, as: finding the address of the desired ℓ tuples in multiple rounds, and then, fetching all the ℓ tuples in one more round.

Taking inspiration form Algorithm 13, we can also obtain the addresses (or line numbers) in a privacy-preserving manner, if only a single tuple contains p. Thus, for the case of finding addresses of ℓ tuples containing p, we divide the whole relation into certain blocks such that each block belongs to one of the following cases:

1. A block contains no occurrence of p, and hence, no fetch operation is needed.
2. A block contains one/multiple tuples but only a single tuple contains p.
3. A block contains h tuples, and all the h tuples contain p.
4. A block contains multiple tuples but fewer tuples contain p.

Finding addresses. We follow an idea of partitioning the database and counting the occurrences of p in the partitions, until each partition satisfies one of the above mentioned cases. Specifically, we initiate a sequence of Query & Answer (Q&A) rounds. In the first round of Q&A, we count occurrences of p in the whole database (or in an assigned input split to a mapper) and then partition the database into ℓ blocks, since we assumed that ℓ tuples contain p. In the second round, we again count occurrences of p in each block and focus on the blocks satisfying Case 4. There is no need to consider the blocks satisfying Case 2 or 3, since we can apply Algorithm 13 in both the cases. However, if the multiple tuples of a block in the second round contain p, *i.e.*, Case 4, we again partition such a block until it satisfies either Case 1, 2 or 3. After that, we can obtain the addresses of the related tuples using the method similar to Algorithm 13.

Fetching tuples. We use the approach described in the naive algorithm for fetching multiple tuples after obtaining the addresses of the tuples.

Example. Here, we give an example to illustrate the above approach. Let an input split consists of 9 tuples, see Figure 10.3, and the number of occurrence of p is two. When the user knows the number of occurrences, she starts Q&A rounds. In each Q&A round, a mapper partitions specific parts of the input split into two blocks, performs AA in each blocks, and sends results, which are occurrences of p in each block, of the form of secret-shares back to the user.

In this example, the user initiates the first Q&A round, and a mapper divides the input

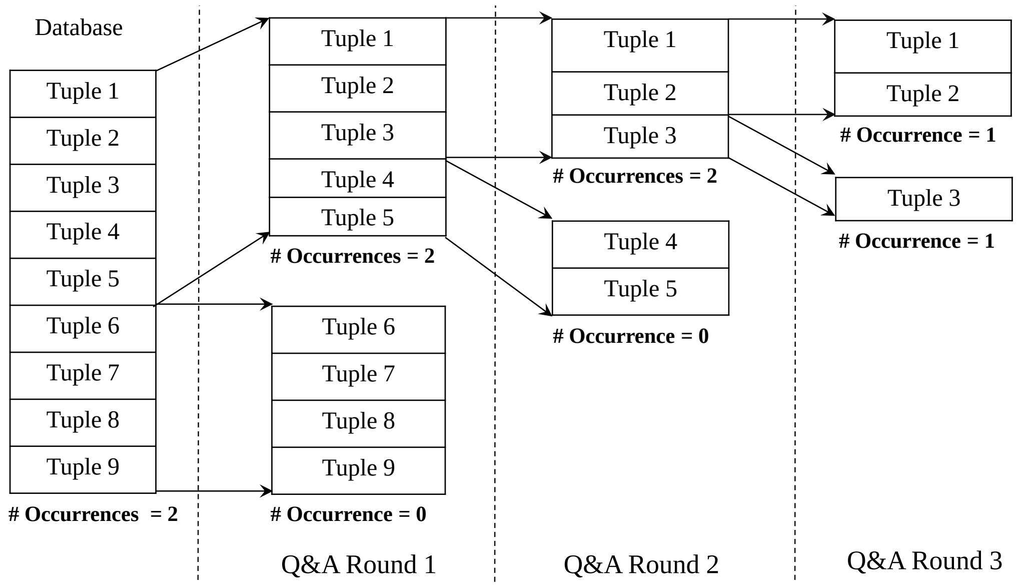


Figure 10.3: Example of Q&A rounds.

split into two parts. In each block, it counts the occurrences of p and sends the results to the user. The user executes a reducer that performs the interpolation. The user knows that the first and the second blocks contain two and zero tuples having p, respectively. The user divides the first block into two parts again in the second Q&A round. The mapper performs an identical operation as it does in the first round, and after three Q&A rounds, the user have all the two tuples having p.

Theorem 10.4 *The maximum number of rounds for obtaining addresses of tuples containing a pattern, p, using Algorithm 14 is $\lfloor log_\ell n \rfloor + \lfloor log_2 \ell \rfloor + 1$, and the communication cost for obtaining such addresses is at most $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell\big)$. The computational cost at a cloud and the computational cost at the user-side is at most $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell n w\big)$ and at most $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell\big)$, respectively, where a relation has n tuples and m attributes, ℓ is the number of tuples containing p, and w is the maximum bit length.*

The proof of the theorem is given in Appendix G.4.

Example. In figure 10.3, in order to fetch tuples containing p, the user needs three rounds, which are less than $\lfloor \log_2 9 \rfloor + \lfloor \log_2 2 \rfloor + 1 = 5$.

10.6 Equijoin

In this section, we show how to perform the equijoin in a privacy-preserving manner using MapReduce. Throughout this section, we consider two relations $X(A, B)$ and $Y(B, C)$ containing n tuples in each, where the joining attribute is B. A trivial way for performing the equijoin in a privacy-preserving manner, as follows: (*i*) fetch all the secret-shares of B-values from all the clouds and perform the interpolation, (*ii*) find tuples of both the relations that have an identical B-value and fetch all those tuples, (*iii*) perform the interpolation on the tuples, and (*iv*) perform a MapReduce job for joining the tuples at the user-side. However, in this approach the user has to perform the interpolation and MapReduce-based join.

In order to decrease the workload at the user-side, we propose two approaches so that the user has to perform only the interpolation on the output tuples of the join. The first approach assumes that the relations X and Y have at most one occurrence of B values in each, and the second approach does not hold any restriction on the occurrences of B-values, *i.e.*, a B-value can occur in multiple tuples of the relations.

10.6.1 A unique occurrence of the joining value

We use string matching operations (a variant of Algorithm 13) on secret-shares for performing the equijoin. The following steps are executed for performing the equijoin when a joining value occurs in at most one tuple of a relation, as:
1. In a cloud:
   a. A mapper reads i^{th} tuple $\langle a_i, b_i\rangle$ of the relation X and provides a pair of $\langle key, value\rangle$, where a key is an identity i and a $value$ is secret-shares of $\langle a_i, b_i\rangle$.
   b. A mapper reads j^{th} tuple $\langle b_j, c_j\rangle$ of the relation Y and provides n pairs of $\langle key, value\rangle$, where a key is an identity from 1 to n and a $value$ is secret-shares of $\langle b_j, c_j\rangle$.
   c. A reducer i is assigned $\langle i, [a_i, b_i]\rangle$, where $a_i, b_i \in X$, and all the tuples of the relation Y. The reducer performs string matching operations on the B values that result in 0 or 1 of the form of secret-share. Specifically, the reducer matches $b_i \in X$ with each $b_j \in Y$, and the resultant of the string matching operation (b_i and b_j) is multiplied by the tuple $\langle b_j, c_j\rangle$. After performing the string matching operation on all the B-values of the relation Y, the reducer adds all the secret-shares of the attributes B and C. The sum of the B-values is multiplied by the tuple $\langle a_i, b_i\rangle$ and the sum of the C-values is appended to this tuple. Thus, a new tuple is obtained as $\langle a', b', c'\rangle$.
2. The user fetches all the outputs of reducers from all the clouds, performs the interpolation, and obtains the outputs of the equijoin.

Example. We consider two relations X and Y, see Figure 10.4. Consider that all values are of the form of secret-shares. Mappers in the cloud read the tuples $\langle a_1, b_1\rangle$, $\langle a_2, b_2\rangle$, and $\langle a_3, b_3\rangle$ and provide $\langle 1, [a_1, b_1]\rangle$, $\langle 2, [a_2, b_2]\rangle$, and $\langle 3, [a_3, b_3]\rangle$, respectively. The mapper reads the tuple $\langle b_1, c_1\rangle$ and provides $\langle 1, [b_1, c_1]\rangle$, $\langle 2, [b_1, c_1]\rangle$, and $\langle 3, [b_1, c_1]\rangle$. A similar operation is also carried out on the tuples $\langle b_2, c_2\rangle$ and $\langle b_4, c_4\rangle$.

A reducer corresponding to key 1 matches b_1 of X with b_1 of Y that results in 1, then b_1

A	B
a_1	b_1
a_2	b_2
a_3	b_3

B	C
b_1	c_1
b_2	c_2
b_4	c_4

Figure 10.4: Two relations $X(A, B)$ and $Y(B, C)$.

A	B
a_1	b_1
a_2	b_1
a_3	b_2

B	C
b_1	c_1
b_1	c_2
b_3	c_3

Figure 10.5: Two relations $X(A, B)$ and $Y(B, C)$ with multi occurrences of a B-value.

of X with b_2 of Y that results in 0, and b_1 of X with b_4 of Y that results in 0. Remember 0 and 1 are of the form of secret-shares. Now, the reducer multiplies the three values (1,0,0) of the form of secret-shares by the tuples $\langle b_1, c_1\rangle$, $\langle b_2, c_2\rangle$, and $\langle b_4, c_4\rangle$, respectively. After that the reducer adds all the B-values and the C-values. Note that we will obtain now only the desired tuple, *i.e.*, $\langle b_1, c_1\rangle$. The reducer multiplies the sum of all the B-values by the tuple $\langle a_1, b_1\rangle$ appended with the sum of all the C-values. The same operation is carried out on other B-values of the relation X. When the user performs the interpolation on the outputs of all the clouds, only the desired output tuples of the equijoin are obtained, and all the other tuples, for example $\langle a_3, b_3\rangle$, hold value zero. In this manner, the user performs the equijoin in a privacy-preserving manner without knowing undesired tuples.

Aside. We assume that the all the A, B, and C values of the relations do not contain zero.

Theorem 10.5 *The communication cost, the computational cost at a cloud, and the computational cost at the user-side for performing the equijoin of two relations X and Y, where a joining value can occur at most one time in a relation, is at most $\mathcal{O}(nmw)$, at most $\mathcal{O}(n^2mw)$, and at most $\mathcal{O}(nmw)$, respectively, where a relation has n tuples and m attributes and w is the maximum bit length.*

The proof of the theorem is given in Appendix G.5.

10.6.2 Multiple occurrences of the joining value

We present an algorithm for performing the equijoin when many tuples of a relation have an identical joining value. Consider two relations $X(A, B)$ and $Y(B, C)$, see Figure 10.5.

Note that if we follow the previous approach, Section 10.6.1, then we take b_1 of X, multiply b_1 by all three B-values of Y, and add all secret-shares. However, after addition, we cannot distinguish two occurrences of b_1 in Y. Hence, we present a new algorithm and system settings for this type of the equijoin.

New System Setting. We need a new system setting only for the equijoin when a joining value occurs many times in a relation, see Figure 10.6. Recall that in the system setting mentioned in Section 10.2, we use c non-communicating clouds to store secret-shares of a

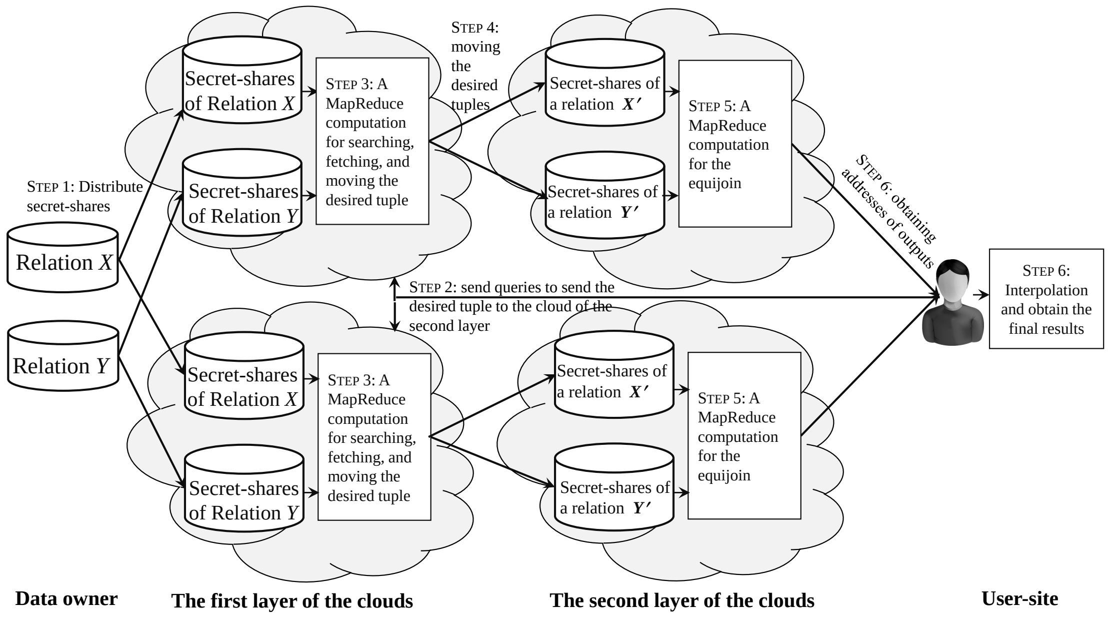

STEP 1: Distribute secret-shares
Relation X
Relation Y
Secret-shares of Relation X
Secret-shares of Relation Y
STEP 3: A MapReduce computation for searching, fetching, and moving the desired tuple
STEP 4: moving the desired tuples
Secret-shares of a relation X′
Secret-shares of a relation Y′
STEP 5: A MapReduce computation for the equijoin
STEP 6: obtaining addresses of outputs
STEP 2: send queries to send the desired tuple to the cloud of the second layer
STEP 6: Interpolation and obtain the final results
Data owner
The first layer of the clouds
The second layer of the clouds
User-site

Figure 10.6: The new system architecture for the equijoin.

relation. Here, we introduce one more layer of the clouds. The clouds within a layer are not allowed to communicate; however, the clouds of the first layer and the second layer can communicate, see STEP 4 of Figure 10.6.

A data owner outsources her databases, *i.e.*, the relations X and Y, of the form of secret-shares to c (non-communicating) clouds of the first layer only once; see STEP 1 in Figure 10.6. The user sends a query of the form of secret-shares to all c clouds of the first layer to find the desired tuples and send them to the clouds of the second layer; see STEP 2 in Figure 10.6. The clouds of the first layer execute the multi-tuple fetch algorithm (presented in Section 10.5.2) and send the desired tuples to the clouds of the second layer; see STEPs 3 and 4 in Figure 10.6. The cloud of the second layer creates two relations from the selected tuples of X and Y, and performs the join operation on secret-shares; see STEP 5 in Figure 10.6. Finally, the user fetches the outputs from the clouds of the second layer and performs the interpolation for obtaining secret-values; see STEP 6 in Figure 10.1. We will explain all these steps with the help of an example shortly.

The Approach. The approach consists of the following three steps, where the second step that perform the equijoin is executed in the clouds, as follows:

1. The user fetches all the B-values of the relations X and Y and performs the interpolation. After the interpolation, the user knows which B-values are identical in both relations and in which tuples they are.

2. For each B-value (say, b_i) that is in both relations:

a. The user requests the clouds of the first layer to send all the tuples containing b_i to a cloud of the second layer. This operation is done using the naive algorithm for fetching multiple tuples, refer to Section 10.5.2.

b. On receiving tuples containing the joining value b_i from the clouds of the first layer, the clouds in the second layer create two new relations corresponding to the tuples of X and Y. Then, the clouds in the second layer execute a MapReduce job that concatenates a tuple of the first relation to all the tuples of the second relation and provides the output of the equijoin, since the two new relations have only one identical B-value.

3. The user fetches all the outputs tuples from the second layer of the clouds and performs the interpolation.

Example. For the relations X and Y, see Figure 10.5, the user fetches all the B-values of both the relations and performs the interpolation. After the interpolation, the user knows that the joining value b_1 appears in the first tuple and second tuple of both the relations. The user follows the naive algorithm for fetching multiple tuples containing b_1 (refer to Section 10.5.2) and asks the clouds of the first layer to send these four tuples to the clouds of the second layer.

The cloud of the second layer creates two new relations X' containing $\langle a_1, b_1\rangle$ and $\langle a_2, b_1\rangle$, and Y' containing $\langle b_1, c_1\rangle$ and $\langle b_1, c_2\rangle$. A mapper reads a tuple and provides $\langle key, value\rangle$ pairs, where a key is an identity and a $value$ is the tuple. A reducer holds all the tuples of both the relations X' and Y' and joins (or concatenates) the first and the second tuples of X' to both the tuples of Y'. Finally, the user fetches the output and executes the interpolation.

Theorem 10.6 *The number of rounds, the communication cost, the computational cost at a cloud, and the computational cost at the user-side for performing the equijoin of two relations X and Y, where a joining value can occur in multiple tuples of a relation, is at most $\mathcal{O}(2k)$, at most $\mathcal{O}(2nwk + 2k\ell^2mw)$, at most $\mathcal{O}(\ell^2kmw)$, and at most $\mathcal{O}(2nw + 2k\ell^2mw)$, respectively, where a relation has n tuples and m attributes, k is the number of identical values of the joining attribute in the relations, ℓ is the maximum number of occurrences of a joining value, and w is the maximum bit length.*

The proof of the theorem is given in Appendix G.5.

10.7 Range Query

A range query finds, for example, all the employees whose salaries are between \$1000 and \$2000. We propose an approach for performing privacy-preserving range queries based on 2's complement subtraction. A number, say x, belongs in a range, say $[a, b]$, if $sign(x-a) =$

0 and $sign(x-b)=0$, where $sign(x-a)$ and $sign(b-x)$ denote the sign bits of $x-a$ and $x-b$, respectively, after 2's complement based subtraction.

Recall that in Section 10.3, we proposed an approach for creating secret-shares of a number, x, using unary representation that provides a vector, where all the values are 0 except only 1 according to the position of the number. The approach works well to count the occurrences of x and fetch all the tuples having x. However, on this vector, we cannot perform subtraction operation. Hence, in order to execute range queries, we represent a number using binary-representation, which results in a vector of length, say l. Then, we use SSS to make secret-shares of every bit in the vector by selecting l different polynomials of an identical degree for each bit position.

The approach. The idea of finding whether a number, x, belongs to the range, $[a,b]$, is based on 2's complement subtraction. In [49], the authors provided an algorithm for subtracting secret-shares using 2's complement. However, we will provide a simple 2's complement based subtraction algorithm for secret-shares, see Algorithm 15. A mapper checks the sign bits after subtraction for deciding the number whether it is in the range or not, as follows:

$$\begin{array}{ll}\text{If } x \in [a,b], & sign(x-a)=0, sign(b-x)=0 \\ \text{If } x < a, & sign(x-a)=1, sign(b-x)=0 \\ \text{If } x > b, & sign(x-a)=0, sign(b-x)=1\end{array} \tag{10.1}$$

After checking each number, we can use one of the following approaches:

1. *A simple solution.* The mapper sends the sign bit's values of the form of secret-shares to the user for each tuple. The user then implements a reduce function that performs the interpolation and creates an array of length n. If the number x in i^{th} tuples belongs in the range $[a,b]$, then the i^{th} position in the array is one. Otherwise, the i^{th} position in the array is zero. Finally, the user fetches all the tuples having value 1 in the array using the naive algorithm for fetching multiple tuples, see Section 10.5.2.
2. It keeps the count of all the numbers that belong in the range and sends the count to the user that interpolates them. After knowing how many numbers are in the range, the user can implement Algorithm 13 or 14 for fetching the desired tuples; see Algorithm 16. However, in this manner, we have to check the numbers whether they are in the range or not at the time of fetching the tuple. In this approach, we use many rounds for fetching the desired tuples; however, at the user-side, the computational cost decreases.

Pseudocodes of 2's complement based subtraction of secret-sharing (Algorithm 15) and of privacy-preserving range query (Algorithm 16) are given in Appendix G.6.

Chapter 11

Conclusion and Future Work

The replication of data and computing protocols provides a way to deal with faulty behavior of the system. However, replication comes with an extra cost of communication and computations. In this thesis, we figured out how the replication of data and computing protocols dominates the system design. Specifically, we investigated the impact of replications in (*i*) a self-stabilizing end-to-end communication algorithm, (*ii*) the model design for MapReduce, (*iii*) an evaluation of MapReduce algorithms for interval join of overlapping intervals and computing marginals of a data cube, and (*iv*) privacy-preserving MapReduce computations.

In Chapter 3, we proposed self-stabilizing end-to-end data communication algorithms for bounded capacity and duplicating dynamic networks. The proposed algorithm inculcates error correction methods for the delivery of messages to their destinations without omission, duplication, or reordering. We considered two nodes, one as the Sender and the other as the Receiver. In many cases, however, two communicating nodes may act both as senders and receivers simultaneously. In such situations, acknowledgment piggybacking may reduce the overhead needed to cope with the capacity irrelevant packets that exist in each direction, from the Sender to the Receiver and the reverse.

In Chapter 4, two new important practical aspects in the context of MapReduce, namely different-sized inputs and the reducer capacity, were introduced. The capacity of a reducer is defined in terms of the reducer's memory size. All reducers have an identical capacity, and any reducer cannot hold inputs whose sizes are more than the reducer capacity. We demonstrated the importance of the reducer capacity by considering two common mapping schema problems of MapReduce: *A2A mapping schema problem*, where every two inputs are required to be assigned to at least one common reducer, and *X2Y mapping schema problem*, where every two inputs, the first input from a set X and the second input from a set Y are required to be assigned to at least one common reducer. Unfortunately, it turned out that finding solutions to the *A2A* and the *X2Y* mapping schema problems using the minimum number of reducers is not possible in polynomial time. Chapter 5 provides near optimal approximation algorithms for the *A2A* and the *X2Y* mapping schema problems. The algorithms are based on a bin-packing algorithm, a pseudo-polynomial bin-packing algorithm, and the selection of a prime number.

In Chapter 6, we investigated impacts of different localities of data and mappers-reducers on a MapReduce computation. We found that it is not necessary to send the whole data to the location of computation if all the inputs do not participate in the final output. Thus, we proposed a new algorithmic technique for MapReduce algorithms, called Meta-MapReduce. Meta-MapReduce decreases a large amount of data to be transferred across clouds by transferring metadata, which is exponentially smaller, for a data field rather than the field itself. Meta-MapReduce processes metadata at the map phase and the reduce phase. We demonstrated the impact of Meta-MapReduce for solving problems of equijoin, skewjoin, and multi-rounds jobs. Also, we suggested a way to incorporate Meta-MapReduce to process geographically distributed data.

In Chapter 7, we focused on the problem of finding pairs of overlapping intervals. In the general case, we want to take the approach where we partition intervals with respect to their length in "large," "medium," and "small." Then, we observed that a preferred way to assign "large" intervals to only their starting- and ending-points overlapping partitions, and small intervals to all overlapping partitions. However, quantifying "large," "medium," and "small" is rather complicated. Thus, we considered simpler cases and analyzed them first, such as when all intervals have an identical length and when one set of intervals has mostly "large" intervals and the other set has mostly "small" intervals.

In Chapter 8, we studied the problem of computing marginals of a data cube in a single-round MapReduce job. We provided lower bounds and several algorithms for assigning inputs to reducers so that each reducer can compute many marginals of a fixed order. In the scope, this was considered to be the problem of "covering" sets of a fixed size ("marginals") by a small number of larger sets that contain them ("handles").

In Chapter 9, we discussed the security and privacy challenges and requirements in MapReduce. We considered four types of adversarial models, namely honest-but-curious, malicious, knowledgeable, and network and nodes access adversaries, and showed how they can impact a MapReduce computation.

In Chapter 10, we provided a privacy-preserving data and computation outsourcing technique for MapReduce computations. Specifically, we proposed a new information-theoretically secure data and computation outsourcing technique. By the proposed techniques, users can execute computations in the public cloud without the need of the database owner, and the cloud cannot learn the database and the computations. We demonstrated the usefulness of the technique with the help of count, search and fetch, equijoin, and range quires. We also compared our technique with existing algorithms and found that our algorithms provide perfect privacy protection without introducing computation and communication overhead.

Future directions. This thesis work is composed of theoretical models in distributed systems and MapReduce. The future plan is to enhance these models and algorithms for a variety of applications. Specifically, most intuitive future directions are listed as below:

- *Data streaming*. Several applications produce a huge amount of data at a time, while several other applications produce a small amount of data continuously; for example, data obtained from sensors or Twitter. In this work, we did not deal with the reception of continuous data at the computation site. Thus, we will attempt to enhance our approximation algorithms, Meta-MapReduce, and interval joins' algoritms, thereby the algorithms will be able to handle data streaming.
- *A model for executing dynamic programming on MapReduce.* Until now there are systems, *e.g.*, Spark, modern Hadoop, that perform well in many aspects. However, the execution of dynamic programming is still untouched in MapReduce. We aim to implement dynamic programming on MapReduce as well. In this case, some natural questions have to be answered, as: (*i*) how to store temporary data; (*ii*) where to store temporary data; (*iii*) for how many iterations to retain temporary data; and (*iv*) what types of operations can be performed on temporary data.
- *Security and privacy models.* Though MapReduce provides efficient large-scale data processing in the public cloud and we have provided a privacy-preserving technique for MapReduce-based computations, the security and privacy issues in MapReduce must be explored further. The future direction in this field has several milestones, as: (*i*) incorporating advanced authorization policies (*e.g.*, role-based or attribute-based access control policy) in MapReduce frameworks; (*ii*) including a trust infrastructure, *e.g.*, trust in the hardware, codes, cloud providers, virtual machines, and file systems (security of MapReduce); and (*iii*) execute a MapReduce computation in geo-distributed locations, while preserving the privacy of data and computations.
- *Replication aspects in the future generation of mobile cellular systems.* We developed a self-stabilizing end-to-end communication algorithm; however, the number of data packets is considerably large in order to achieve the goal. Hence, the algorithm may not perform well in a real-time and life-critical system. We have reviewed some existing solutions to the fifth generation (5G) mobile cellular networks [95] that are supposed to meet an extremely low latency. In future, we will attempt to investigate the role of replication in 5G networks, especially in cloud-based radio access networks and software-defined networks, and the integration of self-stabilizing communication algorithms in 5G networks.

Bibliography

[1] Apache Hadoop. Available at: `http://hadoop.apache.org/`.

[2] Apache YARN. Available at: `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/index.html`.

[3] A. Abelló, J. Ferrarons, and O. Romero. Building cubes with MapReduce. In *DOLAP 2011, ACM 14th International Workshop on Data Warehousing and OLAP, Glasgow, United Kingdom, October 28, 2011, Proceedings*, pages 17–24, 2011.

[4] Y. Afek and G. M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.

[5] F. Afrati, S. Dolev, E. Korach, S. Sharma, and J. D. Ullman. Brief announcement: Assignment of different-sized inputs in MapReduce. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 536–537, 2014.

[6] F. Afrati, S. Dolev, S. Sharma, and J. D. Ullman. Brief-announcement: Meta-MapReduce: A technique for reducing communication in MapReduce computations. In *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*, pages 272–275, 2015.

[7] F. Afrati and et al. Assignment problems of different-sized inputs in MapReduce. Technical Report 14-05, Department of Computer Science, Ben-Gurion University of the Negev, 2014.

[8] F. N. Afrati, S. Dolev, E. Korach, S. Sharma, and J. D. Ullman. Assignment of different-sized inputs in MapReduce. In *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015.*, pages 28–37, 2015.

[9] F. N. Afrati, S. Dolev, E. Korach, S. Sharma, and J. D. Ullman. Assignment problems of different-sized inputs in MapReduce. *ACM Transactions on Knowledge Discovery from Data*, 2016. Accepted.

[10] F. N. Afrati, S. Dolev, S. Sharma, and J. D. Ullman. Bounds for overlapping interval join on MapReduce. In *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint*

Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015., pages 3–6, 2015.

[11] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using Map-Reduce. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 62–73, 2013.

[12] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman. Parallel skyline queries. In *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, pages 274–284, 2012.

[13] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, and J. D. Ullman. Fuzzy joins using MapReduce. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 498–509, 2012.

[14] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Vision paper: Towards an understanding of the limits of Map-Reduce computation. *CoRR*, abs/1204.1754, 2012.

[15] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a Map-Reduce computation. *PVLDB*, 6(4):277–288, 2013.

[16] F. N. Afrati, S. Sharma, J. D. Ullman, and J. R. Ullman. Computing marginals using MapReduce. *CoRR*, abs/1509.08855, 2015.

[17] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a Map-Reduce environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011.

[18] F. N. Afrati and J. D. Ullman. Matching bounds for the all-pairs MapReduce problem. In *17th International Database Engineering & Applications Symposium, IDEAS '13, Barcelona, Spain - October 09 - 11, 2013*, pages 3–4, 2013.

[19] A. V. Aho and J. D. Ullman. *Foundations of Computer Science: C Edition*. W. H. Freeman, 1995.

[20] I. Anderson. *Combinatorial designs and tournaments*. Number 6. Oxford University Press, 1997.

[21] G. Anthes. Security in the cloud. *Commun. ACM*, 53(11):16–18, 2010.

[22] D. Applegate, E. M. Rains, and N. J. A. Sloane. On Asymmetric Coverings and Covering Numbers. *Journal on Combinatorial Designs*, 11:2003, 2003.

[23] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 268–277, 1991.

[24] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 131–140, New York, NY, USA, 2007. ACM.

[25] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A

comparison of join algorithms for log processing in mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 975–986, 2010.

[26] E. Blass, G. Noubir, and T. V. Huu. EPiC: efficient privacy-preserving counting for MapReduce, 2012.

[27] E. Blass, R. D. Pietro, R. Molva, and M. Önen. PRISM - privacy-preserving search in MapReduce. In *Privacy Enhancing Technologies - 12th International Symposium, PETS 2012, Vigo, Spain, July 11-13, 2012. Proceedings*, pages 180–200, 2012.

[28] B. Bollabas. *Combinatorics: set systems, hypergraphs, families of vectors, and combinatorial probability*. Cambridge University Press, 1986.

[29] A. Bui, A. K. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing PIF in tree networks. In *1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings*, pages 78–85, 1999.

[30] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):222–233, 2014.

[31] B. Chawda, H. Gupta, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania. Processing interval joins on Map-Reduce. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 463–474, 2014.

[32] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. *IACR Cryptology ePrint Archive*, 1998:3, 1998.

[33] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for NP-hard problems. chapter Approximation algorithms for bin packing: a survey, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[34] J. N. Cooper, R. B. Ellis, and A. B. Kahng. Asymmetric Binary Covering Codes. *Journal on Combinatorial Theory, Series A*, 100(2):232–249, 2002.

[35] R. M. Corless and N. Fillion. A graduate introduction to numerical methods. *AMC*, 10:12, 2013.

[36] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design (International Computer Science)*. Addison-Wesley Longman, Amsterdam, 4th rev. ed. edition, 2005.

[37] A. Cournier, S. Dubois, and V. Villain. A snap-stabilizing point-to-point communication protocol in message-switched networks. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–11, 2009.

[38] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters.

In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004.

[39] P. Derbeko, S. Dolev, E. Gudes, and S. Sharma. Security and privacy aspects in MapReduce on clouds: A survey. *Computer Science Review*, 20:1–28, 2016.

[40] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[41] L. Ding, G. Wang, J. Xin, X. Wang, S. Huang, and R. Zhang. ComMapReduce: an improvement of MapReduce with lightweight communication mechanisms. *Data & Knowledge Engineering*, 88:224–247, 2013.

[42] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[43] S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil. Stabilizing data-link over non-fifo channels with optimal fault-resilience. *Inf. Process. Lett.*, 111(18):912–920, 2011.

[44] S. Dolev, P. Florissi, E. Gudes, S. Sharma, and I. Singer. A survey on geographically distributed data processing using MapReduce. Dept. of Computer Science, Ben-Gurion University, Israel.

[45] S. Dolev, J. A. Garay, N. Gilboa, and V. Kolesnikov. Brief announcement: swarming secrets. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 231–232, 2010.

[46] S. Dolev, N. Gilboa, and X. Li. Accumulating automata and cascaded equations automata for communicationless information theoretically secure multi-party computation: Extended abstract. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, pages 21–29, 2015.

[47] S. Dolev, A. Hanemann, E. M. Schiller, and S. Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks - (extended abstract). In *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012, Toronto, Canada, October 1-4, 2012. Proceedings*, pages 133–147, 2012.

[48] S. Dolev, A. Israeli, and S. Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM J. Comput.*, 26(1):273–290, 1997.

[49] S. Dolev and Y. Li. Secret shared random access machine. In *Algorithmic Aspects of Cloud Computing - First International Workshop, ALGOCLOUD 2015, Patras, Greece, September 14-15, 2015. Revised Selected Papers*, pages 19–34, 2015.

[50] S. Dolev, Y. Li, and S. Sharma. Private and secure secret shared MapReduce - (extended abstract). In *Data and Applications Security and Privacy XXX - 30th Annual IFIP WG 11.3 Conference, DBSec 2016, Trento, Italy, July 18-20, 2016. Proceedings*,

pages 151–160, 2016.

[51] S. Dolev, E. Schiller, and J. L. Welch. Random walk for self-stabilitzing group communication in ad hoc networks. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, page 259, 2002.

[52] S. Dolev, E. Schiller, and J. L. Welch. Random walk for self-stabilizing group communication in ad-hoc networks. In *21st Symposium on Reliable Distributed Systems (SRDS 2002), 13-16 October 2002, Osaka, Japan*, pages 70–79, 2002.

[53] S. Dolev, E. Schiller, and J. L. Welch. Random walk for self-stabilizing group communication in ad hoc networks. *IEEE Trans. Mob. Comput.*, 5(7):893–905, 2006.

[54] S. Dolev and J. L. Welch. Crash resilient communication in dynamic networks. *IEEE Trans. Computers*, 46(1):14–26, 1997.

[55] F. Emekçi, D. Agrawal, A. El Abbadi, and A. Gulbeden. Privacy preserving query processing using third parties. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 27, 2006.

[56] F. Emekçi, A. Metwally, D. Agrawal, and A. El Abbadi. Dividing secrets to secure data outsourcing. *Inf. Sci.*, 263:198–210, 2014.

[57] B. Fish, J. Kun, Á. D. Lelkes, L. Reyzin, and G. Turán. On the computational complexity of MapReduce. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 1–15, 2015.

[58] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[59] S. Ghosh. *Distributed Systems: An Algorithmic Approach*. Chapman & Hall/CRC Computer & Information Science Series. Taylor & Francis, 2010.

[60] A. Goel and K. Munagala. Complexity measures for Map-Reduce, and comparison to parallel computing. *CoRR*, abs/1211.6526, 2012.

[61] M. T. Goodrich. Simulating parallel algorithms in the MapReduce framework with applications to parallel computational geometry. *CoRR*, abs/1004.4708, 2010.

[62] M. G. Gouda and N. J. Multari. Stabilizing communication protocols. *IEEE Trans. Computers*, 40(4):448–458, 1991.

[63] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 152–159, 1996.

[64] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by,

cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[65] H. Gupta and B. Chawda. ϵ-controlled-replicate: An improvedcontrolled-replicate algorithm for multi-way spatial join processing on Map-Reduce. In *Web Information Systems Engineering - WISE 2014 - 15th International Conference, Thessaloniki, Greece, October 12-14, 2014, Proceedings, Part II*, pages 278–293, 2014.

[66] H. Gupta, B. Chawda, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania. Processing multi-way spatial joins on Map-Reduce. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 113–124, 2013.

[67] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 205–216, 1996.

[68] S. Hoory, N. Linial, and A. Widgerson. Expander graphs and their applications. *Bulletin (New Series) of the AMS*, 43(4):439–561, 2006.

[69] C. Jayalath, J. J. Stephen, and P. Eugster. From the cloud to the atmosphere: Running MapReduce across data centers. *IEEE Trans. Computers*, 63(1):74–87, 2014.

[70] D. S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[71] D. R. Karger and J. Scott. Efficient algorithms for fixed-precision instances of bin packing and euclidean TSP. In *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques, 11th International Workshop, APPROX 2008, and 12th International Workshop, RANDOM 2008, Boston, MA, USA, August 25-27, 2008. Proceedings*, pages 104–117, 2008.

[72] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 938–948, 2010.

[73] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 85–94, 2011.

[74] S. Lee, J. Kim, Y. Moon, and W. Lee. Efficient distributed parallel top-down computation of ROLAP data cube using MapReduce. In *Data Warehousing and Knowledge Discovery - 14th International Conference, DaWaK 2012, Vienna, Austria, September 3-6, 2012. Proceedings*, pages 168–179. 2012.

[75] T. Lee, K. Kim, and H. Kim. Join processing using bloom filter in MapReduce. In *Research in Applied Computation Symposium, RACS '12, San Antonio, TX, USA,*

October 23-26, 2012, pages 100–105, 2012.

[76] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

[77] L. Li, M. Militzer, and A. Datta. rPIR: Ramp secret sharing based communication efficient private information retrieval. *IACR Cryptology ePrint Archive*, 2014:44, 2014.

[78] J. Lin and C. Dyer. Data-intensive text processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.

[79] Y. Liu, X. Jiang, H. Chen, J. Ma, and X. Zhang. MapReduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. In *Advanced Parallel Processing Technologies, 8th International Symposium, APPT 2009, Rapperswil, Switzerland, August 24-25, 2009, Proceedings*, pages 341–355, 2009.

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

[81] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, pages 168–186, 2015.

[82] Y. Luo and B. Plale. Hierarchical MapReduce programming model and scheduling algorithms. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012*, pages 769–774, 2012.

[83] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings.*, pages 149–159, 1986.

[84] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146, 2010.

[85] P. Malhotra, P. Agarwal, and G. Shroff. Graph-parallel entity resolution using LSH & IMM. In *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014.*, pages 41–49, 2014.

[86] P. Malhotra, P. Agarwal, and G. Shroff. Graph-parallel entity resolution using LSH & IMM. In *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014.*, pages 41–49, 2014.

[87] G. S. Manku, A. Jain, and A. D. Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007,*

Banff, Alberta, Canada, May 8-12, 2007, pages 141–150, 2007.

[88] T. Mayberry, E. Blass, and A. H. Chan. PIRMAP: efficient private information retrieval for mapreduce. In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, pages 371–385, 2013.

[89] T. K. Moon. Error correction coding. *Mathematical Methods and Algorithms. Jhon Wiley and Son*, 2005.

[90] A. C. Murthy, V. K. Vavilapalli, D. Eadline, J. Niemiec, and J. Markham. *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Pearson Education, 2013.

[91] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Data cube materialization and mining over MapReduce. *IEEE Trans. Knowl. Data Eng.*, 24(10):1747–1759, 2012.

[92] A. Okcan and M. Riedewald. Processing theta-joins using MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 949–960, 2011.

[93] F. G. Olumofin and I. Goldberg. Privacy-preserving queries over relational databases. In *Privacy Enhancing Technologies, 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings*, pages 75–92, 2010.

[94] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: Locality-aware resource allocation for MapReduce in a cloud. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 58:1–58:11, 2011.

[95] N. Panwar, S. Sharma, and A. K. Singh. A survey on 5G: The next generation of mobile communication. *Physical Communication*, 18:64–84, 2016.

[96] J. Park, D. Lee, B. Kim, J. Huh, and S. Maeng. Locality-aware dynamic VM reconfiguration on MapReduce clouds. In *The 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC'12, Delft, Netherlands - June 18 - 22, 2012*, pages 27–36, 2012.

[97] T. B. Pedersen, Y. Saygın, and E. Savaş. Secret charing vs. encryption-based techniques for privacy preserving data mining. 2007.

[98] A. Pietracaprina, G. Pucci, M. Riondato, F. Silvestri, and E. Upfal. Space-round tradeoffs for MapReduce computations. In *International Conference on Supercomputing, ICS'12, Venice, Italy, June 25-29, 2012*, pages 235–244, 2012.

[99] M. R. Randazzo, M. Keeney, E. Kowalski, D. Cappelli, and A. Moore. Insider threat study: Illicit cyber activity in the banking and finance sector, 2005.

[100] K. Rohitkumar and S. Patil. Data cube materialization using MapReduce. *International Journal of Innovative Research in Computer and Communication*

Engineering, 11(2):6506–6511, 2014.

[101] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[102] H. Takabi, J. B. D. Joshi, and G. Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, 8(6):24–31, 2010.

[103] A. S. Tanenbaum. *Computer networks (4. ed.)*. Prentice Hall, 2002.

[104] F. Tauheed, T. Heinis, and A. Ailamaki. THERMAL-JOIN: A scalable spatial join for dynamic workloads. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 939–950, 2015.

[105] J. D. Ullman. Designing good MapReduce algorithms. *ACM Crossroads*, 19(1):30–34, 2012.

[106] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 495–506, 2010.

[107] B. Wang, H. Gui, M. Roantree, and M. F. O'Connor. Data cube computational model with Hadoop MapReduce. In *WEBIST 2014 - Proceedings of the 10th International Conference on Web Information Systems and Technologies, Volume 1, Barcelona, Spain, 3-5 April, 2014*, pages 193–199, 2014.

[108] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen. G-Hadoop: MapReduce across distributed data centers for data-intensive computing. *Future Generation Comp. Syst.*, 29(3):739–750, 2013.

[109] Z. Wang, Y. Chu, K. Tan, D. Agrawal, A. El Abbadi, and X. Xu. Scalable data cube analysis over big data. *CoRR*, abs/1311.5663, 2013.

[110] L. R. Welch and E. R. Berlekamp. Error correction for algebraic block codes, Dec. 30 1986. US Patent 4,633,470.

[111] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 131–140, 2008.

[112] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *INFOCOM 2010. 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 15-19 March 2010, San Diego, CA, USA*, pages 534–542, 2010.

[113] Z. Yu, C. Wang, C. D. Thomborson, J. Wang, S. Lian, and A. V. Vasilakos. Multimedia applications and security in MapReduce: Opportunities and challenges. *Concurrency and Computation: Practice and Experience*, 24(17):2083–2101, 2012.

[114] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.

[115] C. Zhang, F. Li, and J. Jestes. Efficient parallel kNN joins for large data in MapReduce. In *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 38–49, 2012.

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

[117] G. Zhou, Y. Zhu, and G. Wang. Cache conscious star-join in MapReduce environments. In *2nd International Workshop on Cloud Intelligence (colocated with VLDB 2013), Cloud-I '13, Riva del Garda, Trento, Italy, August 26, 2013*, pages 1:1–1:7, 2013.

[118] D. Zissis and D. Lekkas. Addressing cloud computing security issues. *Future Generation Comp. Syst.*, 28(3):583–592, 2012.

Appendix A

Pseudocodes of the Self-Stabilizing End-to-End Communication Algorithm

Algorithm 5: Self-Stabilizing End-to-End Algorithm (Sender p_s)

Local variables:

$AltIndex \in [0, 2]$: state the current alternating index value

ACK_set: at most $(capacity + 1)$ acknowledgment set, where items contain labels and last delivered alternating indexes, $\langle ldai, lbl\rangle$

Interfaces:

$Fetch(NumOfMessages)$ Fetches $NumOfMessages$ messages from the application and returns them in an array of size $NumOfMessages$ according to their original order

$Encode(Messages[])$ receives an array of messages of length ml each, M, and returns a message array of identical size M', where message $M'[i]$ is the encoded original $M[i]$, the final length of the returned $M'[i]$ is n and the code can bare $capacity$ mistakes

```
1 Function $packet\_set()$ begin
2     foreach $(i, j) \in [1, n] \times [1, pl]$ do let $data[i].bit[j] = messages[j].bit[i]$;
3     return $\{\langle AltIndex, i, data[i]\rangle\}_{i\in[1,n]}$
4 Do forever begin
5     if $(\{AltIndex\} \times [1, capacity + 1]) \subseteq ACK\_set$ then
          $(AltIndex, ACK\_set, messages) \leftarrow$
            $((AltIndex + 1) \bmod 3, \emptyset, Encode(Fetch(pl)))$
6     foreach $pckt \in packet\_set()$ do send $pckt$;
7 Upon receiving $ACK = \langle ldai, lbl\rangle$ begin
8     if $ldai = AltIndex \wedge lbl \in [1, capacity + 1]$ then
9         $ACK\_set \leftarrow ACK\_set \cup \{ACK\}$
```

Algorithm 6: Self-Stabilizing End-to-End Algorithm (Receiver p_r)

```
Local variables:
$LastDeliveredIndex \in [0,2]$: the alternating index value of the last delivered packets
$packet\_set$: packets, $\langle ai, lbl, dat \rangle$, received, where $lbl \in [1,n]$ and $dat$ is data of size
  $pl$ bits

Interfaces:
$Decode(Messages[])$ receives an array of encoded messages, $M'$, of length $n$ each,
  and returns an array of decoded messages of length $ml$, $M$, where $M[i]$ is the
  decoded $M'[i]$. The code is the same error correction coded by the Sender and can
  correct up to $capacity$ mistakes
$Deliver(messages[])$ receives an array of messages and delivers them to the
  application by the order in the array

Macro:
$index(ind) = \{\langle ind, *, * \rangle \in packet\_set\}$

1  Do forever begin
2      if $\{\langle ai, lbl \rangle : \langle ai, lbl, * \rangle \in packet\_set\} \not\subseteq$
         $\{[0,2] \setminus \{LastDeliveredIndex\}\} \times [1,n] \times \{*\} \lor$
         $(\exists \langle ai, lbl, dat \rangle \in packet\_set : \langle ai, lbl, * \rangle \in packet\_set \setminus \{\langle ai, lbl, dat \rangle\}) \lor$
         $(\exists pckt = \langle *, *, data \rangle \in packet\_set : |pckt.data| \neq pl) \lor 1 < |\{ AltIndex : n \leq$
         $|\{ \langle AltIndex, *, * \rangle \in packet\_set\}|\}|$ then $packet\_set \leftarrow \emptyset$;
3      if $\exists\, !\, ind : ind \neq LastDeliveredIndex \land n \leq |index(ind)|$ then
4          foreach $(i,j) \in [1,pl] \times [1,n]$ do
5              let $messages[i].bit[j] = data.bit[i] : \langle ind, j, data \rangle \in index(ind)$
6          $(packet\_set, LastDeliveredIndex) \leftarrow (\emptyset, ind)$
7          $Deliver(Decode(messages))$
8      foreach $i \in [1, capacity + 1]$ do send $\langle LastDeliveredIndex, i \rangle$;
9  Upon receiving $pckt = \langle ai, lbl, dat \rangle$ begin
10     if $\langle ai, lbl, * \rangle \notin packet\_set \land \langle ai, lbl \rangle \in (\{[0,2] \setminus \{LastDeliveredIndex\}\} \times$
         $[1,n]) \land |dat| = pl$ then
11         $packet\_set \leftarrow packet\_set \cup \{pckt\}$
```

A.1 Detailed Description of Algorithms 5 and 6

The pseudo-code in Algorithms 5 and 6 implements the proposed S^2E^2C algorithm from the sender-side, and respectively, receiver-side. The two nodes, p_s and p_r, are the Sender and the Receiver nodes respectively. The Sender algorithm consists of a do forever loop statement (lines 4 to 5 of the Sender algorithm), where the Sender, p_s, assures that all the data structures comprises only valid contents. Namely, p_s checks that the ACK_set_s holds packets with alternating index equal to the Sender's current $AltIndex_s$ and the labels are between 1 and $(capacity + 1)$.

In case any of these conditions is unfulfilled, the Sender resets its data structures (line 5 of the Sender algorithm). Subsequently, p_s triggers the ***Fetch*** and the ***Encode*** interfaces (line 5 of the Sender algorithm). Before sending the packets, p_s executes the $packet\_set()$ function (line 6 of the Sender algorithm).

The Sender algorithm, also, handles the reception of acknowledgments $ACK_s = \langle ldai, lbl\rangle$ (line 7 of the Sender algorithm). Each packet has a distinct label with respect m's message batch. If $ACK_s = \langle ldai, lbl\rangle$ has the value of $ldai$ (last delivered alternating index) equals to $AltIndex$ (line 8 of the Sender algorithm), the Sender p_s stores ACK_s in ACK_set_s (line 9 of the Sender algorithm). When p_s gets such (distinctly labeled) packets $(capacity + 1)$ times, p_s changes $AltIndex_s$, resets ACK_set_s, and calls $Fetch()$ and $Encode()$ interfaces (line 5 of the Sender algorithm).

The Receiver algorithm executes at the Receiver side, p_r. The Receiver p_r repeatedly tests $packet\_set_r$ (line 2 of the Receiver algorithm), and assures that: (*i*) $packet\_set_r$ holds packets with alternating index, $ai \in [0, 2]$, except $LastDeliveredIndex_r$, labels (lbl) between 1 and n and data of size pl, and (*ii*) $packet\_set_r$ holds at most one group of ai that has (distinctly labeled) n packets. When any of the aforementioned conditions do not hold, p_r assigns the empty set to $packet\_set_r$.

When p_r discovers that it has n distinct label packets of identical ai (line 3 of the Receiver algorithm), p_r decodes the payloads of the arriving packets (line 5 of the Receiver algorithm). Subsequent steps include the reset of $packet\_set_r$ and change of $LastDeliveredIndex_r$ to ai (line 6 of the Receiver algorithm). Next, p_r delivers the decoded message (line 7 of the Receiver algorithm). In addition, p_r repeatedly acknowledges p_s by $(capacity + 1)$ packets (line 8 of the Receiver algorithm).

Node p_r receives a packet $pckt_r = \langle ai, lbl, dat\rangle$, see line 9 (the Receiver algorithm). If $pckt_r$ has data (dat) of size pl bits, an alternating index (ai) in the range of 0 to 2, excluding the $LastDeliveredIndex$, and a label (lbl) in the range of 1 to n (line 10 of the Receiver algorithm), p_r puts $pckt_r$ in $packet\_set_r$ (line 11 of the Receiver algorithm).

A.2 Correctness of Algorithms 5 and 6

We define the set of legal executions, and how they implement the S^2E^2C task (Chapter 2), before demonstrating that the Sender and the Receiver algorithms implement that task (Theorems A.5 and A.6).

Given a system execution, R, and a pair, p_s and p_r, of sending and receiving nodes, the S^2E^2C task associates p_s's sending message sequence $s_R = im_0, im_1, im_2, \ldots, im_\ell, \ldots$, with p_r delivered message sequence $r_R = om_0, om_1, om_2, \ldots, om_{\ell'}, \ldots$; see Chapter 2. The Sender algorithm encodes batch of messages, im_ℓ, using an error correction method (Figure 3.2) into a packet sequence, I, that tolerates up to $capacity$ wrong packets (the Sender algorithm, line 5). The Receiver decodes messages, $om_{\ell'}$, from a packet sequence, O (Receiver algorithm, line 7), where every n consecutive packets may have up to $capacity$ packets that were received due to channel faults rather than p_s transmissions. Therefore, our definition of legal execution considers an unbounded suffix of input packets queue, $I = (im_x, im_{x+1}, \ldots)$, which p_s sends to p_r, and a k, such that the packet output suffix starts following the first $k-1$ packets, $O = (om_k, om_{k+1}, \ldots)$, is always a prefix of I. Furthermore, a new packet is included in O infinitely often.

A.2.1 Basic facts

Throughout this section, we refer to R as an execution of the Sender and the Receiver algorithms, where p_s executes the Sender algorithm and p_r executes the Receiver algorithm. Let a_{s_α} be the α^{th} time that the Sender is fetching a new message batch, i.e., executes line 5 (the Sender algorithm). Let a_{r_β} be the β^{th} time that the Receiver is delivering a message batch, i.e., executing line 7 (the Receiver algorithm). Theorem A.1 shows that R includes, infinitely often, the steps a_{s_α} and a_{r_β}.

Recall that an adversarial execution can prevent packet exchange between the Sender and receiver via (selective) packet omissions. Thus, demonstrate the liveness property for nice executions that do not include any omission step.

Theorem A.1 (Liveness) *For every nice execution R, there exists an R's prefix, R', that has $\mathcal{O}(n)$ asynchronous rounds, and it includes at least one a_{s_α} step and least one a_{r_β} step, where n is the packet word length.*

Proof. By line 6 (the Sender algorithm) and line 8 (the Receiver algorithm), node p_i sends packets infinitely often to node p_j, where $i \in \{s, r\}$ and $j \in \{s, r\} \setminus \{i\}$. Our system settings assume that when node p_i sends a packet infinitely often to p_j through the communication channel, node p_j receives that packet infinitely often. This implies that within $\mathcal{O}(n)$ asynchronous rounds, the Receiver, p_r, receives all the n packets in $packet\_set_s()$ and stores them all in $packet\_set_r$, cf. line 11, and thus the condition in line 3 (the Receiver algorithm) is satisfied. Therefore, R' includes at least one a_{r_β} step. Moreover, the same argument implies that within $\mathcal{O}(n)$ asynchronous rounds, the Sender, p_s, receives $(capacity + 1)$ acknowledgments from p_r and stores them in the set ACK_set_s, cf. line 9, and thus the condition in line 5 (the Sender algorithm) is satisfied, where $capacity < n$. Therefore, R' includes at least one a_{s_α} step. ■

Lemmas A.2, A.3 and A.4 are needed for the proof of Theorem A.5, and Theorem A.6.

Lemma A.2 *Let $c_{s_\alpha}(x)$ be the x^{th} configuration between a_{s_α} and $a_{s_{\alpha+1}}$ and $ACK_\alpha = \{ack_\alpha(\ell)\}_{\ell \in [1, capacity+1]}$ be a set of acknowledgment packets, where $ack_\alpha(\ell) = \langle s\_index_\alpha, \ell \rangle$.*

1. *For any given $\alpha > 0$, there is a single index value, $s\_index_\alpha \in [0, 2]$, such that for any $x > 0$, it holds that $AltIndex_s = s\_index_\alpha$ in $c_{s_\alpha}(x)$.*
2. *Between a_{s_α} and $a_{s_{\alpha+1}}$ there is at least one configuration c_{r_β}, in which $LastDeliveredIndex_r = s\_index_\alpha$.*
3. *Between a_{s_α} and $a_{s_{\alpha+1}}$, the Sender, p_s, receives from the channel from p_r to p_s, the entire set, ACK_α, of acknowledgment packets (each packet at least once), and between (the first) c_{r_β} and $a_{s_{\alpha+1}}$ the Receiver must send at least one $ack_\alpha(\ell) \in ACK_\alpha$ packet, which p_s receives, where c_{r_β} is defined in 2.*

Proof. We start by showing that s_index_α exists before showing that c_{r_β} exists and that p_s receives ack_α from p_r between a_{s_α} and $a_{s_{\alpha+1}}$.

The value of $AltIndex_s = s\_index_\alpha$ is only changed in line 5 (the Sender algorithm). By the definition of a_{s_α}, line 5 is not executed by any step between a_{s_α} and $a_{s_{\alpha+1}}$. Therefore, for any given α, there is a single index value, $s\_index_\alpha \in [0, 2]$, such that for any $x > 0$, it holds that $AltIndex_s = s\_index_\alpha$ in $c_{s_\alpha}(x)$.

We show that c_{r_β} exists by showing that, between a_{s_α} and $a_{s_{\alpha+1}}$, there is at least one acknowledge packet, $\langle ldai, lbl \rangle$, that p_r sends and p_s receives, where $ldai = s\_index_\alpha$. This proves the claim because p_r's acknowledgments are always sent with $ldai = LastDeliveredIndex_r$, see line 8 (the Receiver algorithm).

We show that, between a_{s_α} and $a_{s_{\alpha+1}}$, the Receiver p_r sends at least one of the $ack_\alpha(\ell) \in ACK_\alpha$ packets that p_s receives. We do that by showing that p_s receives, from the channel from p_r to p_s, more than $capacity$ packets, i.e., the set ACK_α. Since $capacity$ bounds the number of packets that, at any time, can be in the channel from p_r to p_s, at least one of the ACK_α packets, say $ack_\alpha(\ell')$, must be sent by p_r and received by p_s between a_{s_α} and $a_{s_{\alpha+1}}$. This in fact proves that p_r sends $ack_\alpha(\ell')$ after c_{r_β}.

In order to demonstrate that p_s receives the set ACK_α, we note that $ACK\_set = \emptyset$ in configuration $c_{s_\alpha}(1)$, which immediately follows a_{s_α}, see line 5 (the Sender algorithm). The Sender tests the arriving acknowledgment packet, ack_α, in line 8 (the Sender algorithm). It tests ack_α's label to be in the range of $[1, capacity + 1]$, and that they are of ack_α's form. Moreover, it counts that $(capacity + 1)$ different packets are added to ACK_set by adding them to ACK_set, and not executing line 5 (the Sender algorithm) before at least $(capacity + 1)$ distinct packets are in ACK_set. ■

Lemma A.3 *Let $c_{r_\beta}(y)$ be the y^{th} configuration between a_{r_β} and $a_{r_{\beta+1}}$, and $PACKET_\beta(r\_index'_\beta) = \{packet_\beta(\ell, r\_index'_\beta)\}_{\ell \in [1,n]}$ be a packet set, which could be a subset of the Receiver's $packet\_set_r$, where $packet_\beta(\ell, r\_index'_\beta) = \langle r\_index'_\beta, \ell, *\rangle$.*

1. *For any given $\beta > 0$, there is a single index value, $r\_index_\beta \in [0,2]$, such that for any $y > 0$, it holds that $LastDeliveredIndex_r = r\_index_\beta$ in configuration $c_{r_\beta}(y)$.*
2. *Between a_{r_β} and $a_{r_{\beta+1}}$ there is at least one configuration, c_{s_α}, such that $AltIndex_s \neq r\_index_\beta$.*
3. *There exists a single $r\_index'_\beta \in [0,2] \setminus \{r\_index_\beta\}$, such that the Receiver, p_r, receives all the packets in $PACKET_\beta(r\_index'_\beta)$ at least once between c_{s_α} and $a_{r_{\beta+1}}$, where c_{s_α} is defined in 2, and at least $(n - capacity > 0)$ of them are sent by the Sender p_s between a_{r_β} and $a_{r_{\beta+1}}$.*

Proof. We begin the proof of claim by showing that r_index_β exists before showing that c_{s_α} exists and that p_r receives the packets $packet_{\beta,r\_index'_\beta}(\ell)$ from p_s.

The value of $LastDeliveredIndex_r = r\_index_\beta$ is only changed in line 6 (the Receiver algorithm). By the definition of a_{r_β}, line 6 is not executed by any step between a_{r_β} and $a_{r_{\beta+1}}$. Therefore, for any given β, there is a single index value, $r\_index_\beta \in [0,2]$, such that for any $y > 0$, it holds that $LastDeliveredIndex_r = r\_index_\beta$ in $c_{s_\beta}(y)$.

We show that c_{s_α} exists by showing that the Receiver, p_r, receives all the packets in $PACKET_\beta(r\_index'_\beta)$ from the channel from p_s to p_r, (each at least once) between a_{r_β} and $a_{r_{\beta+1}}$. Since $capacity$ bounds the number of packets that can be in the channel from p_s to p_r, at any time. Hence, a subset, $S_\beta(r\_index'_\beta) \subseteq PACKET_\beta(r\_index'_\beta)$, of at least $((n - capacity) > 0)$ packets must be sent by p_s between a_{r_β} and $a_{r_{\beta+1}}$. This in fact proves that p_s sends $S_\beta(r\_index'_\beta)$ after (the first) c_{s_α}, because p_s uses $r\_index''_\beta$ as the alternating index for all the packets in $S_\beta(r\_index'_\beta)$, see line 6 (the Sender algorithm) and the function $packet\_set()$, as well as by the previous argument we have $r\_index''_\beta = r\_index'_\beta$.

Now, we show that, between a_{r_β} and $a_{r_{\beta+1}}$, the Receiver p_r receives packets, $packet_{\beta,r\_index'_\beta}(\ell) \in PACKET_\beta(r\_index'_\beta)$, with n distinct labels from the channel from p_s to p_r . We note that $packet\_set_r = \emptyset$ in the configuration $c_{r_\beta}(1)$, which immediately follows a_{r_β}, see line 6 (the Receiver algorithm). The Receiver tests the arriving packets, $packet_{\beta,r\_index'_\beta}(\ell)$, in line 10 (the Receiver algorithm). It tests $packet_{\beta,r\_index'_\beta}(\ell)$'s label to be in the range of $[1,n]$, $packet_{\beta,r\_index'_\beta}(\ell)$'s index to be different from $LastDeliveredIndex_r$ and that they are of $packet_{\beta,r\_index'_\beta}(\ell)$'s form. Moreover, it counts that n packets with alternating index different from $LastDeliveredIndex_r$ and n distinct labels are added to $packet\_set_r$ by not executing lines 4 to 7 (the Receiver algorithm) before at least n distinct labels are in $packet\_set_r$. ■

Lemma A.4 borrows notation from lemmas A.2 and A.3.

Lemma A.4 *Suppose that $\alpha, \beta > 2$. Between a_{s_α} and $a_{s_{\alpha+1}}$, the Receiver takes at least one a_{r_β} step, and that between a_{r_β}, and $a_{r_{\beta+1}}$, the Sender takes at least one a_{s_α} step. Moreover, equations $(A.1)$ to $(A.4)$ hold.*

$$\begin{aligned} s\_index_{\alpha+1} &= s\_index_{\alpha} + 1 \bmod 3 && \text{(A.1)} \\ r\_index_{\beta+1} &= r\_index_{\beta} + 1 \bmod 3 && \text{(A.2)} \\ r\_index_{\beta} &= s\_index_{\alpha} && \text{(A.3)} \\ s\_index_{\alpha+1} &= r\_index_{\beta} + 1 \bmod 3 && \text{(A.4)} \end{aligned}$$

Proof.

Between a_{s_α} and $a_{s_{\alpha+1}}$, there is at least one a_{r_β} step. By Lemma A.2 and line 5 (the Sender algorithm), in any configuration, $c_{s_1}(x)$, that is between a_{s_1} and a_{s_2}, the Sender is using a single alternating index, s_index_1, and in any configuration, $c_{s_2}(x)$, that is between a_{s_2} and a_{s_3}, the Sender is using a single alternating index, s_index_2, such that $s\_index_2 = s\_index_1 + 1 \bmod 3$. In a similar manner, we consider configuration, $c_{s_\alpha}(x)$, that is between a_{s_α} and $a_{s_{\alpha+1}}$ and conclude equations $(A.1)$ and $(A.3)$, cf. items (1), and respectively, (2) of Lemma A.2.

Lemma A.2 also shows that for $\alpha \in \{1, 2, \ldots\}$, there are configurations, c_{r_β}, in which $LastDeliveredIndex_r = s\_index_\alpha$. This implies that between a_{s_α} and $a_{s_{\alpha+1}}$, the Receiver changes the value of $LastDeliveredIndex_r$ at least once, where $\alpha \in (1, 2, \ldots)$. Thus, by a_{r_β}'s definition and line 6 (the Receiver algorithm), there is at least one a_{r_β} step between a_{s_α} and $a_{s_{\alpha+1}}$.

Between a_{r_β} and $a_{r_{\beta+1}}$, there is at least one a_{s_α} step. By Lemma A.3 and line 6 (the Receiver algorithm), in any configuration, $c_{r_1}(y)$, that is between a_{r_1} and a_{r_2}, the Receiver is using a single $LastDeliveredIndex_r$, r_index_1, and in any configuration, $c_{r_2}(y)$, that is between a_{r_2} and a_{r_3}, the Receiver is using a single $LastDeliveredIndex_r$, r_index_2, such that $r\_index_2 = r\_index_1 + 1 \bmod 3$. In a similar manner, we consider configuration, $c_{r_\beta}(y)$, that is between a_{r_β} and $a_{r_{\beta+1}}$ and conclude equations $(A.2)$ and $(A.4)$, cf. items (1), and respectively, (2) of Lemma A.3.

Lemma A.3 also shows that for $\beta \in \{1, 2, \ldots\}$, there are configurations, c_{s_α}, in which $AltIndex_s \neq r\_index_\beta$. This implies that between a_{r_β} and $a_{r_{\beta+1}}$, the Sender changes the value of $AltIndex_s$ at least once. Thus, by a_{s_α}'s definition, there is at least one a_{s_α} step between a_{r_β} and $a_{r_{\beta+1}}$.

■

A.2.2 Closure

The closure property proof considers all the alternating indices that are in a given configuration, c, such as the packet set indices, $\langle ind, lbl\rangle \in packet\_set = \{\langle AltIndex, lbl, dat\rangle\}$, and the indices of the acknowledgment packet set, $\langle ind, lbl\rangle \in ACK\_set = \{\langle AltIndex, lbl\rangle\}$. Given $X \in \{packet\_set, ACK\_set\}$, we define $index(ind, X) = \{\langle ind, lbl\rangle : \langle ind, lbl\rangle \in X \vee \langle ind, lbl, *\rangle \in X\}$. We denote by $\{0^{\kappa_0}, 1^{\kappa_1}, 2^{\kappa_2}\}_X$ the fact that in configuration c, it holds $\forall i \in [0, 2] : \kappa_i = |index(i, X)|$. We consider the alternating index sequence, ais, stored in $AltIndex_s, \{0^{\kappa_0}, 1^{\kappa_1}, 2^{\kappa_2}\}_{channel_{s,r}}, \{0^{\kappa_0}, 1^{\kappa_1}, 2^{\kappa_2}\}_{packet\_set_r}$, LDI_r, $\{0^{\kappa_0}$, 1^{κ_1}, $2^{\kappa_2}\}_{channel_{r,s}}$, and $\{0^{\kappa_0}$, 1^{κ_1}, $2^{\kappa_2}\}_{ACK\_set_s}$ in this order, where $LDR_r = LastDeliveredIndex_r$ as well as $channel_{s,r}$ and $channel_{r,s}$ are the communication channel sets from the Sender to the Receiver, and respectively, from the Receiver to the Sender. We show that a configuration, c, in which $ais = y$, $\{*^*\}$, $\{z^{\kappa_z}\}_{z\in[0,2]\setminus\{y\}}$, y, $\{*^*\}$, $\{y^{capacity+1}\}$ is a safe configuration (Theorem A.5), where $y \in [0, 2]$ and $capacity \geq (\sum_{z\in[0,2]\setminus\{y\}} \kappa_z)$. Namely, c starts an execution that is in $LE_{S^2E^2C}$.

Theorem A.5 (**S^2E^2C closure**) *Suppose that in R's first configuration, c, it holds that $ais = y$, $\{*^*\}$, $\{z^{\kappa_z}\}_{z\in[0,2]\setminus\{y\}}$, y, $\{*^*\}$, $\{y^{capacity+1}\}$ is a safe configuration, where $y \in [0, 2]$ and $capacity \geq (\sum_{z\in[0,2]\setminus\{y\}} \kappa_z)$. Then, c is safe.*

Proof. The correctness proof shows after configuration c, the system reaches configurations in which: (1) the Sender, p_s, increments its alternating index and starts transmitting a new message batch, m, (2) p_r, receives between $(n - capacity)$ and n of m's packets (with that new alternating index), and (3) p_s receives at least one acknowledgment (with an alternating index) in which the p_r acknowledges m's packets. The proof shows that this is how p_s and p_r exchange messages and alternative indices. Therefore, c starts a legal execution. For the sake of presentation simplicity, we assume that $y = 0$.

In c, p_s's state satisfies the condition $(ACK\_set = \{AltIndex\} \times [1, capacity + 1])$ of line 5 (the Sender algorithm). Therefore, p_s increments $AltIndex$ (mod 3), empties ACK_set_s and fetches a new batch of pl messages, m, that it needs to sent to p_r. Thus, the system reaches configuration c' in which $ais = 1$, $\{*^*\}$, $\{*^*\}$, 0, $\{*^*\}$, $\{\}$.

Note that by lines 5 to 6 (the Sender algorithm), p_s does not stop sending m's packets with alternating indices 1 until ACK_set_s has $(capacity + 1)$ packets with the alternating index that is equal to $AltIndex_s = 1$. Until that happens, lines 8 and 9 (the Sender algorithm) implies that p_s accepts acknowledgments that their alternating index is $AltIndex_s = 1$, i.e., $ais = 1$, $\{1^*, *^*\}$, $\{*^*\}$, 0, $\{*^*\}$, $\{1^*\}$.

By line 3, as well as lines 9 and 11 (the Receiver algorithm), p_r does not stop accepting

m's packets, which have alternating indices 1, until $packet\_set_r$ has n packets with an alternating index that is different from $LastDeliveredIndex_r = 0 \neq 1$. Recall that the communication channel set, $channel_{s,r}$, from the Sender to the Receiver contains at most $capacity$ packets. Therefore, once p_r has n packets in $packet\_set$ (with alternating index $ai \neq 0$), p_r must have received at least $(n - capacity)$ of these packets from p_s. Thus, the system reaches configuration c'' in which $ais = 1$, $\{1^*, *^*\}$, $\{1^n, 2^{\kappa_2}\}$, 0, $\{*^*\}$, $\{1^*\}$, where $capacity \geq \kappa_2$.

By lines 3 to 7 (the Receiver algorithm), p_r empties $packet\_set_r$, updates $LastDeliveredIndex_r$ with the alternating index, 1, of m's packets, before decoding and delivering the messages encoded by $packet\_set_r$, as well as starting to send acknowledgements with the alternating index $LastDeliveredIndex_r = 1$, see line 8 (the Receiver algorithm). Thus, the system reaches configuration c''' in which $ais = 1$, $\{1^*, *^*\}$, $\{\}$, 1, $\{1^*, *^*\}$, $\{1^*\}$.

By line 8 (the Receiver algorithm) and lines 8 and 9 (the Sender algorithm), p_r keeps on acknowledging m's packets until p_s receives $(capacity + 1)$ packets of acknowledgement from p_r. Thus, the system reaches configuration c'''' in which $ais = 1$, $\{1^*, *^*\}$, $\{0^*, 2^*\}$, 1, $\{1^*, *^*\}$, $\{1^{(capacity+1)}\}$.

Note that, for $y = 1$, this lemma claims that c'''' is safe. Moreover, since we started in a configuration in which the communication channel sets from the Sender to the Receiver, and the Receiver to the Sender had no $n > capacity$, and respectively, $(capacity + 1)$ packets with the alternating index 1 exist, the Sender must have received at least one acknowledgment for m's packet with the alternating index 1 only after the Receiver receives at least one of for m's packet with alternating index 1, which happened after p_s had fetched the batch messages of m and incremented its alternating index to 1. Therefore, c starts a legal execution. ■

A.2.3 Convergence

Lemma A.4 facilitates the proof of Theorem A.6.

Theorem A.6 (**S^2E^2C convergence**) *Within $\mathcal{O}(n)$ asynchronous rounds of any nice execution, the system reaches a safe configuration (from which a legal execution starts), where n is the packet word length.*

Proof. Theorem A.1 shows that R includes, infinitely often, the steps a_{s_α} and a_{r_β}. In fact, they appear within $\mathcal{O}(n)$ asynchronous rounds. We show that within a constant number of their appearance in R, the system reaches a safe configuration.

The proof of this theorem borrows notation from lemmas A.2 and A.3. Let $c_{s_\alpha}(1)$

and $c_{r_\beta}(1)$ be the first configurations between a_{s_α} and $a_{s_{\alpha+1}}$, and respectively, between a_{r_β} and $a_{r_{\beta+1}}$. Moreover, s_index_α and r_index_β are $AltIndex_s$'s value in $c_{s_\alpha}(1)$, and respectively, $LastDeliveredIndex_r$'s value in $c_{r_\beta}(1)$. Suppose that in $c_{s_\alpha}(1)$, it holds that $ais = s\_index_\alpha$, $\{*^*\}$, $\{*^*\}$, r_index_β, $\{*^*\}$, $\{*^*\}$. Let $capacity \geq (\sum_{z\in[0,2]\setminus\{r\_index_{\beta+1}\}} \kappa_z)$. We show that, within $\mathcal{O}(n)$ asynchronous rounds, there is a configuration in which $ais = r\_index_{\beta+1}$, $\{*^*\}$, $\{z^{\kappa_z}\}_{z\in[0,2]\setminus\{r\_index_{\beta+1}\}}$, $r\_index_{\beta+1}$, $\{*^*\}$, $\{r\_index_{\beta+1}^{capacity+1}\}$.

By Lemma A.4, $\forall\alpha, \beta > 2$ it holds that between $c_{s_\alpha}(1)$ and $c_{s_{\alpha+1}}(1)$ the system execution includes $c_{r_\beta}(1)$ in which Equation $(A.3)$ holds. Namely, $\forall\alpha, \beta > 3$, it holds that $r\_index_{\beta+1} = s\_index_\alpha$, and thus, in $c_{r_\beta}(1)$ it holds that $ais = r\_index_{\beta+1}$, $\{*^*\}$, $\{z^{\kappa_z}\}_{z\in[0,2]\setminus\{r\_index_{\beta+1}\}}$, $r\_index_{\beta+1}$, $\{*^*\}$, $\{r\_index_{\beta+1}^{\kappa_{r\_index_{\beta+1}}}\}$. The rest of the proof shows that $capacity \geq (\sum_{z\in[0,2]\setminus\{r\_index_{\beta+1}\}} \kappa_z)$ and $capacity + 1 = \kappa_{r\_index_{\beta+1}}$, and it follows by arguments similar to the ones in the proof of Theorem A.5. ■

Appendix B

Proof of NP-Hardness of Mapping Schema Problems (Chapter 4)

Theorem 4.5 *The problem of finding whether a mapping schema of m inputs of different input sizes exists, where every two inputs are assigned to at least one of $z \geq 3$ identical-capacity reducers, is NP-hard.*

Proof. The proof is by a reduction from the partition problem [58] that is a known NP-complete problem. The partition problem is defined as follows: given a set $I = \{i_1, i_2, \ldots, i_m\}$ of m positive integer numbers, it is required to find two disjoint subsets, $S_1 \subset I$ and $S_2 \subset I$, so that the sum of numbers in S_1 is equal to the sum of numbers in S_2, $S_1 \cap S_2 = \emptyset$, and $S_1 \cup S_2 = I$.

We are given m inputs whose input size list is $W = \{w_1, w_2, \ldots, w_m\}$, and the sum of the sizes is $s = \Sigma_{1 \leq i \leq m} w_i$. We add $z-3$ additional inputs, $ai_1, ai_2, \ldots, ai_{z-3}$, each of size $\frac{s}{2}$. We call these new $z-3$ ($ai_1, ai_2, \ldots, ai_{z-3}$) inputs the *medium inputs*. In addition, we add one more additional input, ai', of size $\frac{(z-2)s}{2}$ that we call the *big input*. Further, we assume that the reducer capacity is $\frac{(z-1)s}{2}$.

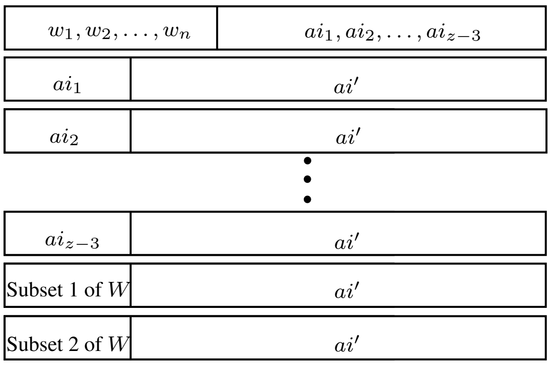


Figure B.1: Proof of NP-hardness of the *A2A mapping schema problem* for $z > 2$ identical-capacity reducers, Theorem 4.5.

The proof proceeds in two steps: (*i*) we prove that in case the m original inputs can be partitioned, then all the inputs can be assigned to the z reducers such that every two inputs are assigned to at least one reducer, (*ii*) we prove that in case the mapping schema for all the inputs over the z reducers is successful, then there are two disjoint subsets S_1 and S_2 of the m original inputs that satisfy the partition requirements. We can assume that if the sum is not divisible by 2, then the answer to the partition problem is surely "no," so the reduction of the partition problem to the *A2A mapping schema problem* is trivial.

We first show that if there are two disjoint subsets S_1 and S_2 of equal size of the m original inputs, then there must exist a solution to the *A2A mapping schema problem*. Recall that any of the reducers can hold a set of inputs whose sum of the sizes is at most $\frac{(z-1)s}{2}$, and the sum of the sizes of the new $z-3$ medium inputs is exactly $\frac{(z-3)s}{2}$. Hence, all the m original inputs $(i_1, i_2, \ldots, i_m)$ and a list of the $z-3$ medium inputs can be assigned to a single reducer (out of the z reducers), and this assignment uses $s + \frac{(z-3)s}{2}$ capacity, which is exactly the capacity of any reducer. Further, the big input, ai', of size $\frac{(z-2)s}{2}$ can share the same reducer with only one medium input ai_i (it could also share with original inputs). Thus, the big input, ai', and all the medium inputs are assigned to $z-3$ reducers (out of the remaining $z-1$ reducers). In addition, the remaining two reducers can be used for the following assignment: the first reducer is assigned the set S_1 and the big input, ai', and the second reducer is assigned the set S_2 and the big input, ai'. The above assignment is a solution to the *A2A mapping schema problem* for the given m original inputs, the $z-3$ medium inputs, and the big input using z reducers, see Figure B.1.

Now, we show that a solution to the *A2A mapping schema problem* — for all the inputs over the z reducers — results in a partition of the m original inputs into two equal-sized blocks. We also show that in a solution to the *A2A mapping schema problem*, each of the m original inputs and every medium input, ai_i, are assigned to exactly two reducers, and the big input, ai', is assigned to exactly $z-1$ reducers. Recall that the total sum of the sizes is $s + \frac{(z-3)s}{2} + \frac{(z-2)s}{2} = \frac{(2z-3)s}{2}$.

Due to the reducer capacity of a single reducer, all the inputs cannot be assigned to a single reducer; only a sublist of the inputs, whose sum of the sizes is at most $\frac{(z-1)s}{2}$, can be assigned to one reducer. Thus, each input is assigned to at least two reducers in order to be coupled with all the other inputs.

Moreover, the big input, ai', can share the same single reducer with only a sublist, S', whose sum of the sizes is at most $\frac{s}{2}$. Hence, the big input, ai', is required to be assigned to at least $z-3$ reducers in order to be paired with the medium inputs ai_i. Furthermore, the big input, ai', can share the same reducer with a sublist of the m original inputs whose sum of the sizes is at most $\frac{s}{2}$. This fact means that the big input, ai', must be assigned to two more reducers. On the other hand, all the medium inputs can share the same reducer with the original m inputs. Thus, here, the total reducer capacity occupied by all the inputs is $2 \times \Sigma_{1 \leq i \leq m} w_i + 2 \times \frac{(z-3)s}{2} + (z-1) \times \frac{(z-2)s}{2} = 2s + (z-3)s + \frac{(z-1)(z-2)s}{2} = \frac{(z-1)zs}{2}$, which is exactly the total capacity of all the z reducers. Thus, each of the m original inputs and each medium input ai_i cannot be assigned more than twice, and hence, each is assigned exactly twice. In addition, the big input, ai', is assigned to exactly $z-1$ reducers. This fact also shows that all the reducers are entirely filled with distinct inputs. Thus, a solution to the *A2A mapping schema problem* yields partitions of the m original inputs to S_1 and S_2 blocks, where the sum of the input sizes of any block is exactly $\frac{s}{2}$. Therefore, if there is a polynomial-time algorithm to construct the mapping schema, where every input is required to be paired with every other input, then the mapping schema finds the partitions of the m original inputs in polynomial time. ■

Theorem 4.6 *The problem of finding whether a mapping schema of m and n inputs of different input sizes that belongs to list X and list Y, respectively, exists, where every two inputs, the first from X and the second from Y, are assigned to at least one of $z \geq 2$ identical-capacity reducers, is NP-hard.*

Proof. The proof is by a reduction from the partition problem [58] that is a known NP-complete problem. We are given a list of inputs $I = \{i_1, i_2, \dots, i_m\}$ whose input size list is $W = \{w_1, w_2, \dots, w_m\}$, and the sum of the sizes is $s = \Sigma_{1 \leq i \leq m} w_i$. We add $z-2$ additional inputs, $ai_1, ai_2, \dots, ai_{z-2}$, each of size $\frac{s}{2}$. We call these new $z-2$ $(ai_1, ai_2, \dots, ai_{z-2})$ inputs the *big inputs*. In addition, we add one more additional input, ai', of size 1 that we call the *small input*. Further, we assume that the reducer capacity is $1 + \frac{s}{2}$. Now, the list I holds $m + z - 1$ inputs.

For the *X2Y mapping schema problem*, we consider m original inputs and the $z-2$ big inputs as a list X, and the small input as a list Y. A solution to the *X2Y mapping schema problem* assigns each of the m original inputs and each big input (of the list X) with the small input of the list Y.

The proof proceeds in two steps: (*i*) we prove that in case the m original inputs can be partitioned, then all the m original inputs, the $z-2$ big inputs, and the small input can be assigned to the z reducers such that they satisfy the *X2Y mapping schema problem*, (*ii*) in case the *X2Y mapping schema problem* is successful, then there are two disjoint subsets, S_1 and S_2, of the m original inputs that satisfy the partition requirements.

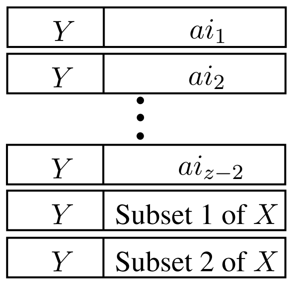


Figure B.2: Proof of NP-hardness of the *X2Y mapping schema problem* for $z > 1$ identical-capacity reducers, Theorem 4.6.

We first show that if there are two disjoint subsets S_1 and S_2 of equal size of the m original inputs, then there must exist a solution to the *X2Y mapping schema problem*. Recall that any of the reducers can hold a set of inputs whose sum of sizes is at most $1 + \frac{s}{2}$, and the sum of the sizes of the new $z-2$ big inputs is exactly $\frac{s}{2}$. Hence, the small input, ai', of size 1 and each big input, ai_i, can be assigned to $z-2$ reducers (out of the z reducers), and this assignment uses $1+\frac{s}{2}$ capacity, which is exactly the capacity of any reducer. In addition, the remaining two reducers can be used for the following assignment: the first remaining reducer is assigned the set S_1 and the small input, ai', and the second remaining reducer is assigned the remaining original inputs, S_2, and the small input, ai'. The above assignment is a solution to the *X2Y mapping schema problem* (for the given $m+z-2$ inputs of the list X and the one input of the list Y using z reducers, see Figure B.2).

Now, we prove the second claim that a solution to the *X2Y mapping schema problem* results in a partition of the m original inputs into two equal-sized blocks. Recall that the total sum of the sizes is $s+\frac{(z-2)s}{2}+1=\frac{z\times s}{2}+1$.

Due to the reducer capacity of a single reducer, all the inputs cannot be assigned to a single reducer; only a sublist of the inputs, whose sum of the sizes is at most $1+\frac{s}{2}$, can be assigned to a single reducer. We show that the small input, ai', must be assigned to all the z reducers. The small input, ai', of size one can share the same single reducer with only a subset, S', whose sum of the sizes is at most $\frac{s}{2}$. Hence, the small input, ai', is required to be assigned to $z-2$ reducers (out of z reducers) in order to be paired with all the big inputs ai_i. and the remaining two reducers in order to be paired with all the m original inputs. This fact results in that a solution to the *X2Y mapping schema problem* yields partitions of the m original inputs to S_1 and S_2 blocks, where the sum of the input sizes of any block is exactly $\frac{s}{2}$. Therefore, if there is a polynomial-time algorithm to construct the mapping schema, where every input of one list is required to be paired with every other input of another list, then the mapping schema finds the partitions of the m original inputs in polynomial time.
■

Appendix C

Pseudocodes of Approximation Algorithms for Mapping Schema Problems and their Proofs (Chapter 5)

C.1 Preliminary Proofs of Theorems on Lower and Upper Bounds

Theorem 5.1 (Lower bounds on the communication cost and number of reducers) *For a list of inputs and a given reducer capacity q, the communication cost and the number of reducers, for the A2A mapping schema problem, are at least $\frac{s^2}{q}$ and $\frac{s^2}{q^2}$, respectively, where s is the sum of all the input sizes.*

Proof. Since an input i is replicated to at least $\lfloor \frac{s-w_i}{q-w_i} \rfloor$ reducers, the communication cost for the input i is $w_i \times \lfloor \frac{s-w_i}{q-w_i} \rfloor$. Hence, the communication cost for all the inputs will be at least $\sum_{i=1}^{m} w_i \frac{s-w_i}{q-w_i}$. Since $s \geq q$, we can conclude $\frac{s-w_i}{q-w_i} \geq \frac{s}{q}$. Thus, the communication cost is at least $\sum_{i=1}^{m} w_i \frac{s}{q} = \frac{s^2}{q}$.

Since the communication cost, the total number of bits to be assigned to reducers, is at least $\frac{s^2}{q}$, and a reducer can hold inputs whose sum of the sizes is at most q, the number of reducers must be at least $\frac{s^2}{q^2}$. ∎

Theorem 5.2 (Lower bound on the communication cost) *Let $q > 1$ be the reducer capacity, and let $\frac{q}{k}$, $k > 1$, is the bin size. Let the sum of the given inputs is s. The communication cost, for the A2A mapping schema problem, is at least $s\lfloor \frac{\frac{sk}{q}-1}{k-1} \rfloor$.*

Proof. A bin can hold inputs whose sum of the sizes is at most $\frac{q}{k}$. Since the total sum of the sizes is s, it is required to divide the inputs into at least $x = \frac{sk}{q}$ bins. Now, each bin can be considered as an identical sized input.

Since a bin i is required to be sent to at least $\lfloor \frac{x-1}{k-1} \rfloor$ reducers (to be paired with all the other bins), the sum of the number of copies of (x) bins sent to reducers is at least $x\lfloor \frac{x-1}{k-1} \rfloor$. We need to multiply this by $\frac{q}{k}$ (the size of each bin) to find the communication cost. Thus, we have at least

$$x\Big\lfloor \frac{x-1}{k-1} \Big\rfloor \frac{q}{k} =$$

$$\frac{sk}{q}\Big\lfloor \frac{\frac{sk}{q}-1}{k-1} \Big\rfloor \frac{q}{k} = s\Big\lfloor \frac{\frac{sk}{q}-1}{k-1} \Big\rfloor \approx \frac{s^2}{q} \cdot \frac{k}{k-1}$$

communication cost. ■

Theorem 5.3 (Upper bounds on communication cost and number of reducers for $k = 2$ using bin-packing) *The bin-packing-based approximation algorithm using a bin size $b = \frac{q}{2}$ where q is the reducer capacity achieves the following upper bounds: the number of reducers, and the communication cost, for the A2A mapping schema problem, are at most $\frac{8s^2}{q^2}$, and at most $4\frac{s^2}{q}$, respectively, where s is the sum of all the input sizes.*

Proof. A bin i can hold inputs whose sum of the sizes is at most b. Since the total sum of the sizes is s, it is required to divide the inputs into at least $\frac{s}{b}$ bins. Since the FFD or BFD bin-packing algorithm ensures that all the bins (except only one bin) are at least half-full, each bin of size $\frac{q}{2}$ has at least inputs whose sum of the sizes is at least $\frac{q}{4}$. Thus, all the inputs can be placed in at most $\frac{s}{q/4}$ bins of size $\frac{q}{2}$. Since each bin is considered as a single input, we can assign every two bins to a reducer, and hence, we require at most $\frac{8s^2}{q^2}$ reducers. Since each bin is replicated to at most $4\frac{s}{q}$ reducers, the communication cost is at most $\sum_{1\leq i\leq m} w_i \times 4\frac{s}{q} = 4\frac{s^2}{q}$. ■

C.2 Lower Bounds for Equal-Sized Inputs

Theorem 5.4 (Lower bounds on the communication cost and number of reducers) *For a given reducer capacity $q > 1$ and a list of m inputs of size one, the communication cost and the number of reducers ($r(m, q)$), for the A2A mapping schema problem, are at least $m\lfloor \frac{m-1}{q-1} \rfloor$ and at least $\lfloor \frac{m}{q} \rfloor \lfloor \frac{m-1}{q-1} \rfloor$, respectively.*

Proof. Since an input i is required to be sent to at least $\lfloor \frac{m-1}{q-1} \rfloor$ reducers, the sum of the number of copies of (m) inputs sent to reducers is at least $m\lfloor \frac{m-1}{q-1} \rfloor$, which result in at least $m\lfloor \frac{m-1}{q-1} \rfloor$ communication cost.

There are at least $m\lfloor \frac{m-1}{q-1} \rfloor$ total number of copies of (m) inputs to be sent to reducers and a reducer can hold at most q inputs; hence, $r(m, q) \geq \lfloor \frac{m}{q} \rfloor \lfloor \frac{m-1}{q-1} \rfloor$. ■

C.3 Algorithm for an Odd Value of the Reducer Capacity (Algorithm 7A)

Algorithm 7: Part A

```
Inputs: $m$: the number of bins obtained after placing all the given $m'$ inputs (of size
  $\leq \frac{q}{k}$, $k > 3$ is an odd number) to bins each of size $\frac{q}{k}$,
$q$: the reducer capacity.
Variables:
$A$: A set $A$, where the total inputs in the set $A$ is $y = \lfloor \frac{q}{2} \rfloor (\lfloor \frac{2m}{q+1} \rfloor + 1)$
$B$: A set $B$, where the total inputs in the $B$ is $x = m - y$
$Team[i, j]$ : represents teams of reducers, where index $i$ indicates $i^{th}$ team and index $j$
  indicates $j^{th}$ reducer in $i^{th}$ team. Consider $u = \lceil \frac{y}{q - \lceil q/2 \rceil} \rceil$. There are $u - 1$ teams of
  $v = \lceil \frac{u}{2} \rceil$ reducers in each team.
$groupA[]$ : represents disjoint groups of inputs of the set $A$, where $groupA[i]$ indicates
  $i^{th}$ group of $\lceil \frac{q-1}{2} \rceil$ inputs of the set $A$.
1  Function $create\_group(y)$ begin
2      for $i \leftarrow 1$ to $u$ do $groupA[i] \leftarrow \langle i, i+1 \ldots, i + \frac{q-1}{2} - 1 \rangle, i \leftarrow i + \frac{q-1}{2}$ ;
3      $2\_step\_odd\_q(1, u)$, $Last\_Team(groupA[])$, $Assign\_input\_from\_B(Team[])$
4  Function $2\_step\_odd\_q(lower, upper)$ begin
5      if $\lfloor \frac{upper - lower}{2} \rfloor < 1$ then return;
6      else
7          $mid \leftarrow \lceil \frac{upper - lower}{2} \rceil$, $Assignment(lower, mid, upper)$
8          $2\_step\_odd\_q(lower, mid)$, $2\_step\_odd\_q(mid + 1, upper)$
9  Function $Assignment(lower, mid, upper)$ begin
10     while $mid > 1$ do
11         foreach $(a, t) \in [lower, lower + mid - 1] \times [0, mid - 1]$ do
               $Team\left[(u - 2 \cdot mid + 1) + t, a - \lfloor \frac{a-1}{mid} \rfloor \cdot \frac{mid}{2}\right] \leftarrow$
               $\langle groupA[a], groupA[value\_b(a, t, mid, upper)] \rangle$ ;
12 Function $value\_b(a, t, mid, upper)$ begin
13     if $a + t + mid < upper + 1$ then return $(a + t + mid)$ ;
14     else if $a + t + mid > upper$ then return $(a + t)$ ;
15 Function $Last\_Team(lower, mid, upper)$ begin
16     foreach $i \in [1, v]$ do $Team[u - 1, i] \leftarrow groupA[2 \times i - 1], groupA[2 \times i]$ ;
17 Function $Assign\_input\_from\_B(Team[])$ begin
18     foreach $(i, j) \in [1, u - 1] \times [1, v]$ do $Team[i, j] \leftarrow B[i]$ ;
```

Algorithm description. First, we divide m inputs (that are actually bins of size $\frac{q}{k}$, $k > 3$, after placing all the given m inputs to m' bins, each of size $\frac{q}{k}$) into two sets A and B. Then, we make $u = \lceil \frac{y}{q - \lceil q/2 \rceil} \rceil$ disjoint groups of y inputs of the set A such that each group holds $\frac{q-1}{2}$ inputs, lines 1, 2. (Now, each of the groups is considered as a single input that we call the *derived input*.) We do not show the addition of dummy inputs and assume that u is a power of 2. Function $2\_step\_odd\_q(lower, upper)$ recursively divides the derived inputs into two halves, line 10. Function $Assignment(lower, mid, upper)$ (line 9) pairs every two derived inputs and assigns them to the respective reducers (line 7). Each reducer of the last team is assigned using function $Last\_Team(groupA[])$, lines 15, 16.

Note that functions $2\_step\_odd\_q(lower, upper)$, $Assignment(lower, mid, upper)$, and $value\_b(lower, t, mid, upper)$ take two common parameters, namely $lower$ and $upper$ where $lower$ is the first derived input and $upper$ is the last derived input (*i.e.*, u^{th} group) at the time of the first call to functions, line 3. Once all-pairs of the derived inputs are assigned to reducers, line 7, function $Assign\_input\_from\_B(Team[])$ assigns i^{th} input of the set B to all the $\lceil \frac{u}{2} \rceil$ reducers of i^{th} team, lines 17, 18. After that, Algorithm 7A is invoked over inputs of the set B to assign each pair of the remaining inputs of the set B to reducers until every pair to the remaining inputs is assigned to reducers.

C.4 Algorithm for an Even Value of the Reducer Capacity (Algorithm 7B)

Algorithm 7: Part B

```
Inputs: $m$: the number of bins obtained after placing all the given $m'$ inputs (of size
  $\leq \frac{q}{k}$, $k \geq 4$ is an even number) to bins each of size $\frac{q}{k}$,
$q$: the reducer capacity.
Variables:
$Team[i, j]$ : represents teams of reducers, where index $i$ indicates $i^{th}$ team and index $j$
  indicates $j^{th}$ reducer in $i^{th}$ team. Consider $u = \frac{2m}{q}$. There are $u - 1$ teams of $\lceil \frac{u}{2} \rceil$
  reducers in each team.
$groupA[]$ : represents disjoint groups of inputs of the set $A$, where $groupA[i]$ indicates
  $i^{th}$ group of $\lceil \frac{q}{2} \rceil$ inputs of the set $A$.
1 Function $create\_group(m)$ begin
2     for $i \leftarrow 1$ to $u$ do $groupA[i] \leftarrow \langle i, i+1 \ldots, i + \frac{q}{2} - 1 \rangle, i \leftarrow i + \frac{q}{2}$ ;
3     $2\_step\_even\_q(1, u)$, $Last\_Team(1, \lceil \frac{u-1}{2} \rceil, u)$
4 Function $2\_step\_even\_q(lower, upper)$ begin
5     if $\lfloor \frac{upper-lower}{2} \rfloor < 1$ then return;
6     else
7         $mid \leftarrow \lceil \frac{upper-lower}{2} \rceil$, $Assignment(lower, mid, upper)$
8         $2\_step\_even\_q(lower, mid)$, $2\_step\_even\_q(mid + 1, upper)$
```

C.5 Proof of Lemmas and Theorems related to Algorithms 7A and 7B

Lemma 5.10 *Let q be the reducer capacity. Let the size of an input is $\lceil \frac{q-1}{2} \rceil$. Each pair of $u = 2^i$, $i > 0$, inputs can be assigned to $2^i - 1$ teams of 2^{i-1} reducers in each team.*

Proof. The proof is by induction on i.

Basis case. For $i = 1$, we have $u = 2$ inputs, and we can assign them to a team of one reducer of capacity q. Hence, Lemma 5.11 holds for ($i = 1$) two inputs.

Inductive step. Assume that the inductive hypothesis — there is a solution for $u = 2^{i-1}$ inputs, where all-pairs of $u = 2^{i-1}$ inputs are assigned to $2^{i-1} - 1$ teams of 2^{i-2} reducers in each team and have the team property (each team has one occurrence of each input, which we will prove in algorithm correctness) — is true. Now, we can build a solution for $u = 2^i$ inputs, as follows:

(a) Divide $u = 2^i$ inputs into two groups of 2^{i-1} inputs in each group,

(b) Recursively create teams for each of the two groups,

(c) Create some of the teams for the 2^i inputs by combining the j^{th} team from the first group with the j^{th} team from the second group. Since by the inductive hypothesis we have a solution for $u = 2^{i-1}$ inputs, we can assign inputs of these two groups to $2 \cdot (2^{i-1} - 1)$ teams of 2^{i-2} reducers in each team. And, by combining j^{th}, where $j = 1, 2, \ldots, (2^{i-1} - 1)$, teams of each group, there are $2^{i-1} - 1$ teams of 2^{i-1} reducers in each team; see Teams 5-7 for 8 inputs in Figure 5.3.

(d) Create 2^{i-1} additional teams that pair the inputs from the first group with inputs from the second group. In each team, the j^{th} input from the first group is assigned to the j^{th} reducer. In the first team, the j^{th} input from the second group is also assigned to the j^{th} reducer. In subsequent teams, the assignments from the second group rotate, so in the t^{th} team, the j^{th} input from the second group is assigned to reducer $k + j - (2^{i-1} - 1)(modulo 2^{i-1})$; see Teams 1-4 for 8 inputs in Figure 5.3.

By steps (c) and (d), there are total $2^{i-1} - 1 + 2^{i-1} = 2^i - 1$ teams of 2^{i-1} reducers in each team, and these teams holds each pair of the $u = 2^i$ inputs. ■

Theorem 5.11 (The communication cost obtained using Algorithm 7A or 7B) *For a given reducer capacity $q > 1$, $k > 3$, and a list of m inputs whose sum of sizes is s, the communication cost, for the A2A mapping schema problem, is at most $\frac{q}{2k} \lceil \frac{sk}{q(k-1)} \rceil (\lceil \frac{sk}{q(k-1)} \rceil - 1)$.*

Proof. Since the FFD or BFD bin-packing algorithm ensures that all the bins (except only one bin) are at least half-full, each bin of size $\frac{q}{k}$ has at least inputs whose sum of the sizes is at least $\frac{q}{k/2}$. Thus, all the inputs can be placed in at most $x = s/(q/(k/2)) = \frac{sk}{2q}$ bins of size $\frac{q}{k}$. Now, each bin can be considered as an identical sized input.

According to the construction given in Algorithm 7A, there are at most $g = \lceil \frac{2x}{k-1} \rceil$ groups (derived inputs) of the given x bins. In order to assign each pair of the derived inputs, each derived input is required to assign to at most $g-1$ reducers. In addition, the size of each input (bin) is $\frac{q}{k}$, therefore we have at most

$$\frac{q}{k} \times g(g-1)/2 = \frac{q}{k} \times \left\lceil \frac{2x}{k-1} \right\rceil \Big(\left\lceil \frac{2x}{k-1} \right\rceil - 1 \Big)/2$$

$$= \frac{q}{2k} \times \left\lceil \frac{sk}{q(k-1)} \right\rceil \Big(\left\lceil \frac{sk/q}{k-1} \right\rceil - 1 \Big) > \frac{4s^2}{q}$$

communication cost. ■

C.5.1 Correctness of Algorithm 7A

The algorithm correctness proves that every pair of inputs is assigned to reducers. Specifically, we prove that all those pairs of inputs, $\langle i, j \rangle$ and $\langle i', j' \rangle$, of the set A are assigned to a team whose $i \neq i'$ and $j \neq j'$ (Claim C.1). Then that all the inputs of the set A appear exactly once in each team (Claim C.2). We then prove that the set B holds $x \leq y - 1$ inputs, when $q = 3$ (Claim C.3). At last we conclude in Theorem C.4 that Algorithm 7A assigns each pair of inputs to reducers.

Note that we are proving all the above mentioned claims for $q = 3$; the cases for $q > 3$ can be generalized trivially where we make $u = \lceil \frac{y}{q - \lceil q/2 \rceil} \rceil$ derived inputs from y inputs of the set A (and assign in a manner that all the inputs of the A are paired with all the remaining $m-1$ inputs).

Claim C.1 *Pairs of inputs $\langle i, j \rangle$ and $\langle i', j' \rangle$, where $i = i'$ or $j = j'$, of the set A are assigned to different teams.*

Proof. First, consider $i = i'$ and $j \neq j'$, where $\langle i, j \rangle$ and $\langle i', j' \rangle$ must be assigned to two different teams. If $j \neq j'$, then both the j values may have an identical value of $lower$ and mid but they must have two different values of t (see lines 13, 14 of Algorithm 7A), where $j = lower + t + mid$ or $j = lower + t$. Thus, for two different values of j , we use two different values of t, say t_1 and t_2, that results in an assignment of $\langle i, j \rangle$ and $\langle i', j' \rangle$ to two different teams t_1 and t_2, (note that teams are also selected based on the value of t, $(y - 2 \cdot mid + 1) + t$, see line 7 of Algorithm 7A, where for $q = 3$, we have $u = y$). Suppose now that $i \neq i'$ and $j = j'$, where $\langle i, j \rangle$ and $\langle i', j' \rangle$ must be assigned to two different teams. In this case, we also have two different values of t, and hence, two different t values assign

$\langle i, j \rangle$ and $\langle i', j' \rangle$ to two different teams ($(y - 2 \cdot mid + 1) + t$, line 7 of Algorithm 7A).

Hence, it is clear that pairs $\langle i, j \rangle$ and $\langle i', j' \rangle$, where $i \neq i'$ and $j \neq j'$, are assigned to a team. ■

Claim C.2 *All the inputs of the set A appear exactly once in each team.*

Proof. There are the same number of pairs of inputs of the set A and the number of reducers ($(y - 1)\lceil \frac{y}{2} \rceil$) that can provide a solution to the *A2A mapping schema problem* for the y inputs of the set A. Recall that $(y-1)\lceil \frac{y}{2} \rceil$ reducers are arranged in the form of $(y-1)$ teams of $\lceil \frac{y}{2} \rceil$ reducers in each team, when $q = 3$. Note that if there is an input pair $\langle i, j \rangle$ in team t, then the team t cannot hold any pair that has either i or j in the remaining $\lceil \frac{y}{2} \rceil - 1$ reducers. For the given y inputs of the set A, there are at most $\lceil \frac{y}{2} \rceil$ disjoint pairs $\langle i_1, j_1 \rangle$, $\langle i_2, j_2 \rangle$, ..., $\langle i_{\lceil y/2 \rceil}, j_{\lceil y/2 \rceil} \rangle$ such that $i_1 \neq i_2 \neq \ldots \neq i_{\lceil y/2 \rceil} \neq j_1 \neq j_2 \neq \cdots \neq j_{\lceil y/2 \rceil}$. Hence, all y inputs of the set A are assigned to a team, where no input is assigned twice in a team. ■

Claim C.3 *When the reducer capacity $q = 3$, the set B holds at most $x \leq y - 1$ inputs.*

Proof. Since a pair of inputs of the set A requires at most $q - 1$ capacity of a reducer and each team holds all the inputs of the set A, an input from the set B can be assigned to all the reducers of the team. In this manner, all the inputs of the set A are also paired with an input of the set B. Since there are $y - 1$ teams and each team is assigned an input of the set B, the set B can hold at most $x \leq y - 1$ inputs. ■

Theorem C.4 *Algorithm 7A assigns each pair of the given m inputs to at least one reducer in common.*

Proof. We have $(y-1)\lceil \frac{y}{2} \rceil$ pairs of inputs of the set A of size $q-1$, and there are the same number of reducers; hence, each reducer can hold one input pair. Further, the remaining capacity of all the reducers of each team can be used to assign an input of B. Hence, all the inputs of A are paired with every other input and every input of B (as we proved in Claims C.2 and C.3). Following the fact that the inputs of the set A are paired with all the m inputs, the inputs of the set B is also paired by following a similar procedure on them. Thus, Algorithm 7A assigns each pair of the given m inputs to at least one reducer in common. ■

C.5.2 Correctness of Algorithm 7B

We show that every pair of inputs is assigned to reducers. Specifically, Algorithm 7B satisfies two claims, as follows:

Claim C.5 *Pairs of derived inputs $\langle i, j\rangle$ and $\langle i', j'\rangle$, where $i \neq i'$ or $j \neq j'$, are assigned to a team.*

Claim C.6 *All the given m inputs appear exactly once in each team.*

We do not prove Claims C.5 and C.6. Note that Claim C.5 follows Claims C.1, where Claims C.1 shows that all the pairs of inputs of the set A (in case $q = 3$) and all the pairs of derived inputs of the set A (in case $q > 3$) $\langle i, j\rangle$ and $\langle i', j'\rangle$, where $i \neq i'$ or $j \neq j'$ are assigned to a team. Also, Claim C.6 follows Claim C.2, where Claim C.2 shows that all the inputs of the set A appear in each team only once, while in case of Algorithm 7B the set A is considered as a set of m inputs.

Theorem C.7 *Algorithm 7B assigns each pair of the given m inputs to at least one reducer in common.*

Proof. Since there are the same number of pairs of the derived inputs and the number of reducers, it is possible to assign one pair to each reducer that results in all-pairs of the m inputs. ■

C.6 The *First Extension to the AU method* (Algorithm 8)

Theorem 5.12 (The communication cost obtained using Algorithm 8) *Algorithm 8 requires at most $p(p+1) + z$ reducers, where $z = \frac{2l^2(p+1)^2}{q^2}$, and results in at most $qp(p+1) + z'$ communication cost, where $z' = \frac{2l^2(p+1)^2}{q}$, q is the reducer capacity, and p is the nearest prime number to q.*

When $l = q - p$ equals to one, we have provided an extension of the *AU method* in Section 5.2.3, and in this case, we have an optimum mapping schema for q and $m = q^2 + q + 1$ inputs.

Proof. In case of $l > 1$, a single reducer cannot be used to assign all the inputs of the set B. Since Algorithm 2 is based on the *AU method*, Algorithm 7A, and Algorithm 7B, we always use at most $p(p+1) + z$ reducers, where z ($= \frac{2l^2(p+1)^2}{q^2}$) reducers are used to assign each pair of inputs of the set B based on Algorithms 7A or 7B (for the value of z, the reader may refer to Theorem 11 of the technical report [7]). Thus, the communication

Algorithm 8: The *first extension to the AU method.*

```
Inputs: $m$: total number of unit-sized inputs, $q$: the reducer capacity.
Variables:
$A$: A set $A$, where the total inputs in the set $A$ is $y = p^2$, where $p$ is a near most prime
  number to $q$ such that $p + l = q$
$B$: A set $B$, where the total inputs in the $B$ is $x \leq m - y$
$Bin[i, j]$ : represents teams of bin, where index $i$ indicates $i^{th}$ bin and index $j$ indicates
  $j^{th}$ reducer in $i^{th}$ team. There are $p + 1$ teams of $p$ bins (each of size $p$), in each team.
$Team[i, j]$ : represents teams of reducers, where index $i$ indicates $i^{th}$ team and index $j$
  indicates $j^{th}$ reducer in $i^{th}$ team. There are $p + 1$ teams of $p$ reducers (each of
  capacity $q$) in each team.
$groupB[]$ : represents disjoint groups of inputs of the set $B$, where $groupB[i]$
  indicates $i^{th}$ group of $\lceil \frac{x}{q-p} \rceil$ inputs of the set $B$.
1 Function $Assignment(A, B)$ begin
2     $Bin[] \leftarrow$ The AU method$(y, p)$
3     foreach $(i, j) \in [1, p + 1] \times [1, p]$ do $Team[i, j] \leftarrow Bin[i, j]$ ;
4     for $i \leftarrow 1$ to $x$ do $groupB[i] \leftarrow \langle i, i + 1 \ldots, i + \frac{x}{q-p} \rangle, i \leftarrow i + \frac{x}{q-p}$ ;
5     foreach $(i, j) \in [1, p + 1] \times [1, p]$ do $Team[i, j] \leftarrow groupB[i]$ ;
6     if $q$ is an odd number then Algorithm 7A$(x, q)$ ;
7     else if $q$ is an even number then Algorithm 7B$(x, q)$ ;
```

cost is at most $qp(p + 1) + z'$, where z' ($= \frac{2l^2(p+1)^2}{q}$) is the maximum communication cost required by Algorithm 7A or 7B for assigning $(p + 1)l$ inputs of the set B. ■

C.6.1 Correctness of Algorithm 8

The correctness shows that all-pairs of inputs are assigned to reducers. Specifically, we show that each pair of inputs of the set A is assigned to $p(p + 1)$ reducers that use only p capacity of each reducer (Claims C.8 and C.9). Then, we prove that the set B holds $x \leq m - p^2$ inputs. At last, we conclude that Algorithm 8 assigns each pair of inputs to reducers.

Claim C.8 *All the inputs of the set A are assigned to $p(p+1)$ reducers, and the assignment of the inputs of the set A uses only p capacity of each reducer.*

Claim C.9 *All the inputs of the set A appear in each team exactly once.*

We are not proving Claims C.8 and C.9 here. Claims C.8 and C.9 follow the correctness of the *AU method*; hence, all the inputs of the set A are placed to $p + 1$ teams of p bins (each of size q) in each team, and the assignment of each such bin only uses p capacity of each reducer. Further two bins cannot be assigned to a reducer because $2 \times p > q$. Claim C.9 also follows the correctness of the *AU method*, and hence, all the inputs of the set A appear only once in each team.

Claim C.10 *When the reducer capacity is q, the set B holds $x \leq m - p^2$ inputs, where p is the nearest prime number to q.*

Proof. There are $p + 1$ teams of p reducers in each team, and inputs of the set A use $q - p$ capacity of each of the reducers. Hence, each reducer can hold $q - p$ additional unit-sized (almost identical-sized) inputs. Since inputs of the set A appear in each team (Claim C.9), an assignment of $q - p$ additional unit-sized inputs to all the reducers of a team provides pairs of all the inputs of the set A with additional inputs. In this manner, $p+1$ teams, which hold p^2 inputs of the set A, can hold at most $(p + 1) \times (q - p)$ additional inputs. Since $p^2 < m \leq p^2 + (p+1) \times (q - p)$, the set B can hold $x \leq m - p^2$ inputs. ■

Theorem C.11 *Algorithm 8 assigns each pair of inputs to reducers.*

We are not proving Theorem C.11 here. The proof of Theorem C.11 considers the fact that all the inputs of the set A are paired with each other using the *AU method*, and they are also paired with all the remaining inputs of the set B. Further, inputs of the set B will be paired with each other by using Algorithm 7A or 7B (Theorems C.4 or C.7).

C.7 The *Second Extension to the AU method* (Algorithm 9)

Algorithm 9: The *second extension to the AU method.*

```
  Inputs: $m$: total number of unit-sized inputs.
  $q$: the reducer capacity.
1 Function $Assignment(m)$ begin
2     $bottom\text{-}up\_tree(m)$, $assignment\_tree(root\_node\_of\_bottom\text{-}up\_tree)$
```

Theorem 5.16 (The communication cost obtained using Algorithm 9) *Algorithm 9 requires at most $q \times (q(q+1))^{l-1}$ reducers and results in at most $q^2 \times (q(q+1))^{l-1}$ communication cost.*

Proof. For a given $m = q^l$, $l > 2$, the assignment tree has height l (Lemma 5.17), and (according to Algorithm 10(c)) l^{th} level has $q \times (q(q+1))^{l-1}$ reducers providing an assignment of each pairs of inputs. Hence, Algorithm 10(c) uses $q(q(q+1))^{l-1}$ reducers, and the communication cost is at most $q^2 \times (q(q+1))^{l-1}$. ■

C.8 A Theorem related to A Big Input

Theorem 5.17 (Upper bounds from algorithms) *For a list of m inputs where a big input, i, of size $\frac{q}{2} < w_i < q$ and for a given reducer capacity q, $q < s' < s$, an input is replicated to at most $m-1$ reducers for the A2A mapping schema problem, and the number of reducers and the communication cost are at most $m-1+\frac{8s^2}{q^2}$ and $(m-1)q+\frac{4s^2}{q}$, respectively, where s' is the sum of all the input sizes except the size of the big input and s is the sum of all the input sizes.*

Proof. The big input i can share a reducer with inputs whose sum of the sizes is at most $q-w_i$. In order to assign the input i with all the remaining $m-1$ small inputs, it is required to assign a sublist of $m-1$ inputs whose sum of the sizes is at most $q-w_i$. If all the small inputs are of size almost $q-w_i$, then a reducer can hold the big input and one of the small inputs. Hence, the big input is required to be sent to at most $m-1$ reducers that results in at most $(m-1)q$ communication cost.

Also, each pair of all the small inputs is assigned to reducers (by first placing them to bins of size $\frac{q}{2}$ using FFD or BFD bin-packing algorithm). The assignment of all the small inputs results in at most $\frac{8s'^2}{q^2} < \frac{8s^2}{q^2}$ reducers and at most $\frac{4s'^2}{q} < \frac{4s^2}{q}$ communication cost (Theorem 5.3). Thus, the number of reducers is at most $m-1+\frac{8s^2}{q^2}$ and the communication cost is at most $(m-1)q+\frac{4s^2}{q}$. ■

C.9 Theorems related to the *X2Y Mapping Schema Problem*

Theorem 5.18 (Lower bounds on the communication cost and number of reducers) *For a list X of m inputs, a list Y of n inputs, and a given reducer capacity q, the communication cost and the number of reducers, for the X2Y mapping schema problem, are at least $\frac{2\cdot sum_x\cdot sum_y}{q}$ and $\frac{2\cdot sum_x\cdot sum_y}{q^2}$, respectively.*

Proof. Since an input i of the list X and an input j of the list Y are replicated to at least $\frac{sum_y}{q}$ and $\frac{sum_x}{q}$ reducers, respectively, the communication cost for the inputs i and j are $w_i \times \frac{sum_y}{q}$ and $w_j \times \frac{sum_x}{q}$, respectively. Hence, the communication cost will be at least $\sum_{i=1}^{m} w_i\frac{sum_y}{q} + \sum_{j=1}^{n} w_j\frac{sum_x}{q} = \frac{2\cdot sum_x\cdot sum_y}{q}$.

Since the total number of bits to be assigned to reducers is at least $\frac{2\cdot sum_x\cdot sum_y}{q}$ and a reducer can hold inputs whose sum of the sizes is at most q, the number of reducers must be at least $\frac{2\cdot sum_x\cdot sum_y}{q^2}$. ■

Theorem 5.19 (Upper bounds from the algorithm) *For a bin size b, a given reducer* *capacity $q = 2b$, and with each input of lists X and Y being of size at most b, the number of reducers and the communication cost, for the X2Y mapping schema problem, are at most $\frac{4 \cdot sum_x \cdot sum_y}{b^2}$, and at most $\frac{4 \cdot sum_x \cdot sum_y}{b}$, respectively, where sum_x is the sum of input sizes of the list X, and sum_y is the sum of input sizes of the list Y.*

Proof. A bin i can hold inputs whose sum of the sizes is at most b. Hence, it is required to divide inputs of the lists X and Y into at least $\frac{sum_x}{b}$ and $\frac{sum_y}{b}$ bins, respectively. Since the FFD or BFD bin-packing algorithm ensures that all the bins (except only one bin) are at least half-full, each bin of size b has at least inputs whose sum of the sizes is at least $\frac{b}{2}$. Thus, all the inputs of the lists X and Y can be placed in at most $\frac{sum_x}{b/2}$ and $\frac{sum_y}{b/2}$ bins of size b, respectively.

Let x ($=\frac{2 \cdot sum_x}{b}$) and y ($=\frac{2 \cdot sum_y}{b}$) bins are used to place inputs of the lists X and Y, respectively. Since each bin is considered as a single input, we can assign each of the x bins with each of the y bins at reducers, and hence, we require at most $\frac{4 \cdot sum_x \cdot sum_y}{b^2}$ reducers. Since each bin that is containing inputs of the list X (resp. Y) is replicated to at most $\frac{2 \cdot sum_y}{b}$ (resp. at most $\frac{2 \cdot sum_x}{b}$) reducers, the replication of individual inputs of the list X (resp. Y) is at most $\frac{2 \cdot sum_y}{b}$ (resp. at most $\frac{2 \cdot sum_x}{b}$) and the communication cost is at most $\sum_{1 \leq i \leq m} w_i \times \frac{2 \cdot sum_y}{b} + \sum_{1 \leq j \leq n} w_j \times \frac{2 \cdot sum_x}{b} = \frac{4 \cdot sum_x \cdot sum_y}{b}$. ■

Appendix D

Proofs of Theorems related to Meta-MapReduce (Chapter 6)

Theorem 6.1 (The communication cost for join of two relations) *Using Meta-MapReduce, the communication cost for the problem of join of two relations is at most $2nc + h(c+w)$ bits, where n is the number of tuples in each relation, c is the maximum size of a value of the joining attribute, h is the number of tuples that actually join, and w is the maximum required memory for a tuple.*

Proof. Since the maximum size of a value of the joining attribute, which works as a metadata in the problem of join, is c and there are n tuples in each relation, users have to send at most $2nc$ bits to the site of mappers-reducers. Further, tuples that join at the reduce phase have to be transferred from the map phase to the reduce phase and then from the user's site to the reduce phase. Since there are at most h tuples join and the maximum size of a tuple is w, we need to transfer at most hc and at most hw bits from the map phase to the reduce phase and from the user's site to the reduce phase, respectively. Hence, the total communication cost is at most $2nc + h(c + w)$ bits. ■

Theorem 6.2 (The communication cost for skew join) *Using Meta-MapReduce, the communication cost for the problem of skew join of two relations is at most $2nc + rh(c+w)$ bits, where n is the number of tuples in each relation, c is the maximum size of a value of the joining attribute, r is the replication rate, h is the number of distinct tuples that actually join, and w is the maximum required memory for a tuple.*

Proof. From the user's site to the site of mappers-reducers, at most $2nc$ bits are required to move (according to Theorem 6.1). Since at most h distinct tuples join and these tuples are replicated to r reducers, at most rhc bits are required to transfer from the map phase to the reduce phase. Further, h tuples of size at most w to be transferred from the map phase to the reduce phase, and hence, at most rhw bits are assigned to reducers. Thus, the total communication cost is at most $2nc + rh(c + w)$ bits. ■

Theorem 6.3 (The communication cost when joining attributes are large) *Using Meta-MapReduce for the problem of join where values of joining attributes are large, the communication cost for the problem of join of two relations is at most $6n \cdot log\ m + h(c+w)$ bits, where n is the number of tuples in each relation, m is the maximal number of tuples in two relations, h is the number of tuples that actually join, and w is the maximum required memory for a tuple.*

Proof. The maximal number of tuples having different values of a joining attribute in all relations is m, which is upper bounded by $2n$; hence, a mapping of hash function of m values into m^3 values will result in a unique hash value for every of the m keys with a high probability. Thus, we use at most $3 \cdot log\ m$ bits for metadata of a single value, and hence, at most $6n \cdot log\ m$ bits are required to move metadata from the user's site to the site of mappers-reducers. Since there are at most h tuples join and the maximum size of a tuple is w, we need to transfer at most hc and at most hw bits from the map phase to the reduce phase and from the user's site to the reduce phase, respectively. Hence, the total communication cost is at most $6n \cdot log\ m + h(c + w)$ bits. ■

Theorem 6.4 (The communication cost for k relations and when joining attributes are large) *Using Meta-MapReduce for the problem of join where values of joining attributes are large, the communication cost for the problem of join of k relations, each of the relations with n tuples, is at most $3knp \cdot log\ m + h(c + w)$ bits, where n is the number of tuples in each relation, m is the maximal number of tuples in k relations, p is the maximum number of dominating attributes in a relation, h is the number of tuples that actually join, and w is the maximum required memory for a tuple.*

Proof. According to Theorem 6.3, at most $3 \cdot log\ m$ bits for metadata are required for a single value; hence, at most $3knp \cdot log\ m$ bits are required to move metadata from the user's site to the site of mappers-reducers. Since at most h tuples join and the maximum size of a tuple is w, at most hc and at most hw bits from the map phase to the reduce phase and from the user's site to the reduce phase, respectively, are transferred. Hence, the total communication cost is at most $3knp \cdot log\ m + h(c + w)$ bits. ■

Appendix E

Proof of Theorems related to Interval Join (Chapter 7)

E.1 Proof of Theorems and Algorithm related to Unit-Length and Equally Spaced Intervals

Theorem 7.1 (Minimum replication rate) *Let there be two relations: X and Y, each of them containing n unit-length and equally spaced intervals in the range $[0, k)$, and let q be the reducer size. The replication rate for joining each interval of the relation X with all its overlapping intervals of the relation Y, is at least $\frac{2n}{qk}$.*

Proof. A reducer i can have $q_i \leq q$ intervals, and it can cover at most $\left(\frac{q_i}{2}\right)^2$ pairs of intervals, one interval from X and the other from Y. Since an interval that does not have starting-point before 1 and after $k-1$ has at least $\frac{2n}{k}+1$ overlapping intervals, there are at least $\left(n-\frac{2n}{k}\right)\left(\frac{2n}{k}+1\right)$ pairs of overlapping intervals. In addition, there are at least $\frac{2n^2}{k^2}$ pairs of overlapping intervals, where at least one member of the pair starts before 1 or after $k-1$. Thus, we have at least

$$\left(n-\frac{2n}{k}\right)\left(\frac{2n}{k}+1\right)+\frac{2n^2}{k^2}=\frac{2n}{k}\left(n-\frac{n}{k}+\frac{k}{2}-1\right)$$

pairs of overlapping intervals.

Consider that there are z reducers, and we can set the following equation

$$\sum_{1\leq i\leq z}\frac{q_i^2}{4}\geq\frac{2n}{k}\left(n-\frac{n}{k}+\frac{k}{2}-1\right)$$

In order to bound the replication rate, we divide both the sides of the above equation by the total number of inputs, *i.e.*, $2n$,

$$\sum_{1\le i\le z} \frac{\frac{q_i^2}{4}}{2n} \ge \frac{1}{k}\Big(n - \frac{n}{k} + \frac{k}{2} - 1\Big)$$

$$\sum_{1\le i\le z} \frac{q_i^2}{8n} \ge \frac{n}{2k}$$

Now, we manipulate the above equation to have a lower bound on the replication rate by arranging the terms of the above equation so that the left side becomes the replication rate. We separate a factor q_i from q_i^2, and one q_i can be replaced by the upper bound on the reducer size, q. Note that the equation still holds.

$$\sum_{1\le i\le z} q\frac{q_i}{8n} \ge \frac{n}{2k}$$

By moving some terms from the left side of the above equation, we can get a lower bound on the replication rate, $r = \sum_{1\le i\le z} \frac{q_i}{2n}$, as follows:

$$\sum_{1\le i\le z} \frac{q_i}{2n} \ge \frac{2n}{qk}.$$

■

Theorem 7.2 (Minimum communication cost) *Let there be two relations: X and Y, each of them containing n unit-length and equally spaced intervals in the range $[0, k)$, and let q be the reducer size. The communication cost for joining of each interval of the relation X with all its overlapping intervals of the relation Y is at least $\frac{4n^2}{qk}$.*

Proof. Since an interval, say i, of unit-length is replicated to at least $\frac{2n}{qk}$ reducers, the communication cost for the interval i is at least $\frac{2n}{qk}$. Hence, the total communication cost for all $2n$ intervals will be at least $\frac{2n}{qk} \cdot 2n$. ■

Explaining pseudocode of Algorithm 10. A mapper takes an interval $x_i \in X$ (line 2) and produces $\langle key, value\rangle$ pairs (line 5). The function $\mathit{find\_blocks}(x_i)$ finds a set, Z, of blocks where an interval x_i crosses (line 4), and for each member of the set, Z, the map function generates a $\langle key, x_i\rangle$ pair (line 5). Note that the key represents a block where the interval x_i exists, and the number of $\langle key, value\rangle$ pairs for the interval x_i equals to the cardinality of the set Z.

Also, a mapper processes an interval $y_i \in Y$ (line 6) and produces two $\langle key, value\rangle$ pairs (line 9), where the first pair and the second pair are corresponding to a block where y_i

Algorithm 10: 2-way interval join algorithm for overlapping intervals.

```
Inputs: $X$ and $Y$: two relations, each with $n$ intervals.
Variables: $w$: The length of a block $w = \frac{q-c}{4c}$, where $c = \frac{n}{k}$;
$P$: The number of blocks and reducers;
$Z$: A set.
1  Partition the time-range into $P$ blocks, each of length $w$
2  Function $Map\_for\_X(x_i \in X)$ begin
3      $Z \leftarrow \emptyset$
4      $Z \leftarrow find\_blocks(x_i)$
5      For each member, $u$, of the set $Z$ $emit\langle u, x_i\rangle$
6  Function $Map\_for\_Y(y_i \in Y)$ begin
7      $sp \leftarrow starting\_point(y_i)$
8      $ep \leftarrow ending\_point(y_i)$
9      $emit\langle sp, y_i\rangle$, $emit\langle ep, y_i\rangle$
10 Function $reduce(\langle key, list\_of\_values[]\rangle)$ begin
11     for $j \leftarrow 1$ to $P$ do
12         Reducer $i$ receives $\langle i, list\_of\_values[x_a, x_b, \ldots, y_a, y_b, \ldots]\rangle$
13         Perform interval join over overlapping intervals
```

has the starting-point and the ending-point, respectively (line 8). The $value$ represents the interval y_i itself. In the reduce phase, a reducer i fetches all the intervals of the relations X and Y that have a key i (line 12) and provides the final outputs, line 13.

Now, we show the correctness of Algorithm 10. Before that we provide two definitions, as follows:

Definition 1 *Post-intervals of an interval of $i \in X$: Consider that an interval $i \in X$ is in a block p. An interval j of the relation X that has its starting-point after the starting-point of the interval i in the block p, is called a* post-interval *of the interval i.*

Definition 2 *Pre-intervals of an interval of $i \in X$: Consider that an interval $i \in X$ is in a block p. An interval j of the relation X that has started before the interval i and has either its ending-points in the block p or crosses the block p, is called a* pre-interval *of the interval i.*

Theorem 7.3 (Algorithm correctness) *Let there be two relations: X and Y, each of them containing n unit-length and equally spaced intervals in the range $[0, k)$. Let w be the length of a block, and let $q = 4wc + c$ is the reducer size, where $c = \frac{n}{k}$. Algorithm 10 assigns each pair of overlapping intervals to at least one reducer in common.*

Proof. We consider two cases such as $w \geq 1$ and $w < 1$ and prove two arguments for both the cases, as follows:

1. A reducer is big enough to hold all the intervals of a block.
2. Each pair of overlapping intervals is assigned to at least one reducer.

Case 1: $w \geq 1$**.** First we count the number of intervals that exist in a block. Since the spacing between adjacent intervals is $\frac{k}{n}$ and the length of a block is w, there are at most $\frac{w}{k/n}$ points where an interval can start or end. Since at a single point an interval can start as well as an interval can end, there are at most $\frac{4wn}{k}$ intervals of both the relations in a block. Hence, the reducer can hold all the intervals of a block.

Case 2: $w < 1$**.** First, we show that a reducer is big enough to hold all the intervals of a block. Since the spacing between adjacent intervals is $\frac{k}{n}$ and the length of a block is w, a block can have at most $\frac{2wn}{k}$ intervals of the relation Y. The assignment of all the intervals of the relation Y that exist in a block occupies $\frac{2wn}{k}$ space of a reducer. Note that in a block of length w, there are at most $\frac{wn}{k}$ post-intervals of the interval i and at most $\frac{n}{k}$ pre-intervals of the interval i. Thus, we can fill the remaining space of the reducer, *i*.e., $\frac{2wn}{k} + c$, by the interval i, post-intervals of i that lie in the block p, and pre-intervals of i that lie in block p to a single reducer. Thus, the reducer can hold all the intervals of a block.

Now, we show how a pair of overlapping intervals, say x_i and y_i, is assigned to a reducer. The analysis is same as given in [31]. Consider an interval $y_i \in Y$ having its starting-point or ending-point in a block p_i. This interval is assigned to a reducer corresponding to the block p_i. The interval x_i is replicated to all the reducers corresponding to the blocks in which x_i exists. Since x_i overlaps with y_i, the interval x_i is also assigned to the same reducer where y_i has already assigned. The reducer corresponding to the block p_i (where x_i and y_i exist) will provide the desired results. ■

Theorem 7.4 (Maximum replication rate) *Let there be two relations: X and Y, each of them containing n unit-length and equally spaced intervals in the range $[0, k)$. Let w be the length of a block, and let $q = 4wc + c < 2n$ is the reducer size, where $c = \frac{n}{k}$. The replication of an interval, for joining each interval of the relation X with all its overlapping intervals of the relation Y, is (i) at most 2 for $w \geq 1$ and (ii) at most $\frac{4c}{q-c}$ for $w < 1$.*

Proof. First, we consider the case of $k > w \geq 1$. Here, an interval of the relation X crosses at most two blocks, and hence, the interval of X can be replicated to at most 2 reducers, according to its starting-point and ending-point. However, an interval of the relation Y is sent to at most two reducers in any case, according to its starting-point and ending-point. Hence, the replication of an interval is at most 2.

For $w < 1$, according to Algorithm 10, an interval crosses at most $w + 1$ blocks, and hence, an interval of the relation X is required to be sent to at most $w+1$ reducers. Also, an interval of the relation Y is replicated to at most two reducers; thus, the average replication rate is $\frac{w+3}{2}$. In this case, we have $q = 4wc+c < 2n$, and we replace w by $\frac{1}{r}$ that provides us

$r = \frac{4c}{q-c}$. Therefore, an interval is required to be sent to at most $\frac{4c}{q-c}$ reducers when $w < 1$ and $q = 4wc + c$, where $c = \frac{n}{k}$. ■

Theorem 7.5 (Maximum communication cost) *Let there be two relations: X and Y, each of them containing n unit-length and equally spaced intervals in the range $[0, k)$. Let w be the length of a block, and let $q = 4wc + c < 2n$ is the reducer size, where $c = \frac{n}{k}$. The communication cost for joining of each interval of the relation X with all its overlapping intervals of the relation Y is (i) at most $4n$ for $w \geq 1$ and (ii) at most $\frac{8nc}{q-c}$ for $w < 1$.*

Proof. In the case of $w \geq 1$, since an interval is replicated to at most 2 reducers, the communication cost for this interval is at most 2. Hence, the total communication cost for all $2n$ intervals is at most $4n$.

In the case of $w < 1$, since an interval is replicated to at most $\frac{4c}{q-c}$ reducers, the communication cost for this interval is at most $\frac{4c}{q-c}$. Hence, the total communication cost for all $2n$ intervals is at most $\frac{4c}{q-c} \cdot 2n$. ■

E.2 Proof of Theorems and Algorithm related to Variable-Length and Equally Spaced Intervals

Theorem 7.6 (Minimum replication rate) *Let there be two relations: X containing n small and equally spaced intervals and Y containing n big and equally spaced intervals, and let q be the reducer size. Let s be the spacing between every two successive intervals, and let l_{min} be the length of the smallest interval. The replication rate for joining of each interval of the relation X with all its overlapping intervals of the relation Y is at least $\frac{2l_{min}}{qs}$.*

Proof. A reducer i can have $q_i \leq q$ intervals, and it can cover at most $\left(\frac{q_i}{2}\right)^2$ pairs of intervals, one interval from X and the other from Y. Since an interval that starts after l_{min} has at least $\frac{2l_{min}}{s} + 1$ overlapping intervals, there are at least $\left(n - \frac{l_{min}}{s}\right)\left(\frac{2l_{min}}{s} + 1\right)$ pairs of overlapping intervals. In addition, there are at least $\frac{l_{min}^2}{s^2}$ pairs of overlapping intervals, where at least one member of the pair starts before l_{min} time. Thus, we have at least

$$\left(n - \frac{l_{min}}{s}\right)\left(\frac{2l_{min}}{s} + 1\right) + \frac{l_{min}^2}{s^2} =$$

$$\frac{2n}{s}\left(l_{min} - \frac{l_{min}^2}{2ns} + \frac{s}{2l_{min}} - \frac{l_{min}}{2n}\right)$$

pairs of overlapping intervals.

Consider that there are z reducers, and we can set the following equation

$$\sum_{1\leq i\leq z} \frac{q_i^2}{4} \geq \frac{2n}{s}\Big(l_{min} - \frac{l_{min}^2}{2ns} + \frac{s}{2l_{min}} - \frac{l_{min}}{2n}\Big)$$

In order to bound the replication rate, we divide both the sides of the above equation by the total number of inputs, *i.e.*, $2n$,

$$\sum_{1\leq i\leq z} \frac{\frac{q_i^2}{4}}{2n} \geq \frac{1}{s}\Big(l_{min} - \frac{l_{min}^2}{2ns} + \frac{s}{2l_{min}} - \frac{l_{min}}{2n}\Big)$$

$$\sum_{1\leq i\leq z} \frac{q_i^2}{8n} \geq \frac{l_{min}}{2s}$$

Now, we manipulate the above equation to have a lower bound on the replication rate by arranging the terms of the above equation so that the left side becomes the replication rate. We separate a factor q_i from q_i^2, and one q_i can be replaced by the upper bound on the reducer size, q. Note that the equation still holds to be true.

$$\sum_{1\leq i\leq z} q\frac{q_i}{8n} \geq \frac{l_{min}}{2s}$$

By moving some terms from the left side of the above equation, we can get a lower bound on the replication rate, $r = \sum_{1\leq i\leq z} \frac{q_i}{2n}$, as follows:

$$\sum_{1\leq i\leq z} \frac{q_i}{2n} \geq \frac{2l_{min}}{qs}.$$

■

Theorem 7.7 (Minimum communication cost) *Let there be two relations: X containing n small and equally spaced intervals and Y containing n big and equally spaced intervals, and let q be the reducer size. Let s be the spacing between every two successive intervals, and let l_{min} be the length of the smallest interval. The communication cost for joining of each interval of the relation X with all its overlapping intervals of the relation Y is at least $\frac{4nl_{min}}{qs}$.*

Proof. Since an interval, say i, is replicated to at least $\frac{2l_{min}}{qs}$ reducers, the communication cost for the interval i is at least $\frac{2l_{min}}{qs}$. Hence, the total communication cost for all $2n$ intervals will be at least $\frac{2l_{min}}{s} \cdot 2n$. ■

Theorem 7.9 (Maximum replication rate) *Let there be two relations: X containing n small and equally spaced intervals and Y containing n big and equally spaced intervals. Let s be the spacing between every two successive intervals, let w be the length of a block, and let l_{min} be the length of the smallest interval. Let $q = 4wc + c$ is the reducer size, where $c = \frac{l_{min}}{s}$. The replication rate for joining of each interval of the relation X with all its overlapping intervals of the relation Y is (i) at most 2 for $w \geq l_{min}$ and (ii) at most $\frac{4c}{q-c}$ for $w < l_{min}$.*

Proof. First, we consider the case of $k > w \geq l_{min}$. Here, an interval of the relation X crosses at most only two blocks, and hence, the interval of X can be replicated to at most 2 reducers, according to its starting-point and ending-point. However, an interval of the relation Y is sent to at most two reducers in any case, according to its starting-point and ending-point. Hence, the replication of an interval is at most 2.

For $w < l_{min}$, according to Algorithm 10(a), an interval crosses at most $w + 1$ blocks, and hence, an interval of the relation X is required to be sent to at most $w + 1$ reducers. Also, an interval of the relation Y is replicated to at most two reducers; thus, the average replication rate is $\frac{w+3}{2}$. In this case, we have $q = 4wc + c$, and we replace w by $\frac{1}{r}$ that provides us $r = \frac{4c}{q-c}$. Therefore, an interval is required to be sent to at most $\frac{4c}{q-c}$ reducers when $w < l_{min}$ and $q = 4wc + c$, where $c = \frac{l_{min}}{s}$. ■

Theorem 7.10 (Maximum communication cost) *Let there be two relations: X containing n small and equally spaced intervals and Y containing n big and equally spaced intervals. Let s be the spacing between every two successive intervals, let w be the length of a block, and let l_{min} be the length of the smallest interval. Let $q = 4wc + c$ is the reducer size, where $c = \frac{l_{min}}{s}$. The communication cost for joining of each interval of the relation X with all its overlapping intervals of the relation Y is (i) at most $4n$ for $w \geq l_{min}$ and (ii) at most $\frac{8nc}{q-c}$ for $w < l_{min}$.*

Proof. In the case of $w \geq l_{min}$, since an interval is replicated to at most 2 reducers, the communication cost for this interval is at most 2. Hence, the total communication cost for all $2n$ intervals is at most $4n$.

In the case of $w < l_{min}$, since an interval is replicated to at most $\frac{4c}{q-c}$ reducers, the communication cost for this interval is at most $\frac{4c}{q-c}$. Hence, the total communication cost for all $2n$ intervals is at most $\frac{4c}{q-c} \cdot 2n$. ■

E.2.1 Proof of the Theorem related to the General Case

Theorem 7.12 (Algorithm correctness) *Let there be two relations: X and Y, each of them containing n intervals. Let S be the total length of all the intervals in one relation, let w*

be the length of a block, let T be the length of time in which all intervals exist, and let $q = \frac{3nw+S}{T}$ is the reducer size. Algorithm 10(c) assigns each pair of overlapping intervals to at least one reducer in common.

Proof. We prove two arguments, as follows:

1. A reducer is big enough to hold all the intervals of a block.
2. Each pair of overlapping intervals is assigned to at least one reducer.

First, we show that a reducer is big enough to hold all the intervals of a block. Following Algorithm 10(c), each of the n intervals of the relation Y is sent to at most two reducers. Since there are $\frac{T}{w}$ reducers, a reducer receives $\frac{2nw}{T}$ inputs from Y in average. Since the length of all the intervals of the relation X is S, the average length of intervals is $\frac{S}{n}$. Following Algorithm 10(c), an interval of X is sent to $1 + \frac{S}{nw}$ reducers. Since there are $\frac{T}{w}$ reducers, the reducer receives $(1 + \frac{S}{nw})\frac{nw}{T}$ inputs from X in average. Thus, a reducer receives at most $\frac{2nw}{T} + \frac{nw}{t}(1 + \frac{S}{nw}) = \frac{3nw+S}{T}$ inputs, which is equal to the given reducer size.

We can prove the second argument, *i.e.*, each pair of overlapping intervals is assigned to at least one reducer, in a way similar to Theorem 7.3. ■

Theorem 7.13 (Replication rate) *Let there be two relations: X and Y, each of them containing n intervals. Let S be the total length of all the intervals in one relation, let w be the length of a block, let T be the length of time in which all intervals exist, and let $q = \frac{3nw+S}{T}$ is the reducer size. The replication rate for joining each interval of the relation X with all its overlapping intervals of the relation Y is at most $\frac{3}{qT-S}\frac{S}{2}$.*

Proof. Since an interval of the relations X and Y is replicated to $1 + \frac{S}{nw}$ and 2 reducers, respectively. Hence, the average replication is $\frac{3}{2} + \frac{1}{2}\frac{S}{nw}$. We have $q = \frac{3nw+S}{T}$, by replacing $w = \frac{qT-S}{3n}$, we get the average replication rate equals to $\frac{3}{qT-S}\frac{S}{2}$. ■

Appendix F

Proof of Theorems related to Computing Marginals of a Data Cube (Chapter 8)

F.1 Related Work on Data Cube

There have been a number of papers that look at the problem of using MapReduce to compute marginals. Probably the closest work to what we present in Chapter 8 is in [100]. This paper expresses the goal of minimizing communication, and of partitioning the work among reducers. It does not, however, present concrete bounds or algorithms that meet or approach those bounds, as we will do here.

The paper [91] considers constructing a data cube using a nonassociative aggregation function and also examines how to deal with nonuniformity in the density of tuples in the cube. Like all the other papers mentioned, it deals with constructing the entire data cube using multiple rounds of MapReduce. We consider how to compute only the marginals of one order, using one round. We may assume that locally, at each reducer, higher-order marginals are computed by aggregating lower-order marginals for efficiency, but this method does not result in additional MapReduce rounds.

The paper [3] looks at using MapReduce to form a data cube from data stored in Bigtable. [74] and [107] are implementations of known algorithms in MapReduce. Finally, [109] talks about extending MapReduce to compute data cubes more efficiently.

F.2 Proof of Theorems

Theorem 8.4 *If $q = d^m$, then we can solve the problem of computing all k^{th}-order marginals of an n-dimensional data cube with $r = C(n, m, k)$.*

Proof. Each marginal in the set of $C(n, m, k)$ handles can be turned into a team of reducers, one for each of the d^{n-m} ways to fix the dimensions that are not in the handle. Each input gets sent to exactly one member of the team for each handle – the reducer that corresponds to fixed values that agree with the input. Thus, each input is sent to exactly $C(n, m, k)$ reducers. ■

Theorem 8.5 (Lower bound on the number of handles) *For an n-dimensional data cube, where the marginals are of size two, the minimum number of handles is $\lfloor \frac{n(n-1)}{6} \rfloor$.*

Proof. The number of marginals for an n-dimensional data cube where the marginals are of size two is $\binom{n}{2} = \frac{n(n-1)}{2}$. However, a handle of size three cannot cover more than three marginals, because there are only three subsets of size two contained in a set of size three. Thus, the minimum number of handles is $\lfloor \frac{n(n-1)}{6} \rfloor$. ■

Theorem 8.6 (Upper bound on the number of handles) *For an n-dimensional data cube, where $n = 3^i$, $i > 0$, and the marginals are of size two, the number of handles is bounded above by*

$$C(n, 3, 2) \leq \frac{n^2}{6}, n = 3^i, i > 0.$$

Proof. The proof is by induction on n.

Basis case. For $n = 3$, we show that $C(3, 3, 2) = 1$ is true, since we need only one handle consisting of all three dimensions. In addition, $1 < \frac{3^2}{6}$. Hence, Theorem 8.6 holds for $n = 3$.

Inductive step. Let $p = 3^{i-1}, i > 0$. Assume that the inductive hypothesis — $C(p, 3, 2) \leq \frac{p^2}{6}$ is true. Now, by following steps 1 and 2 of Algorithm 1, we can build a solution for a data cube of $n = 3p$ dimensions and show that $C(3p, 3, 2) \leq \frac{(3p)^2}{6}$. Recall that using step 1 of Algorithm 1, we construct p^2 handles. Using step 2 of Algorithm 1, we recursively construct handles for each group by applying the inductive hypothesis on a group.

We will show that in steps 1 and 2 of Algorithm 1, we constructed a set of handles for a data cube of $n = 3^i$, $i > 0$, dimensions so that all the marginals of size two are covered. It will therefore follow that $C(3^i, 3, 2) \leq (3^{i-1})^2 + 3 \times C(3^{i-1}, 3, 2)$, $i > 0$. By the inductive hypothesis, $C(p, 3, 2) \leq \frac{p^2}{6}$, where $p = 3^{i-1}$. By substituting the value of $C(p, 3, 2)$ in the inequality $C(3^i, 3, 2) \leq (3^{i-1})^2 + 3 \times C(3^{i-1}, 3, 2)$, we have $C(3^i, 3, 2) \leq (3^{i-1})^2 + 3 \times \frac{(3^{i-1})^2}{6} = \frac{(3^i)^2}{6}$. Hence, we constructed at most $\frac{n^2}{6}$ handles for a data cube of $n = 3^i$, $i > 0$, dimensions, where the marginals are of size two.

Now, we have to show that all the marginals of size two for a data cube of $n = 3^i$, $i > 0$, dimensions will be covered by the handles of size three, which were constructed in steps 1 and 2 of Algorithm 1. In order to show that we consider two cases based on the dimensions of the marginals, as follows:

a. *Two dimensions of the marginals are in two groups.* In this case, whatever two dimensions we select from two groups, there is a dimension of the third group such that the sum of the indexes of the three dimensions is 0 mod p, $p = 3^{i-1}$, $i > 0$, and we have created all the handles consisting of those three dimensions in step 1 of Algorithm 1. Hence, all the marginals having two dimensions in two groups are covered.

b. *Two dimensions of the marginals are in a group.* In this case, the marginals have any two dimensions in a group (for example, D_1D_2 is a marginal of a 9-dimensional data cube). In order to cover such marginals, we need handles that have any three dimensions of the same group, and by the inductive hypothesis, we have handles that cover marginals from a single group, in step 2 of Algorithm 1. Hence, all the marginals having two dimensions in a group are covered.

Thus, we cover all the marginals of size two of a data cube of $n = 3^i$, $i > 0$ dimensions, using $C(n, 3, 2) \leq \frac{n^2}{6}$ handles of size three. ■

Theorem 8.8 (Upper bound on the number of handles) *For an n-dimensional data cube, where the marginals are of size two, the number of handles is bounded above by*

$$C(n, 3, 2) \leq \frac{(n-1)^2}{4}$$

where $n \geq 5$ and n is odd[1].

Proof. The proof is by induction on n.

Basis case. For $n = 5$, we show that $C(n, 3, 2) = C(5, 3, 2) = 4$ is true. Let the five dimensions be $D_1D_2D_3D_4D_5$. When $n = 10$, there are ten marginals of size two. Then, four handles such as $\langle D_1, D_4, D_5\rangle$, $\langle D_2, D_4, D_5\rangle$, $\langle D_3, D_4, D_5\rangle$, and $\langle D_1, D_2, D_3\rangle$ cover all the 10 marginals. Hence, Theorem 8.9 holds for $n = 5$.

Inductive step. Assume that the inductive hypothesis — $C(n-2, 3, 2) \leq \frac{(n-3)^2}{4}$ is true. Now, by following steps 1 and 2 of Algorithm 2, we can build a solution to an n-dimensional data cube and show that $C(n, 3, 2) \leq \frac{(n-1)^2}{4}$. In step 1, we created $n-2$ handles, and in step 2, we recursively created handles for the first $n-2$ dimensions by applying the inductive hypothesis.

We will show that in steps 1 and 2 of Algorithm 2, we created handles for an n-dimensional data cube so that all the marginals of size two are covered. It will therefore follow that $C(n, 3, 2) \leq n - 2 + C(n-2, 3, 2)$. By the inductive hypothesis, $C(n-2, 3, 2) \leq \frac{(n-3)^2}{4}$. By substituting the value of $C(n-2, 3, 2)$ in the inequality $C(n, 3, 2) \leq n - 2 + C(n-2, 3, 2)$, we have $C(n, 3, 2) \leq n - 2 + \frac{(n-3)^2}{4} = \frac{(n-1)^2}{4}$. Hence,

[1]When n is even, the number of handles is bounded above by $C(n, 3, 2) \leq \lceil\frac{(n-1)^2}{4}\rceil$, where $n \geq 8$.

we create at most $\frac{(n-1)^2}{4}$ handles for an n-dimensional data cube, where the marginals are of size two.

Now, we have to show that all the marginals of size two will be covered by the handles of size three that were created in steps 1 and 2. In order to show that we consider three cases based on the existence of the last two dimensions in the marginals, as follows:

a. Marginals contain the last two dimensions. In this case, we need handles that have each one of the first $n-2$ dimensions and the last two dimensions. All such handles are created in step 1. Hence, all the marginals containing the last two dimensions are covered.

b. Marginals do not contain the last two dimensions. In this case, all the marginals must have any two of the first $n-2$ (an example of such a marginal is $\langle D_1, D_2\rangle$ of a 7-dimensional data cube). In order to cover such marginals, we need handles that have any three of the first $n-2$ dimensions. By the inductive hypothesis, there is a handle, which has three of the first $n-2$ dimensions, chosen in step 2 that covers any marginal that has any two of the first $n-2$ dimensions. Hence, all the marginals that do not contain the last two dimensions are covered.

c. Marginals contain the $(n-1)^{th}$ or the n^{th} dimensions but not both. In this case, the marginals have one out of the first $n-2$ dimensions and either of the last two dimensions. But whichever one of the first $n-2$ dimensions are in the marginals, there is one handle that has exactly one of the first $n-2$ dimensions and the last two dimensions, created in step 1. Hence, all the marginals with either of the last two dimensions are also covered.

Therefore, we cover all the marginals of an n-dimensional data cube, where the marginals are of size two, using $N(n, n-2) \leq \frac{(n-1)^2}{4}$ handles, where $n \geq 5$ and n is odd. ∎

F.2.1 Aside: solving recurrences

We are going to propose several recurrences that describe inductive constructions of sets of handles. While we do not want to explain how one discovers the solution to each recurrence, there is a general pattern that can be used by the reader who wants to see how the solutions are derived; see [19].

A recurrence like $C(n) \leq n-2+C(n-2)$ from Section 8.2.6 will have a solution that is a quadratic polynomial, say $C(n) = an^2 + bn + c$. It turns out that the constant term c is needed only to make the basis hold, but we can get the values of a and b by replacing the inequality by an equality, and then recognizing that the terms depending on n must be 0. In this case, we get

$$an^2 + bn + c = n - 2 + a(n-2)^2 + b(n-2) + c$$

or

$$an^2 + bn + c = n - 2 + an^2 - 4an + 4a + bn - 2b + c$$

Cancelling terms and bringing the terms with n to the left, we get

$$n(4a - 1) = 4a - 2b - 2$$

Since a function of n cannot be a constant unless the coefficient of n is 0, we know that $4a - 1 = 0$, or $a = 1/4$. The right side of the equality must also be 0, so we get $4(1/4) - 2b - 2 = 0$, or $b = -1/2$. We thus know that $C(n) = n^2/4 - n/2 + c$ for some constant c, depending on the basis value.

Theorem 8.9 (Upper bound on the number of handles) *For an n-dimensional data cube, where the marginals are of size two, the number of handles is bounded above by*

$$C(n, m, 2) \leq \frac{n^2}{2(m-1)}$$

where $n \geq 5$.

Using the technique suggested in this Section F.2.1 along with the obvious basis case $C(m, m, 2) = 1$, we get the solution:

$$C(n, m, 2) \leq \frac{n^2}{2(m-1)} - \frac{n}{2} + 1 - \frac{m}{2(m-1)}$$

Note that asymptotically, this solution uses $\frac{n^2}{2(m-1)}$ handles, while the lower bound is $\frac{n(n-1)}{m(m-1)}$.

Appendix G

Pseudocodes and Theorems related to Privacy-Preserving MapReduce-based Computations (Chapter 10)

G.1 Algorithm for Creating Secret-Shares

Algorithm 11: Algorithm for creating secret-shares

```
   Inputs: $\mathcal{R}$: a relation having $n$ tuples and $m$ attributes, $c$: the number of clouds
   Variables: $letter$: represents a letter
1  Function $create\_secret\text{-}shares(\mathcal{R})$ begin
2      for $(i, j) \in (n, m)$ do
3          foreach $letter[i, j]$ do $Make\_shares(letter[i, j])$
4  Function $Make\_shares(letter[i, j])$ begin
5      Make unary-vectors, where the position of letter has value 1 and all the other
         values are 0
6      Use $n$ polynomials of an identical degree for creating secret-shares of 0 and 1,
         where $n$ is length of the vector
7      Send secret-shares to $c$ clouds
```

G.2 Count Algorithm

Steps in Counting a Pattern p using Algorithm 12

1. The user creates secret-share of p (see line 1) and sends secret-share of p, length (x) of p, and the attribute (m') where to count p, to c clouds; see line 2.

2. The cloud executes a map function, line 3, that

Algorithm 12: Algorithm for privacy-preserving count operation in the clouds using MapReduce

```
Inputs: $R$: a relation of the form of secret-shares having $n$ tuples and $m$ attributes, $p$:
  a searching pattern, $c$: the number of clouds, $N_j^{(i)}$: defined in Table 10.2
Output: $\ell$: the number of occurrences of $p$
Interfaces: $length(p)$: finds length of $p$
$attribute(p)$: which attribute of the relation has to be searched for $p$
Variables: $int\_result_i$: the output at $i^{th}$ cloud after executing the map function
$SS_k[i,j]$: shows a letter of the form of secret-share at $i^{th}$ position of $k^{th}$ string in $j^{th}$
  attribute
$result[]$: at the user-side to store outputs of all the clouds
User-side:
1  Compute secret-shares of $p$: $p' \leftarrow Make\_shares(p)$          // Algorithm 11
2  Send $p', x \leftarrow length(p), m' \leftarrow attribute(p)$ to $c$ clouds
Cloud $i$:
3  $int\_result_i \leftarrow MAP\_count(p', x, m')$
4  Send $int\_result_i$ back to the user
User-side:
5  $result[i] \leftarrow int\_result_i, \forall i \in \{1, \cdots, c\}$
6  Compute the final output: $\ell \leftarrow REDUCE(result[])$
7  Function $MAP\_count(p', x, m')$ begin
8      for $i \in (1, n)$ do $temp+ = Automata(SS_i[*, m'], p')$
9      return($\langle key, N_{x+1}^n \rangle$)
10 Function $Automata(SS_i[*, m'], p')$ begin
       $N_1 = 1$
       $N_2^i = N_1 \times (SS_i[1, m'] \times p'[1])$
       $N_3^i = N_2^i \times (SS_i[2, m'] \times p'[2])$
           $\vdots$
       $N_{x+1}^i = N_{x+1}^i + N_x^i \times (SS_i[x, m'] \times p'[x])$
       return($N_{x+1}^i$)
11 Function $REDUCE(result[])$ begin
       return(Assign $result[]$ to a reducer that performs the interpolation)
```

a. Reads each value of the form of secret-share of the m' attribute and executes AA containing $x + 1$ nodes

b. Executes accumulating-automata (AA), line 10, and computes the final output of the form of $\langle key, value \rangle$, where a key is an identity of the input split over which the map function was executed, and the value of the form of a secret-share is the final output that shows the total number of occurrence of p; line 9.

c. The final output after executing AA on each tuple is provided to the user, line 4.

3. The user executes a reduce function, for obtaining the final output. The outputs from all

the cloud are assigned to reducers based on the keys, and reducers perform the interpolation to provide the final output ℓ, line 11.

Theorem 10.1 (Cost for *count* operation) *The communication cost, the computational cost at a cloud, and the computational cost at the user-side for counting the occurrences of a pattern is at most $\mathcal{O}(1)$, at most nw, and at most $\mathcal{O}(1)$, respectively, where n is the number of tuples in a relation and w is the maximum bit length.*

Proof. Since a user sends a patterns of bit length w and receives c values from the clouds, the communication cost is almost constant that is $\mathcal{O}(1)$. The cloud works on a specific attribute containing n values, each of bit length at most w; hence, the computational cost at a cloud is at most nw. The user only performs the interpolation on the c values; hence, the computational cost at the user-side is also constant, $\mathcal{O}(1)$. ■

G.3 Single Tuple Fetch Algorithm

Theorem 10.2 *The communication cost, the computational cost at a cloud, and the computational cost at the user-side for fetching a single tuple containing a pattern is at most $\mathcal{O}(mw)$, at most $\mathcal{O}(nmw)$, and at most $\mathcal{O}(mw)$, respectively, where a relation has n tuples and m attributes and w is the maximum bit length.*

Proof. The user sends a pattern of bit length w and receives c secret-shares and, eventually, a tuple containing m attributes of size at most mw. Thus, the communication cost is at most $\mathcal{O}(mw)$ bits. The cloud counts the occurrences of the pattern in a specific attribute containing n values, and then again, performs a similar operation on the n tuples with multiplication of the resultant to each m values of bit length at most w. Hence, the computational cost at the cloud is at most $\mathcal{O}(nmw)$. The user performs the interpolation on c values to know the occurrences of the pattern, and then, again performs the interpolation on c tuples containing m attributes. Thus, the computational cost at the user-side is at most $\mathcal{O}(mw)$. ■

Steps in Fetching a Single Tuple containing a Pattern p using Algorithm 13

1. The user executes Algorithm 12 for counting occurrences, say ℓ, of p; line 1.
2. If ℓ is one, the user sends secret-shares of p, length, x, of p, and attribute, m', where p occurs, to c clouds; line 3.
3. The cloud executes a map function that
   a. Executes AA on i^{th} value of the m' attribute (line 9), and this provides a value, say val, either 0 or 1 of form of secret-shares. Multiply val by all the values of m attributes in the i^{th} tuples; line 10.

Algorithm 13: Algorithm for privacy-preserving search operation and **fetching a single tuple** from the clouds

```
Inputs: $R$, $n$, $m$, $p$, and $c$ are defined in Algorithm 12
Output: A tuple $t$ containing $p$
Variables: $\ell$: the number of occurrences of $p$
$int\_result\_search_i$: the output at a cloud $i$ after executing the fetching a single tuple
  in a privacy-preserving manner
$result\_search[]$: an array to store outputs of all the clouds
$SS_k[i, j]$: defined in Algorithm 12
User-side:
1  Compute secret-shares of $p$: $p' \leftarrow Make\_shares(p)$ and execute Algorithm 12 for
     obtaining the total number of occurrences ($\ell$) of $p$
2  if $\ell > 1$ then Execute Algorithm 14
3  else Send $p', x \leftarrow length(p)$ and $m' \leftarrow attribute(p)$ to $c$ clouds
Cloud $i$:
4  $int\_result\_search_i \leftarrow MAP\_single\_tuple\_fetch(p', x, m')$
5  Send $int\_result\_search_i$ back to the user
User-side:
6  $result\_search[i] \leftarrow int\_result\_search[i], \forall i \in \{1, \cdots, c\}$
7  Obtain the tuple $t \leftarrow REDUCE(result\_search[])$
8  Function $MAP\_single\_tuple\_fetch(p', x, m')$ begin
9      for $i \in (1, n)$ do
           $temp+ = Automata(SS_i[*, m'], p')$                    // Algorithm 12
10         for $j \in (1, m)$ do $temp \times SS_i[*, j]$
11     for $(j, i) \in (m, n)$ do $S_j \leftarrow$ add all the shares of $j^{th}$ attribute
12     return($\langle key, N^n_{x+1}||S_1||S_2|| \ldots ||S_m \rangle$)
13 Function $REDUCE(result\_search[])$ begin
       return(Assign $result\_search[]$ to a reducer that performs the interpolation)
```

b. When the execution of AA is completed on all the n secret-shares of the m' attribute, add all the secret-shares an attribute; line 11.

c. The cloud sends the final output of AA and sum of each attribute's secret-shares to the user; line 12.

4. The user receives c tuples and executes a reduce function that performs the interpolation and provides the desired tuple; lines 7 and 13.

G.4 Multi-tuple Fetch Algorithm

We first prove the theorem related to multi-tuple fetch using the naive algorithm, given in Section 10.5.2.

Theorem 10.3 *After obtaining the addresses of the desired tuples containing a pattern, p, the communication cost, the computational cost at a cloud, and the computational cost at the user-side for fetching the desired tuples is at most $\mathcal{O}((n+m)\ell w)$, $\mathcal{O}(\ell nmw)$, and at most $\mathcal{O}((n+m\ell)w)$, respectively, where a relation has n tuples and m attributes, w is the maximum bit length, and ℓ is the number of tuples containing p.*

Proof. In the first round of the naive algorithm, the user receives n secret-shares, each of bit-length at most w, of a particular attribute. In the second round, the user sends a $\ell \times n$ matrix and receives ℓ tuples, each of size at most mw. Thus, the maximum number of bits flow is $\mathcal{O}((n+m)\ell w)$. A mapper performs string matching operations on n secret-shares of a particular attribute in the first round and then matrix multiplication on all the n tuples and m attributes in the second round. Hence, the computational cost at the cloud is at most $\mathcal{O}(\ell nmw)$. The computational cost at the user-side is at most $\mathcal{O}((n+m\ell)w)$, since the user works on the n secret-shares of a specific attribute, creates a $\ell \times n$ matrix in the first round, and then works on ℓ tuples containing m values, each of size at most w bits. ■

Tree-based Algorithm 14

Algorithm 14's pseudocode description. A user creates secret-shares of p and obtains the number of occurrence, ℓ, see line 1. When the occurrences $\ell = 1$, we can perform Algorithm 13 for fetching the only tuple having p, see line 2. When the occurrences $\ell > 1$, the user needs to know the addresses of all the ℓ tuples contain p. Thus, the user requests to partition the input split/relation to ℓ blocks, and hence, sends ℓ and p of the form of secret-shares to the clouds, see line 3.

The mappers partition the whole relation or input split into ℓ blocks, perform privacy-preserving count operation in each blocks, and send all the results back to the user, see lines 4 - 6. The user again executes a reduce function that performs the interpolation and provides the number of occurrences of p in each block, see line 8. Based on the number of occurrences of p in each block, the user decides which block needs further partition, and there are four cases, as follows:

1. The block contains no occurrence of p: it is not necessary to handle this block.
2. The block contains only one tuple containing p: it is easy to determine its address using function $Address\_fetch()$ that is based on AA; see lines 10 and 14.
3. The block contains h tuples and each h tuple contains p: directly know the addresses, *i.e.*, all the h tuples are required to fetch; see line 11.
4. The block contains h tuples and more than one, but less than h tuples contain p: we cannot know the addresses of these tuples. Hence, the user recursively requests to partition that block and continues the process until the subblocks satisfy the above mentioned Case 2 or Case 3; see lines 9 and 13.

Algorithm 14: Algorithm for privacy-preserving search operation and fetching multiple tuples from the clouds

```
Inputs: $R$, $n$, $m$, $p$, and $c$ are defined in Algorithm 12
Outputs: Tuples containing $p$
Variables: $\ell$: the number of occurrences of $p$
$int\_result\_block\_count_i[j]$: at $i^{th}$ cloud to store the number of occurrences of the
  form of secret-shares in $j^{th}$ block
$result\_block\_count[]$: at the user-side to store the count of occurrences of $p$ of the
  form of secret-shares in each block at each cloud
$count[]$: at the user-side to store the count of occurrences of $p$ in each block
$Address[]$: stores the addresses of the desired tuples
User-side:
1  Compute secret-shares of $p$: $p' \leftarrow Make\_shares(p)$ and execute Algorithm 12 for
     obtaining the total number of occurrences ($\ell$) of $p$
2  if $\ell = 1$ then Execute Steps 3 to 13 of Algorithm 13
3  else Send $p', x \leftarrow length(p)$, $m' \leftarrow attribute(p)$, $\ell$ to $c$ clouds
Cloud $i$:
4  Partition $R$ into $\ell$ equal blocks, where each block contains $h = \frac{n}{\ell}$ tuples
5  $int\_result\_block\_count_i[j] \leftarrow$ Execute $MAP\_count(p', x, m')$ $j^{th}$ block,
     $\forall j \in \{1, \cdots, \ell\}$
6  Send $int\_result\_block\_count_i[j]$ back to the user
User-side:
7  $result\_block\_count[i, j] \leftarrow int\_result\_block\_count_i[j]$, $\forall i \in \{1, \cdots, c\}$,
     $\forall j \in \{1, \cdots, \ell\}$
8  Compute $count[j] \leftarrow REDUCE(result\_block\_count[i, j])$
9  if $count[j] \notin \{0, 1, h\}$ then
       Question the clouds about $j^{th}$ block and send $\langle p', count[j], m', j\rangle$ to clouds
10 else if $count[j] = 1$ then
       $Address \leftarrow Address\_fetch(p', x, j)$
11 else if $count[j] = h$ then
       $Address \leftarrow (j-1)h + 1, (j-1)h + 2, \cdots, (j-1)h + h$
12 Fetch the tuples whose addresses are in $Address$ using a method described in the naive
     algorithm
Cloud $i$:
13 if Receive $\langle p', count[j], m', j\rangle$ then Perform Steps 4 to 6 to $j^{th}$ block recursively
14 Function $Address\_fetch(p', x, j)$ begin
       $line\_number \leftarrow 0$
15     for $i \in ((j-1)h + 1, (j-1)h + h)$ do
16         $line\_number += Automata(SS_i[*, m'], p') \times i$        // Algorithm 12
17     return($line\_number$)
```

When the user obtains the addresses of all the tuples containing p, she fetches all the tuples using a method described for the naive algorithm, see line 12.

Theorem 10.4 *The maximum number of rounds for obtaining addresses of tuples containing a pattern, p, using Algorithm 14 is* $\lfloor log_\ell n\rfloor + \lfloor log_2 \ell\rfloor + 1$*, and the communication cost for obtaining such addresses is at most* $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell\big)$*. The computational cost at a cloud and the computational cost at the user-side is at most* $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell n w\big)$ *and at most* $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell\big)$*, respectively, where a relation has* n *tuples and* m *attributes,* ℓ *is the number of tuples containing p, and* w *is the maximum bit length.*

Proof. According to the description of Algorithm 14, in the current Q&A round, we partition the blocks that are specified in last round into ℓ blocks equally. Thus, in i^{th} round of Q&A, the number of the items contained in each sub-block is at most $\frac{n}{\ell^i}$.

After $\lfloor \log_\ell n\rfloor$ rounds of Q&A, the number of the items contained in every block is fewer than ℓ. At this time, note that there may be some blocks still contain more than one tuple containing p. Thus, we need at most $\lfloor \log_2 \ell\rfloor$ rounds for determining the addresses of those tuples. When the user finishes partitioning all the blocks that contains more than two tuples containing p, it needs at most one more round for obtaining the addresses of related tuples. Thus, the number of Q&A rounds is at most $\lfloor log_\ell n\rfloor + \lfloor log_2 \ell\rfloor + 1$.

Notice that for each round, there are at most $\frac{\ell}{2}$ blocks containing more than two tuples containing p that indicates that at most $\frac{\ell}{2}$ blocks need further partitioning. So in every Q&A round (except the first round requires ℓ answers), each cloud only needs to perform the count operation for $\frac{\ell}{2}$ sub-blocks and send the results back to the user. When the cloud finishes partitioning, it has to perform $Address\_fetch()$ operation to determine the addresses. It requires at most ℓ words transition between the user and each cloud. Therefore, the communication cost is at most $\mathcal{O}\big((log_\ell n + log_2 \ell)\cdot \ell\big)$.

A cloud performs the count operation in each round, hence the computational cost at the cloud is at most $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell n w\big)$. In each round, the user performs the interpolation for obtaining the occurrences of the pattern in each block; hence the computational cost at the user-side is at most $\mathcal{O}\big((log_\ell n + log_2 \ell)\ell\big)$. ■

G.5 Proof of Theorems related to Privacy-Preserving Equijoin

Theorem 10.5 *The communication cost, the computational cost at a cloud, and the computational cost at the user-side for performing the equijoin of two relations* X *and* Y*, where a joining value can occur at most one time in a relation, is at most* $\mathcal{O}(nmw)$*, at* *most $\mathcal{O}(n^2mw)$, and at most $\mathcal{O}(nmw)$, respectively, where a relation has n tuples and m attributes and w is the maximum bit length.*

Proof. Since the user receives the whole relation of n tuples and at most $2m-1$ attributes, the communication cost is at most $\mathcal{O}(nmw)$, and due to the interpolation on the n tuples, the computational cost at the user-side is at most $\mathcal{O}(nmw)$. A mapper compares each value of the joining attribute of the relation X to all n values of the joining attribute of the relation Y, and it results in at most n^2 comparisons. Further, the output of the comparison is multiplied by $m-1$ attributes of the corresponding tuple of the relation Y. Hence, the computational cost at a cloud is at most $\mathcal{O}(n^2mw)$. ■

Theorem 10.6 *The number of rounds, the communication cost, the computational cost at a cloud, and the computational cost at the user-side for performing the equijoin of two relations X and Y, where a joining value can occur in multiple tuples of a relation, is at most $\mathcal{O}(2k)$, at most $\mathcal{O}(2nwk + 2k\ell^2mw)$, at most $\mathcal{O}(\ell^2kmw)$, and at most $\mathcal{O}(2nw + 2k\ell^2mw)$, respectively, where a relation has n tuples and m attributes, k is the number of identical values of the joining attribute in the relations, ℓ is the maximum number of occurrences of a joining value, and w is the maximum bit length.*

Proof. Since there are at most k identical values of the joining attribute in both the relations and all the k values can have different number of occurrences in the relations, the user has to send at most $2k$ matrices (following an approach of the naive algorithm for fetching multiple tuples) in $\mathcal{O}(2k)$ rounds.

The user sends at most $2k$ matrices, each of n rows and of size at most w; hence, the user sends at most $\mathcal{O}(2knw)$ bits. Since at most ℓ tuples have an identical value of the joining attribute in one relation, the equijoin provides at most ℓ^2 tuples. The user receives at most ℓ^2 tuples for each k value having at most $2m-1$ attributes; hence, the user receives at most $\mathcal{O}(2k\ell^2mw)$ bits. Therefore, the communication cost is at most $\mathcal{O}(2nwk + 2k\ell^2mw)$ bits.

The cloud of the first layer executes the naive algorithm for fetching multiple tuples for all k values of both relations having $2n$ tuples; hence the clouds of the first layer performs at most $\mathcal{O}(2nkw)$ computation. In the second layer, a cloud performs the equijoin (or concatenation) of at most ℓ tuples for each k value; thus, the computational cost at the cloud is at most $\mathcal{O}(\ell^2kmw)$.

The user first interpolates at most $2n$ values of bit length w of the joining attribute, and then, interpolates $k\ell^2$ tuples containing at most $2m-1$ attributes of bit length w. Therefore, the computational cost at the user-side is at most $\mathcal{O}(2nw + 2k\ell^2mw)$. ■

G.6 Pseudocodes related to Range Query

Algorithm 15: $SS\text{-}SUB(A, B)$: 2's complement based subtraction of secret-sharing

```
Inputs: $A = [a_{t-1}a_{t-2}\cdots a_1a_0], B = [b_{t-1}b_{t-2}\cdots b_1b_0]$ where $a_i, b_i$ are secret-shares
  of bits of 2's complement represented number, $t$: the length of $A$ and $B$ in binary
  form
Outputs: $rb_{t-1}$: the sign bit of $B - A$
Variable: $carry[]$: to store the carry for each bit addition
$rb$: to store the result for each bit addition
1 $a_0 \leftarrow 1 - a_0$                                  // Invert of the LSB of $A$
2 $carry[0] \leftarrow a_0 + b_0 - a_0 \cdot b_0$
3 $rb_0 \leftarrow a_0 + b_0 - 2 \cdot carry[0]$                       // $\bar{a}_0 + b_0 + 1$
4 for $i \in (i, t-1)$ do
      $a_i \leftarrow 1 - a_i$                              // invert each bit $A \to \bar{A}$
      $rb_i \leftarrow a_i + b_i - 2a_ib_i$
      $carry[i] \leftarrow a_ib_i + carry[i-1] \cdot rb_i$                 // The carry bit
      $rb_i += carry[i-1] - 2 \cdot carry[i-1] \cdot rb_i$
5 return($rb_{t-1}$)                                     // The sign bit of $B - A$
```

Algorithm 15's pseudocode description. Algorithm 15 provides a way to perform 2's complement based subtraction on secret-shares. We follow the definition of 2's complement subtraction to convert $B - A$ into $B + \bar{A} + 1$, where $\bar{A} + 1$ is 2's complement representation of $-A$. We start at the least significant bit (LSB), invert a_0, calculate $\bar{a}_0 + b_0 + 1$ and its carry bit, see lines 1- 3. Then, we go through the rest of the bits, calculate the carry and the result for each bit, see line 4. After finishing all the computations, the most significant bit (MSB) or the sign bit is returned; see line 5.

Algorithm 15 is similar to the algorithm presented in [49], but simpler, as we only need the sign bit of the result. After obtaining secret-shares of sign bits of $x - a$ and $b - x$, we perform an extra calculation:

$$1 - \big(sign(x - a) + sign(b - x)\big). \tag{G.1}$$

According to Equation 10.1, if $x \in [a, b]$, the result of Equation G.1 is secret-share of 1; otherwise, the result is secret-share of 0. Based on Equation G.1, we can obtain the number of occurrences, which are in the required range in the database.

Algorithm 16's pseudocode description. Algorithm 16 works in two phases, as: first, it counts the occurrences of numbers that belong in a range, and second, it fetches all those corresponding tuples. A user creates secret-shares of the range numbers a, b and sends them to c clouds, see lines 1 and 2.

Algorithm 16: Algorithm for privacy-preserving range query in the clouds using MapReduce

```
Inputs: $R$, $n$, $m$, and $c$: defined in Algorithm 12, $[a, b]$: a searching range
Output: $\ell$: the number of occurrence in $[a, b]$
Variables: $int\_result_i$: is initialized to 0 and the output at $i^{th}$ cloud after executing
  the $MAP\_range\_count$ function
User-side:
1  Compute secret-shares of $a, b$: $a' \leftarrow Make\_shares(a)$, $b' \leftarrow Make\_shares(b)$
2  Send $a'$, $b'$, $m' \leftarrow attribute(a)$ to $c$ clouds
Cloud $i$:
3  for $i \in (1, n)$ do $int\_result_i \leftarrow MAP\_range\_count(a', b', m')$
4  Send $int\_result_i$ back to the user
User-side:
5  $result[i] \leftarrow int\_result_i, \forall i \in \{1, \cdots, c\}$
6  $\ell \leftarrow REDUCE(result[])$
7  Execute Algorithm 13 if $\ell = 1$; otherwise, execute Algorithm 14
8  Function $MAP\_range\_count(a', b', m')$ begin
9      $sign_{x-a'} \leftarrow SS\text{-}SUB(x, a')$                // Algorithm 15
10     $sign_{b'-x} \leftarrow SS\text{-}SUB(b', x)$                // Algorithm 15
11     return$(\langle key, 1 - sign_{x-a'} - sign_{b'-x} \rangle)$
12 Function $REDUCE(result[])$ begin
       return(Assign $result[]$ to a reducer that performs the interpolation)
```

The cloud executes a map function that checks each number in an input split by implementing Algorithm 15, see lines 3 and 8. The map function, see line 8, provides 1 (of the form of secret-share) if $x \in [a, b]$; otherwise, 0 (of the form of secret-share) if $x \notin [a, b]$. The cloud provides the number of occurrences (of the form of secret-shares) that belong in the ranges to the user, see line 4. The user receives all the values from c clouds and execute a reduce function that interpolates them to obtain the count, see lines 5 and 6. After obtaining the number of occurrences, say ℓ, the user can fetch the corresponding tuples by following Algorithm 13 or 14.

Degree reduction. Note that in range query, we utilize 2's complement subtraction and each secret-shared bit of the operands. However, during the subtraction procedure, the degree of the polynomial (for secret-sharing) increases. For example, one can check that degree of MSB doubles when one subtraction completed. In [49], the authors add two more players for degree reduction, if we do not have enough clouds to recover the secrets, we can follow the same line as the degree reduction algorithm presented in [49]. For simplicity, we do not give details of the algorithms for degree reduction, interested readers may refer to [49, 45].

Theorem G.1 *The communication and the computational costs of the range count query have the same order of magnitude as Algorithm 12, and the communication and computational cost of fetching multi-tuples satisfying a range have the same order of magnitude as Algorithm 13 or 14.*

Note that the function $MAP\_range\_count()$, see line 8 of Algorithm 16, works on each value of a specific attribute as we did in the count queries, Section 10.4. Once we know all the occurrences of tuples satisfying a range, we find their address using Algorithm 13 or 14. Thus, the communication and the computational costs have an identical order of magnitude as Algorithm 13 or 14.

בחלק השני אנו גם מעריכים השפעות על עלות התקשורת עבור פתרון בעיית interval join עבור מרווחים חופפים ומספקים אלגוריתם interval join דו כיווני עבור מרחב הזיכרון של reducer. אנו גם חוקרים חסמים תחתונים של עלות תקשורת עבור מספר בעיות interval join עבור סוגים שונים של מרווחים. לבסוף, אנו מתייחסים לבעיית חישוב שוליים מרובים לקוביית-נתונים על ידי reducer יחיד. אנו מוצאים את החסם התחתון על עלות התקשורת לבעיה ומספקים מספר אלגוריתמים כמעט-אופטמליים עבור חישוב שולים מרובים.

החלק השלישי מתמקד בהיבטי אבטחה ופרטיות ב-MapReduce. אנו מוצאים אתגרי אבטחה ופרטיות ודרישות בעולם ה-MapReduce, תוך התחשבות במגוון יכולות של יריבים. לאחר מכן, אנו מספקים טכניקה משמרת-פרטיות עבור חישובי MapReduce המבוססת על שכפולי נתונים בצורת שיתוף-סוד של שמיר. אנו מראים כי למרות שיצירת שיתוף-סוד לכל ערך יוצרת שכפול מוגבר, שיכפול זה מונע מן הענן הכן-אך-סקרן, ללמוד על מסד הנתונים או על החישוב. אנו מספקים אלגוריתם שומר פרטיות מבוסס MapReduce לספירה, חיפוש, שליפה, ו- join מבוסס שיוון וטווח.

תקציר

בעוד כשלים בחומרה ותוכנה נפוצים במערכות מחשוב מבוזרות, שכפול הוא דרך להשגת יכולת התאוששות מהירה מכישלון (זמינות, עקביות, ואמינות) על-ידי יצירה ושימור של מספר עותקים של חומרה, תוכנה, נתונים ופרוטוקולי תקשורת. במסגרת תזה זו, אנו מציעים מספר גישות הקשורות לשכפול נתונים (או הודעות). בפרט, גישות לעיצוב אלגוריתמים מתייצבים עצמית לתקשורת קצה-אל-קצה וגישות עבור המודל החישובי MapReduce.

החלק הראשון מציג אלגוריתם מתייצב-עצמית לתקשורת קצה-אל-קצה אשר מעביר הודעות בסדר FIFO מעל ערוץ משכפל, מוגבל-קיבולת אשר אינו FIFO. הכפלת ההודעות נועדה להתמודד עם אובדן של הודעות, אך בעת ובעונה אחת גוררת התמודדות בשליטה על התקשורת מקצה-לקצה. בעבודה זו מוצג אלגוריתם מתייצב-עצמית המתמודד עם הכפלות להעברת הודעות שמעביר את ההודעות באותו הסדר בו הן נשלחות על-ידי השולח אל היעד. כל הודעה מועברת בדיוק פעם אחת ללא השמטה או הכפלה.

החלק השני עוסק בעיצוב מודלים ואלגוריתמים עבור MapReduce. ב-MapReduce, קלט מועתק עבור כמה וכמה Reducer-ים. כמות השיכפול של הקלט הוא גורם משמעותי בעלות התקשורת שמהווה מדד ביצועים עבור MapReduce. מודל חדש עבור MapReduce מוצג כאן, אשר בו לראשונה נלקחים בחשבון הגדרות מציאותיות בהן הקלטים שונים בגודלם. המודל החדש פותח את הדלת עבור אתגרים אלגוריתמיים שימושיים אשר חלקם נפתרים בהמשך. בפרט, אנו מציגים שתי מחלקות של בעיות התאמה, בשם כולם-לכולם ו-X-עבור-Y, ומראים כי בעיות התאמה אלו הם NP-קשות מבחינת אופטימיזציה של עלות התקשורת. אנו מספקים מספר אלגוריתמי קירוב עבור שתי הבעיות. לאחר מכן, אנו מספקים גישה אלגוריתמית חדשה עבור אלגוריתמי MapReduce אשר מקטינה את עלות התקשורת באופן ניכר, ובעת ובעונה אחת מתייחסים ל-"מקומיות" של הנתונים ול-Mappers ו-Reducers. הטכניקה המוצעת מונעת העברה של נתונים אשר אינם משתתפים בפלט הסופי.

<u>**הצהרת תלמיד המחקר עם הגשת עבודת הדוקטור לשיפוט**</u>

אני החתום מטה מצהיר/ה בזאת: (אנא סמן):

___ חיברתי את חיבורי בעצמי, להוציא עזרת ההדרכה שקיבלתי מאת מנחה/ים.

___ החומר המדעי הנכלל בעבודה זו הינו פרי מחקרי <u>מתקופת היותי תלמיד/ת מחקר</u>.

Shantanu Sharma

Digitally signed by Shantanu Sharma
DN: cn=Shantanu Sharma, o=UCI, ou=CS, email=shantnu.sharma@uci.edu, c=US
Date: 2016.09.04 00:49:09 -07'00'

תאריך: **17/02/2016** שם התלמיד/ה: שנטנו שרמה חתימה __

העבודה נעשתה בהדרכת

פרופ' שלומי דולב

במחלקה מדעי המחשב

בפקולטה מדעי הטבע

היבטי שכפול במערכות מבוזרות

מחקר לשם מילוי חלקי של הדרישות לקבלת תואר "דוקטור לפילוסופיה"

מאת

שנטנו שרמה

הוגש לסינאט אוניברסיטת בן גוריון בנגב

אישור המנחה [illegible]________________

אישור דיקן בית הספר ללימודי מחקר מתקדמים ע"ש קרייטמן _____________

ח' באדר א' תשע"ו **17/02/2016**

באר שבע

היבטי שכפול במערכות מבוזרות

מחקר לשם מילוי חלקי של הדרישות לקבלת תואר "דוקטור לפילוסופיה"

מאת

שנטנו שרמה

הוגש לסינאט אוניברסיטת בן גוריון בנגב

ח' באדר א' תשע"ו **17/02/2016**

באר שבע