Processing Theta-Joins on Shared-Nothing Systems

A dissertation presented
by

Alper Okcan

to the Faculty of the Graduate School
of the College of Computer and Information Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Northeastern University
Boston, Massachusetts
April, 2014
NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS APPROVAL FORM

THESIS TITLE: Processing Theta-Joins on Shared-Nothing Systems

AUTHOR: Aper Ozcen

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Thesis Advisor: [Signature] 03/27/2014
Date

Thesis Reader: [Signature] 03/26/2014
Date

Thesis Reader: [Signature] 03/11/14
Date

Thesis Reader: [Signature] 03/07/14
Date

GRADUATE SCHOOL APPROVAL:

Director, Graduate School: [Signature] 03/28/2014
Date

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature: [Signature] 03/28/2014
Date

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.
Abstract

Joins are essential for many large-scale data analysis tasks, and a variety of join conditions must be supported for many applications such as data-driven science, advertising, marketing, and social networks. Efficient parallel execution of joins is crucial to cope with the large volumes of data being collected and generated in many disciplines.

We explore how to efficiently process theta-joins in distributed shared-nothing systems. While equi-joins have been studied extensively, it was unclear how to efficiently distribute computation of arbitrary join conditions on a cluster of nodes. We show how to process theta-joins efficiently in parallel when the goal is to minimize response time. We propose a join model that simplifies creation of and reasoning about parallel join algorithms. Using this model, we introduce a randomized algorithm whose response time is provably within a small constant factor of the lower bound for a variety of join problems. For other popular classes of joins where this does not apply, we develop efficient heuristics.

The drawback of our distributed theta-join algorithms is a potentially high degree of input replication depending on the input data distribution, available input statistics, join condition, and cluster properties. We propose lightweight encoding and decoding strategies in order to reduce the amount of data transferred across the network. These strategies are also applicable to a large and diverse spectrum of applications executed using the MapReduce programming model. Data transfer reduction is achieved by dynamically and adaptively performing mapper-side tasks on the reducers.
Finally, we integrate our optimization techniques into Scolopax, a novel system that supports exploratory analysis for data-driven science. Scolopax can explore a huge space of possible hypotheses, returning a ranked list of those that best match the user preferences. Scolopax supports flexible join predicates used by scientists to compute relationships and correlations in high-dimensional scientific data.
Acknowledgements

I would like to thank my research advisor Professor Mirek Riedewald for his continued guidance and support during my graduate study. I learned everything necessary for being an independent researcher from him. He taught me how to do critical thinking, how to formulate, solve and present research problems. I am also grateful for his patience while guiding me through my research. He was always available for long discussions and open to new ideas. Without his continued guidance and support this dissertation would not have been possible.

I would also like to thank Professor Rajmohan Rajaraman, Professor Yanlei Diao, and Professor Alan Mislove for serving on my thesis committee and for their helpful comments and suggestions.

My sincere thanks also go to Daria Sorokina, Cliff Brunk, and Biswanath Panda, for leading me working on diverse exciting projects during my internships. I owe special thanks to Daria and Cliff for introducing me to a set of inspiring problems that shaped Chapter 4.

The motivation of my research came from the eBird project through our collaboration with the Cornell Lab of Ornithology. I would like to thank Biswanath Panda and Daniel Fink for their feedback and support while I was working on the Scolopax project. I also would like to thank all Northeastern University Masters students who contributed to Scolopax implementation: Priyank Desai, Pratik Gawande, Shweta Memane, Mathi Ramakrishnan, Baturalp Torun, and Yeshwanth Venkatesh.

I would like to thank my colleagues and my friends at Northeastern University who made my years of study very enjoyable. I thank Dimitris Kanoulas, Emrah
Bayraktaroglu, Christos Dimoulas, Evangelos Kanoulas, Pavel Metrikov, Arash Molavi Kakhki, and Bahar Qarabaqi for all the fun we have had.

Special thanks to my former advisor at Middle East Technical University, Professor Asuman Dogac, for introducing me to the field of database systems research and encouraging me to pursue my Ph.D. study.

Finally, I would like to thank Audrey for her personal support and great patience at all times. I would like to thank my brother and my parents for their endless love and support. I dedicate this dissertation to them.
# Contents

Abstract ........................................... iii

Acknowledgements ..................................... v

Contents ........................................... vii

List of Figures ....................................... xi

List of Tables ....................................... xiii

1 Introduction ...................................... 1
   1.1 Challenges and Contributions .................. 3
   1.2 Summary and Dissertation Outline .............. 6

2 Preliminaries ..................................... 7
   2.1 Overview of MapReduce ......................... 7
   2.2 Parallel Join Processing using MapReduce .......... 11
       2.2.1 Limitations and Challenges ................. 12

3 Processing Theta-Joins using MapReduce .......... 13
   3.1 Cost Model for Joins in MapReduce ............ 14
   3.2 Join Model .................................... 17
   3.3 The 1-Bucket-Random Algorithm ................ 19
       3.3.1 Implementing the Cross-Product .......... 19
       3.3.2 Implementing Theta-Joins ................. 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Improving Highly Selective Theta-Joins by Exploiting Statistics</td>
<td>34</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Exploiting Approximate Equi-Depth Histograms</td>
<td>34</td>
</tr>
<tr>
<td>3.4.2</td>
<td>M-Bucket-I and M-Bucket-O Algorithms</td>
<td>36</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Using M-Bucket in MapReduce</td>
<td>42</td>
</tr>
<tr>
<td>3.4.4</td>
<td>The Overall Algorithm</td>
<td>42</td>
</tr>
<tr>
<td>3.5</td>
<td>Experimental Evaluation</td>
<td>43</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Equi-Joins: 1-Bucket-Random vs. Repartition join</td>
<td>44</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Input-Size Dominated Joins</td>
<td>45</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Output-Size Dominated Joins</td>
<td>48</td>
</tr>
<tr>
<td>3.6</td>
<td>Conclusions</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>Optimizing MapReduce Programs using Anti-Combining</td>
<td>51</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>Motivating Examples and Optimization Opportunities for Anti-Combining</td>
<td>54</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Query Suggestion using Log Processing</td>
<td>54</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Processing In-Memory Theta-Joins</td>
<td>57</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Optimization Opportunities</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Anti-Combining Strategies</td>
<td>59</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Eager Sharing Strategy</td>
<td>60</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Lazy Sharing Strategy</td>
<td>65</td>
</tr>
<tr>
<td>4.3.3</td>
<td>The Shared Data Structure</td>
<td>68</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Enabling Adaptive Runtime Optimization</td>
<td>69</td>
</tr>
<tr>
<td>4.4</td>
<td>Experimental Evaluation</td>
<td>75</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Anti-Combining Overhead Analysis</td>
<td>76</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Query-Suggestion</td>
<td>77</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Query-Suggestion With Combiner</td>
<td>77</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Query-Suggestion With Compression</td>
<td>79</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Effect on Disk I/O and CPU</td>
<td>80</td>
</tr>
<tr>
<td>4.4.6</td>
<td>CPU Intensive Workloads</td>
<td>80</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Example summaries from Ornithology domain .......................... 2

2.1 MapReduce overview ......................................................... 9

2.2 Example Equi-join Implementation in MapReduce ....................... 11

3.1 Join matrices for equi-join, similarity-join, and inequality-join. Numbers indicate join attribute values from $S$ and $T$, shaded cells indicate join results. $M(i,j)$ indicates cell numbering. .......................... 16

3.2 Matrix-to-reducer mappings for repartition join algorithm (left), random (center), and balanced (right) approach ............................... 18

3.3 Initial Partitioning by $c_S \times c_T$ Rectangles. ............................ 21

3.4 Matrix-to-reducer mapping example. The join matrix is partitioned into 4 regions, each assigned to one reducer. ............................... 24

3.5 Matrix-to-reducer mapping examples ...................................... 33

3.6 Example join matrix computed based on histograms ..................... 35

3.7 Illustration of the M-Bucket-I Algorithm .................................... 38

3.8 Illustration of Algorithm 7 on a horizontal fragment bounded by $row_f$ and $row_l$ using 2 reducers having maxInput = 5 .......................... 41

3.9 Skew Resistance of 1-Bucket-Random (Output Imbalance) on Synth-$\alpha$ ................................................................. 44

3.10 Skew Resistance of 1-Bucket-Random (Run-time) on Synth-$\alpha$ .... 44

3.11 Experimental Results for Input Size Dominated Joins .................. 46

3.12 Experimental Results for Output Size Dominated Joins ................ 49
LIST OF FIGURES

4.1 Anti-Combining Intuition ............................................. 52
4.2 Query-Suggestion Problem using MapReduce .......................... 56
4.3 (a) Example join matrix to be partitioned (b) Memory-aware partitioning for the join matrix (c) Alternative memory-aware partitioning .... 58
4.4 *EagerSH* Map Phase for Query-Suggestion .......................... 61
4.5 Original Reduce vs. *EagerSH*’s Reduce for Query-Suggestion ........ 63
4.6 *LazySH* Map Phase for Query-Suggestion ............................ 66
4.7 *LazySH*’s Reduce Phase for Query-Suggestion ........................ 67
4.8 Syntactic rewrite of original mapper class to enable Anti-Combining .... 70
4.9 Syntactic rewrite of original reducer class to enable Anti-Combining .... 71
4.10 Total Map Output Size for Query-Suggestion .......................... 78
4.11 Total Map Output Size for Query-Suggestion using Combiner .......... 78
4.12 Total Map Output Size for Query-Suggestion using Combiner and Compression .................................................. 79
4.13 Total CPU Time using Runtime Cost-Based Optimization ............ 82
4.14 Total Map Output Size and Runtime for Theta-Join Query .......... 82
4.15 Memory Size Limit Effect on 1-Bucket-Random and Anti-Combining .... 84
5.1 Example summaries of a prediction model ............................... 86
5.2 Average cluster trajectories for spring migration and cluster map of Tree Swallow .................................................. 87
5.3 Scolopax System Architecture ............................................. 91
5.4 The Scolopax summary generator input screen allows the user to select species of interest, explore areas of interest on the interactive map and create partitions on latitude and longitude. ............................ 92
5.5 The Scolopax summary ranker output screen allows the user to interactively examine the ranked list of summaries, see the corresponding region on the map, filter results for further investigation, and resubmit a refined query. ........................................ 93
5.6 The Scolopax correlation finder input screen allows the user to select species of interest, explore areas of interest on the interactive map, create partitions on latitude and longitude, and express a variety of join conditions.

5.7 The Scolopax correlation finder output screen allows the user to interactively examine the ranked list of summary pairs, see the corresponding region on the map, filter results for further investigation, and resubmit a refined query.

5.8 The Scolopax cluster ranker output screen allows the user to interactively examine the cluster of summaries with the average cluster trajectories, and each individual summary in a cluster.

List of Tables

3.1 M-Bucket-I cost details (seconds) ............................................. 47
3.2 M-Bucket-O cost details (seconds) ............................................. 49
4.1 Encoding Strategies for each Sharing Opportunity ....................... 59
4.2 Total Cost Breakdown of Compression Algorithms ...................... 79
4.3 Total Cost Breakdown of Query-Suggest .................................. 80
CHAPTER 1

Introduction

The rapid growth of data volumes poses a challenge in many scientific disciplines. Scientists have to process data sets collected by large-scale experiments and sensors [75]. For example, the Large Hadron Collider that is built to help scientists to answer key unresolved questions in particle physics produces roughly 15 petabytes of data annually [3]. The Large Synoptic Survey Telescope is expected to produce about 20 terabytes per night [4]. One of the grand challenges of data-driven science is to find interesting patterns in massive high-dimensional data sets that may lead to new hypotheses. This process is currently limited by the large amount of required human effort and the high computational cost.

Consider the following examples from Ornithology domain that illustrate interesting patterns that could be found in observational data and the corresponding hypotheses proposed by the domain scientists. Figure 1.1a is a 1-dimensional summary that shows how the estimated probability of observing the Acorn Woodpecker in California varies with local human population density. After seeing the strong relationship between increasing human population density and lower woodpecker probability, the scientists hypothesized that the bird’s reliance on dying tree branches for storing acorns conflicts with properties of habitats in more densely settled areas. The plots in Figure 1.1b represent the annual estimated observation probability of two bird species in Northern New England. The strikingly complementary bi-annual cycles for Purple Finch and Common Redpoll hint at habitat competition, possibly driven by the availability of local food sources.
Supporting discovery of such patterns is challenging. An interesting hypothesis could be found in some “slice” or “dice” of the data space, e.g., a certain trend might show only in some small region or a combination of habitat features and elevation ranges. Hence to broadly explore possible hypotheses, a huge number and variety of such summaries, both one- and multi-dimensional, needs to be generated. To find correlations, e.g., habitat competition between two species (similar to Figure 1.1b) or variables with similar effect on a species, large collection of summaries must be compared.

Our goal is to develop novel scalable exploratory analysis tools and algorithms in order to help scientists search for potentially interesting hypotheses in very large, high-dimensional data sets. We focus on the join operation which is essential for detecting correlations and relationships between patterns in scientific data [62]. Consider the example in Figure 1.1b that points out habitat competition between two species. In order to be able to capture such relationships and correlations, large collection of summaries must be joined with flexible join predicates. For example, the following query is executed to search for habitat competition between species:
1.1. CHALLENGES AND CONTRIBUTIONS

\[
\text{SELECT } S.\text{plotData}, T.\text{plotData}, S.\text{sumAttr} \\
\text{FROM } \text{Summaries AS S, Summaries AS T} \\
\text{WHERE } S.\text{species} \neq T.\text{species} \\
\quad \text{AND } S.\text{sumAttr} = T.\text{sumAttr} \\
\quad \text{AND } S.\text{region} = T.\text{region} \\
\quad \text{AND } |\phi(S.\text{plotData}, T.\text{plotData})| > \epsilon
\]

This query is a self-join to find summaries of different species having the same summary attribute (x-axis of the plot) and complementary trends in the same region where \( \phi \) is a user-defined distance function and \( \epsilon \) is a threshold parameter set by the user. A variety of other non-equality conditions are necessary for scenarios where scientists may investigate correlations among summaries with different summary attributes in various regions.

1.1 Challenges and Contributions

Given two sets \( S \) and \( T \), the join operation returns the set of all pairs \((s, t)\) that satisfy some join condition \( C(s, t) \), where \( s \in S, t \in T \), and \( C \) is a Boolean function over the attributes of \( S \) and \( T \). A join operation with such a general join condition \( C \) is called a theta-join \([31]\). The join operation is one of the fundamental relational operations and frequently used to combine information from multiple sources.

New computing platforms have emerged to allow large volumes of data to be organized and processed on clusters of machines \([22, 21, 41, 5, 11, 15]\). Joins must be supported by these emerging platforms in order to be able to efficiently perform many large-scale data analysis applications. There has been progress on how to compute equi-joins, a special case of theta-joins where the join predicate is equality. These approaches are mostly based on symmetric partitioning where the input datasets are split into smaller independent partitions based on the join attribute value. This allows each node in the cluster to work on a partition to compute a subset of the join output independently, achieving parallel execution. However,
distributing the workload to multiple nodes is more challenging for arbitrary theta-join conditions.

In order to support arbitrary theta-join predicates, we define a join model that simplifies creation of and reasoning about possible join algorithms. Algorithms derived from the join model differ in how the input data is distributed across the nodes in order to correctly compute the join. This model allows us to derive different parallel theta-join algorithms and it also enables us to reason about the number of input records to be processed and the largest possible number of output records that can be computed at each node in advance.

A major challenge for achieving efficient and scalable theta-join processing in parallel is to balance workload among the nodes in the cluster. Moreover, the dominating cost of a join problem may depend on several factors such as network transfer time, CPU-time, or local I/O-time based on the join condition, input data distribution, join algorithm executed on each node, and the system configuration. Therefore, it is crucial to balance the dominating cost of a given join problem.

Using our join model, we perform an in-depth analysis on how to process arbitrary theta-join predicates efficiently in parallel when the goal is to minimize job completion time. We classify joins as input- and output-size dominated joins since all different types of costs mentioned above tend to increase with increasing input and/or output size. Hence, balancing input and/or output size minimizes job completion time, no matter if network, CPU, or local I/O is the bottleneck.

We propose a randomized join algorithm called 1-Bucket-Random that achieves near-optimal execution time for output-size dominated join problems. This algorithm is capable of computing any theta-join, including the cross-product. It only needs minimal input statistics (cardinality ratio of input sets) and still effectively parallelizes the computation. 1-Bucket-Random can be improved by exploiting input statistics for selective joins. We propose algorithms for a popular class of joins, including equality, inequality and band-joins. We demonstrate the effectiveness of our techniques on a variety of joins using real and synthetic data using MapReduce.

One potential drawback of our join algorithms is a high degree of input replica-
tion depending on the input data distribution, available input statistics, join condition, and cluster properties. High input replication increases the network load, potentially decreasing the throughput of the distributed system. While shared-nothing environments (on which data-intensive computing platforms such as MapReduce tend to be executed) make it easy to increase CPU, memory, and disk resources by adding more machines, this is difficult for the network. Network links and switches are in fact shared resources in the sense that the same link or switch is on the path between many pairs of machines. Hence reducing network load is essential for increasing throughput in highly utilized environments.

We study optimizing the network transfer cost of programs that are executed using the MapReduce programming model. We propose optimization techniques that reduce the amount of data transfer between mapper and reducer nodes, when often massive amounts of data is moved across the network.

Our key insight for reducing the amount of data transferred from mappers to reducers is that for some input records the Map function will produce output that is significantly larger than the input record. This property is observed for join processing since input records need to be replicated to multiple Reducer nodes in order to process certain join conditions efficiently. We propose lightweight encoding and decoding strategies in order to shift mapper-side processing to the reducers. These techniques are applicable to a large and diverse spectrum of applications including join processing, graph algorithms, and multi-query optimization techniques. We present a performance study of these techniques on joins and query log analysis to show their benefits.

Finally, we integrate the proposed techniques in this dissertation into Scolopax, a novel tool that supports exploratory analysis for data-driven science. Scolopax achieves high performance and scalability for flexible join predicates used by scientists to compute relationships and correlations in high-dimensional scientific data.
1.2 Summary and Dissertation Outline

Data-driven science workloads and many other large-scale data analysis applications require computing arbitrary theta-join conditions. Supporting theta-joins on emerging data-intensive platforms such as MapReduce play a key role in enabling data-driven science.

The remainder of this dissertation is organized as follows: In Chapter 2, we introduce the popular large-scale data processing platform MapReduce, state-of-art techniques used to compute equi-joins using such platforms, and challenges with processing arbitrary join predicates in distributed systems. Chapter 3 presents our techniques to efficiently process joins in parallel using MapReduce. Next, we describe our strategies to reduce the amount of data transferred using MapReduce in Chapter 4. Chapter 5 demonstrates Scolopax, a data exploration tool we developed that enables scientists to create hypothesis search queries. Chapter 6 covers the related work in the field and identifies the distinctions of our work. We conclude and present some ideas for future work in Chapter 7.
CHAPTER 2

Preliminaries

Join is one of the most important operations in relational database management systems. It has also emerged as an important operation for large-scale data analytics. For instance in log processing, a large amount of log data is often joined with different reference data sets in order to derive business insights [13], data warehousing [18], data mining [50, 42, 29], time series analysis [9, 10], pattern analysis on social networks, clustering [14, 16], geographic information systems [70] and bioinformatics research. Existing well-known equi-join strategies from parallel and distributed database systems are implemented on large-scale data processing platforms such as MapReduce in order to deal with a massive amount of data. We focus on shared-nothing architectures where the data is partitioned across a set of machines with each machine having sole access to the data it holds.

In this chapter we first introduce the popular large-scale data processing platform MapReduce, and a commonly used algorithm for computing equi-joins using such shared-nothing parallel data processing platforms. Then, we present challenging problems with processing theta-joins efficiently using MapReduce through examples.

2.1 Overview of MapReduce

MapReduce was proposed to simplify large-scale data processing on distributed and parallel architectures, particularly clusters of commodity hardware [22]. The
main idea of this programming model is to hide details of data distribution, fault-tolerance and load balancing and let the user focus on the data processing aspects.

MapReduce has emerged as a popular paradigm for parallel processing, and it has had a great impact on data management research. There are several reasons behind its success. One major reason is the availability of a free open-source implementation, Hadoop [1], and an active developer community that keeps making improvements and adding features. Also, its flexibility to work on unstructured data makes it preferable over parallel database management systems for certain analysis tasks. MapReduce is fault tolerant and it is preferable for tasks that process vast amounts of data since only failing nodes are restarted instead of restarting the query which often occurs with parallel database management systems [13].

The data is stored as key/value pairs on a distributed file system (DFS), partitioned across a cluster of nodes. A MapReduce program consists of two major primitives, Map and Reduce:

\[
\text{map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2) \\
\text{reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(k_3, v_3)
\]

The Map function is applied to an individual input key/value pair in order to compute a set of intermediate key/value pairs. For each key, Reduce works on the list of all values with this key. Any output produced by Reduce is written to a file stored in a distributed file system.

An overview of the MapReduce architecture is given in Figure 2.1. Input records might be distributed across several physical locations on a DFS. Once the MapReduce job is initialized, these records are transferred to mapper nodes in chunks. Note that if the mapper node does not have the chunk locally, the data is transferred from the DFS (step 1). The local input is scanned in the Map phase (step 2) and the map function is applied to each record. The output of the map function call is collected in a buffer before storing it on local disk. A background thread starts to spill the buffer content when the buffer reaches a spill threshold. Before map output is written to disk, the data is partitioned using the user-defined partition
2.1. **OVERVIEW OF MAPREDUCE**

function (or default hash partitioner) that assigns map output records to reducers. Each partition is sorted by key in memory before they are written to the spill file (step 3). Before the map task is finalized, the spill files are merged, preserving the sort order for each partition (step 4). Before the reduce phase starts, the reducer nodes read assigned partitions of map outputs residing at multiple nodes across the cluster. The map outputs are transferred to the reducer nodes and written to disk (step 5). Multiple files transferred from mapper nodes are merged preserving the sort order (step 6). The process starting from step 3 until the end of step 6 where all the reducers have their input in sorted order is referred as the shuffle-and-sort phase [80]. Note that a reducer must have received all its input data in order to start the reduce phase. In other words, the map phase has to complete before reduce phase can start. In the reduce phase, for each key in sorted order, the reduce function is invoked for the set of values that share the key being processed (step 7). Finally, the reduce output is written to DFS.

If the Reduce function is commutative and associative, users can define a Combine function which is applied to local map output before it is sent over the network for the Reduce phase. This reduces the amount of intermediate key/value pairs produced by each map task since aggregated values of the same key are merged partially before they are sent over the network.
The performance of MapReduce is being criticized for ignoring many optimization techniques used in parallel database management systems. Pavlo et al. [67] showed that MapReduce-based systems are slower than parallel database systems for a variety of analytic tasks. Several techniques have been proposed in order to improve the performance of MapReduce programs. Olston et al. [63] highlighted database techniques that can be made compatible with parallel dataflow programs. Jiang et al. [44] identified different factors affecting the performance of MapReduce and evaluated multiple optimization techniques such as exploiting indexes, different decoding schemes for data deserialization, and fingerprinting to reduce the cost of key comparison. Li et al. [55] showed that merge sort used by MapReduce before the reduce phase is I/O intensive and affects the performance of MapReduce. The authors propose to use hashing techniques for better performance and for one-pass analytics. Index based solutions have also been proposed to reduce I/O cost in MapReduce [5, 28]. Multi-query optimization techniques such as data sharing are also applied to MapReduce programs [60, 17, 54]. Multiple jobs that share the same input are merged into a single job to avoid redundant I/O [34, 76]. Scheduling frameworks have been proposed to execute shareable jobs while meeting individual job deadlines [8, 79]. Wolf et al. [82] proposed an alternative scheduler where jobs are decomposed into sub-jobs and cyclic piggybacking is performed instead of batching for sharing scans. Jahani et al. [43] studied a static analysis-style mechanism for automatic detection of selection, projection, and data compression optimizations in MapReduce programs.

High-level declarative languages have been proposed that come with compilers for translating queries into plain MapReduce code in order to simplify distributed programming [12, 20, 64, 68, 85]. Translation of queries into execution plans introduce opportunities for automatic optimization [63]. Lee et al. [53] provide a detailed survey of the recent literature attempting to optimize the MapReduce framework.
2.2 Parallel Join Processing using MapReduce

Many research studies have been devoted to join algorithms in parallel and distributed database systems. Blanas et al. [13] studies performance of several well-known equi-join strategies for MapReduce. In this section, we present repartition join which is the most commonly used equi-join strategy in MapReduce when both data sets are too big to fit in a single machine’s memory.

Consider an equi-join of data sets $S$ and $T$ on a common attribute $A$, i.e., join condition $S.A = T.A$. Repartition join dynamically partitions both $S$ and $T$ on the join key in the map phase, and the corresponding partitions are joined in the reduce phase. More precisely, for each input tuple $s \in S$, Map outputs the key-value pair $(s.A, s)$. Notice that $s$ is also augmented by adding an attribute origin which indicates that the tuple came from $S$. Tuples from $T$ are processed similarly. For each join attribute value, Reduce then computes the cross-product between the corresponding tuples that have origin $S$ with the tuples whose origin is $T$. This process is illustrated in Figure 2.2 for a subset of input tuples having join attribute value equal to 1.
2.2.1 Limitations and Challenges

The repartition join algorithm presented in the previous section suffers from various problems. First, the degree of parallelism is limited by the number of distinct values of $A$ in the input data sets. The reason is that all tuples with the same $A$-value have to be processed by the same invocation of Reduce. The second problem is caused by data skew. If some $A$-value occurs very frequently, the reducer processing it receives an overly large share of work, both for processing the input and writing the large result to the DFS, thus delaying the completion of the job.

More importantly, this algorithm does not generalize to other joins, e.g., inequality joins. Consider a theta-join between data sets $S$ and $T$ with an inequality condition like $S.A \leq T.A$. Such joins seem inherently difficult for MapReduce, because each $T$-tuple has to be joined not only with $S$-tuples that have the same $A$ value, but also those with different (smaller) $A$ values. It is not obvious how to map the inequality join condition to a key-equality based computing paradigm.

One might consider the following “naive” approach. Assume all values of $A$ are non-negative integers. To ensure that a $T$-tuple joins with all $S$-tuples of equal or smaller $A$-value, we can make Map output each $T$-tuple for all possible smaller $A$-values as keys. More precisely, for each input tuple $s \in S$, Map only outputs $(s.A, s)$, but for each $t \in T$, it outputs $(a, t)$ for every $a \leq T.A$. It is easy to see that for any $(s, t)$ satisfying $S.A \leq T.A$, the reducer working on key $s.A$, and only this reducer, will produce this output tuple. Hence the join is correctly implemented.

However, this algorithm also suffers from three major problems. First, it generates a potentially huge amount of “duplicates” of $T$-tuples that depends on the values of the join attribute. Second, if attribute $A$ is not integer or can have negative values, one cannot enumerate all smaller values of $A$ for a given value $t.A$. For these cases, the Map function needs to know the set of distinct values of $A$ in $S$ to produce the right duplicates of $T$. Finally, this approach does not generalize to other theta-join types. We propose a general framework that works for any theta-join in Chapter 3.
CHAPTER 3

Processing Theta-Joins using MapReduce

As described in Chapter 2, the main challenges for implementing joins in distributed systems are handling data skew and the difficulty of data partitioning for non-equi-join predicates. In this chapter, we show how to process arbitrary theta-join conditions efficiently in parallel using MapReduce. In Section 3.1, we propose a reducer-centered cost model based on total run-time of a given job. Using this model, we propose to balance the dominating cost among the reducer nodes. We also discuss how this approach generalizes to other shared-nothing systems beyond MapReduce. In Section 3.2, we propose a join model that simplifies creation of and reasoning about possible join implementations in MapReduce. This model allows us to perform data partitioning for arbitrary join predicates. We propose a randomized algorithm called 1-Bucket-Random for computing any join, including the cross-product, in a single MapReduce job in Section 3.3. This algorithm only needs minimal input statistics (cardinality ratio of the input data sets) and still effectively parallelizes the computation. We show that it is close to optimal for joins with large output size. For highly selective joins, we show that even though better implementations in MapReduce might exist, they often cannot be used, leaving 1-Bucket-Random as the best available option. In Section 3.4, we propose algorithms for a popular class of highly selective joins, including equality, inequality and band-joins, that often improve on 1-Bucket-Random, as long as sufficiently detailed input statistics are available. No modifications to the MapReduce environment are necessary, and the user does not have to write any special-purpose code to
manage data flow. Everything is automatically achieved by simply specifying the appropriate Map and Reduce functions. We demonstrate the effectiveness of our techniques on a variety of joins using real and synthetic data in Section 3.5.

### 3.1 Cost Model for Joins in MapReduce

In the following analysis, we generally assume that all computing nodes in the cluster by design have approximately the same computational capabilities. This holds in practice for clusters of virtual machines created in the Cloud, but also for physical clusters running on commodity hardware.

For a given join operator and its inputs, we want to **minimize job completion time** for a given number of processing nodes. Job completion time includes all phases of MapReduce, from when the first data tuple is transferred to a mapper node until the last output tuple is written back to the DFS. Short job completion time is desirable from a user’s point of view. As we discuss below, it also inherently leads to a load-balancing approach. This is in line with previous work on distributed and parallel systems, where load balancing ideas usually play a central role.

We observe that all join-related costs can be expressed as functions of reducer input and output. Since the cost for transferring data from the DFS to the mapper nodes and the cost for reading the input tuples locally at each mapper is not affected by the concrete Map and Reduce functions, we do not need to take these costs into account for the optimization. Map and Reduce functions affect the costs from producing Map function output until writing the final join result back to the DFS. To analyze the completion time of these MapReduce job phases, consider a single reducer. As introduced in Section 2.1, the reducer receives a subset of the mapper output tuples as its input. It sorts the input by key, reads the corresponding value-list for a key, computes the join for this list, and then writes its locally created join tuples to the DFS. The cost of some of these operations depend on input e.g., map output transfer and sorting. Some operations affect output-related costs e.g., writing output results to DFS. The cost of the actual join computation depends on
3.1. **COST MODEL FOR JOINS IN MAPREDUCE**

both input and output size. We make the assumption that the response time of the reducer (from receiving the first record until finishing writing the output) is monotonic in input and output size. This holds approximately in many cases, if not always true.

If input-related costs dominate the computation time, then we have to minimize \( \text{max-reducer-input} \), i.e., the maximum over the input sizes assigned to any reducer. This will minimize the completion time of the reduce phase and is achieved by balancing input as evenly as possible across the reducers. Similarly, if output-related costs dominate, then we need to minimize \( \text{max-reducer-output} \), i.e., the maximum over the output size produced by any reducer. That is achieved by balancing output evenly over all reducers. Finally, if both input and output-related costs are of similar magnitude, then ideally both should be balanced.

Blanas et al. showed that the Map phase takes longer than the Reduce phase for a special n-to-1 equi-join used for log processing [13]. For such cases where the Map phase is the bottleneck, algorithms based on our cost model neither improve nor worsen the performance. However, for theta-joins bottleneck is often the Reduce phase. Our experiments also support our cost model for various join operators including equi-join.

Notice that our cost model makes no assumption about which type of cost dominates, i.e., it includes cases where network transfer time, CPU-time, or local I/O-time dominate. All these costs tend to increase with increasing input and/or output size, hence balancing input and/or output size minimizes job completion time, no matter if network, CPU, or local I/O is the bottleneck.

We say that a join problem is **input-size dominated** if reducer-input related costs dominate job completion time. If reducer-output related costs dominate job completion time, then the join problem is **output-size dominated**. If neither clearly dominates the other, we have an **input-output balanced** problem. Notice that the join problem category depends on the specific join implementation selected.

How does this cost model generalize to other distributed systems? Data stored in distributed systems is usually decomposed into subsets, called fragments. Frag-
Figure 3.1: Join matrices for equi-join, similarity-join, and inequality-join. Numbers indicate join attribute values from $S$ and $T$, shaded cells indicate join results. $M(i, j)$ indicates cell numbering.

Implementation allows parallel execution of a query by executing subqueries, each operating on a different fragment. Certain operations such as joins require retrieving data from multiple fragments. For example, in order to compute cross-product of two relations, all pairs of fragments must be matched at some node. Based on the query and the data distribution, different allocation algorithms can be used to minimize data communication.

In MapReduce, as introduced in Section 2.1, the data is also fragmented across the nodes of the DFS. Specified sized chunks of fragments are assigned to mapper nodes. However, the data distribution within these chunks and which physical mapper node will work on which set of chunks are not known in advance. Therefore, our cost model assumes that all the data transfer between mapper nodes and reducer nodes go through the network although some of the mapper output may be assigned to a reducer node which already has the data available locally.

Our cost model is applicable beyond MapReduce, to any shared-nothing system. However, systems with detailed data fragmentation statistics can exploit data locality and reduce input transfer cost which is not applicable in MapReduce context. For output-size dominated join problems, our cost model is applicable beyond MapReduce.
3.2 Join Model

We model a join between two data sets $S$ and $T$ with a join-matrix $M$ and employ this representation for creation of and reasoning about different join implementations in MapReduce. Figure 3.1 shows example data sets and the corresponding matrix for a variety of join predicates. For row $i$ and column $j$, matrix entry $M(i, j)$ is set to true (shaded in the picture) if the $i$-th tuple from $S$ and $j$-th tuple from $T$ satisfy the join condition, and false (not filled) otherwise. Since any join is a subset of the cross-product, the matrix can represent any join condition.

Our goal is to have each join output tuple be produced by exactly one reducer, so that expensive post-processing or duplicate elimination is avoided. Hence, given $r$ reducers we want to map each matrix cell with value $M(i, j) = \text{true}$ to exactly one of the $r$ reducers. We will also say that reducer $R$ covers a join matrix cell, if this cell is mapped to $R$.

There are many possible mappings that cover all true-valued matrix cells. Our goal is to find that mapping from join matrix cells to reducers that minimizes job completion time. Hence we want to find mappings that either balance reducer input share (for input-size dominated joins), or balance reducer output share (for output-size dominated joins), or achieve a compromise between both (for input-output balanced joins).

Figure 3.2 illustrates the tradeoffs in choosing different mappings. The left image is the mapping used by the repartition join implementation in MapReduce. All tuples with the same join attribute value are mapped to the same reducer. This results in a poor balance of both reducer input and output load. E.g., reducer R2 receives 5 input tuples and creates 6 output tuples, while reducer R3 works with 3 input and 2 output tuples. Note that this algorithm cannot be improved with different partition functions because of the reduce call for key 7. The reducer receives values for key 7 will have at least 5 input and 6 output records.

The other two images correspond to new equi-join algorithms that we have not seen in the literature before. (Using our formulation of join implementations as a
mapping from \texttt{true} matrix entries to the set of reducers, it is easy to come up with many more algorithms.) The center image represents a very fine-grained mapping. Even though the 5-th and 6-th tuple from $S$ ($S_5, S_6$) and the 6-th tuple from $T$ ($T_6$) all have the same join attribute value, result tuple ($S_5, T_6$) is produced by reducer $R_2$, while ($S_6, T_6$) is produced by reducer $R_1$. The example also illustrates how skew is effectively addressed, e.g., by breaking the big output chunk for tuples with join value 7 into many small pieces. The downside of the better output load balancing is the significantly greater input size for every reducer, caused by the duplication of tuples to enable each reducer to generate the desired results. E.g., the second and third tuples from $S$ have to be sent to all three reducers. Also notice that both $R_2$ and $R_3$ could produce outputs ($S_2, T_2$) and ($S_3, T_2$), because they both have the corresponding input tuples. To enforce the matrix-to-reducer mapping (and avoid duplicate output), the algorithm would have to pass information about the mapping to each reducer.

The mapping on the right illustrates how we can achieve the best of both worlds. Overly large output chunks are effectively broken up, while input duplication is kept low and reducer input and output are both well-balanced. This is achieved

Figure 3.2: Matrix-to-reducer mappings for repartition join algorithm (left), random (center), and balanced (right) approach
3.3. THE 1-BUCKET-RANDOM ALGORITHM

despite the mapping covering not only true cells, but also some like \( M(2, 1) \) that do not contribute to the join output. (Those do not affect the join result, because Reduce eliminates them.) Our new algorithms represent practical implementations of this basic idea: balance input and output costs while minimizing duplication of reducer input tuples. We will repeatedly make use of the following important lemma.

**Lemma 3.2.1.** A reducer that is assigned to \( c \) cells of the join matrix \( M \) will receive at least \( 2\sqrt{c} \) input tuples.

**Proof.** Consider a reducer that receives \( m \) tuples from \( S \) and \( n \) tuples from \( T \). This reducer can cover at most \( m \cdot n \) cells of the join matrix \( M \). Hence to cover \( c \) matrix cells, it has to hold that \( m \cdot n \geq c \). Considering all possible non-negative values \( m \) and \( n \) that satisfy \( m \cdot n \geq c \), the sum of \( m \) and \( n \) is minimized for \( m = n = \sqrt{c} \). \( \square \)

Note that in these examples we assumed that all true-valued cells are known for illustration purposes, which is not realistic. In Section 3.3, we provide a solution that does not require any prior knowledge about the matrix cells. We also discuss how to find to-be-covered cells by using histograms in Section 3.4.

### 3.3 The 1-Bucket-Random Algorithm

The examples in Section 2.2.1 illustrate the challenges for implementing joins in distributed systems: data skew and the difficulty of data partitioning for non-equi-join predicates. We now introduce 1-Bucket-Random, an algorithm that addresses these challenges, and provide strong analytical results about its properties.

#### 3.3.1 Implementing the Cross-Product

Since the cross-product combines every tuple from \( S \) with every tuple from \( T \), the corresponding join matrix has all entries set to true. We explain how 1-Bucket-Random performs matrix-to-reducer mapping, show that it is near-optimal for com-
puting the cross-product, and discuss how these results extend to processing of theta-joins.

### 3.3.1.1 Analytical Results

To minimize max-reducer-output given \( r \) reducers, each reducer should produce exactly \(|\mathcal{S}||\mathcal{T}|/r\) tuples of the cross-product. (As usual, \(|\mathcal{S}|\) denotes the cardinality of a set \(\mathcal{S}\).) To achieve this lower bound for max-reducer-output, the matrix-to-reducer mapping has to partition the matrix such that exactly \(|\mathcal{S}||\mathcal{T}|/r\) of the matrix cells are mapped to each of the \(r\) reducers. Notice that if cell \(M(i, j)\) is assigned to reducer \(k\), then reducer \(k\) needs both the \(i\)-th tuple from \(\mathcal{S}\) and the \(j\)-th tuple from \(\mathcal{T}\) to be able to create the combined output tuple. Hence the \(i\)-th tuple from \(\mathcal{S}\) has to be sent to each reducer whose region in the matrix intersects the \(i\)-th row. Similarly, the \(j\)-th tuple from \(\mathcal{T}\) has to be sent to all reducers whose regions intersect the \(j\)-th column of \(M\). As Figure 3.4 illustrates, depending on the number of different reducers assigned to cells in each row and column, input tuples might be duplicated many times.

Lemma 3.2.1 implies the following for any mapping that balances reducer output size to \(|\mathcal{S}||\mathcal{T}|/r\) per reducer: The max-reducer-input cannot be lower than \(2\sqrt{|\mathcal{S}||\mathcal{T}|/r}\). As the following theorem shows, for some special cases we can actually match this lower bound with a square-based matrix-to-reducer mapping.

**Theorem 3.3.1.** For \(\mathcal{S} \times \mathcal{T}\), consider matrix-to-reducer mappings that perfectly balance the entire output to \(|\mathcal{S}||\mathcal{T}|/r\) tuples per reducer. Let (1) the number of reducers be a square, i.e., \(\sqrt{r}\) is an integer and let (2) \(|\mathcal{S}|\) and \(|\mathcal{T}|\) be multiples of \(\sqrt{|\mathcal{S}||\mathcal{T}|/r}\), i.e., \(|\mathcal{S}| = c_S\sqrt{|\mathcal{S}||\mathcal{T}|/r}\) and \(|\mathcal{T}| = c_T\sqrt{|\mathcal{S}||\mathcal{T}|/r}\) for integers \(c_S, c_T > 0\). Under these conditions, max-reducer-input is minimized by partitioning the matrix into \(c_S\) by \(c_T\) squares of size \(\sqrt{|\mathcal{S}||\mathcal{T}|/r}\) by \(\sqrt{|\mathcal{S}||\mathcal{T}|/r}\) each.

For other examples where \(|\mathcal{S}|, |\mathcal{T}|,\) and \(r\) do not satisfy the properties required by Theorem 3.3.1, the problem can be formulated as an integer linear programming problem. These problems are generally NP-hard, hence it can be expensive to solve.
3.3. THE 1-BUCKET-RANDOM ALGORITHM

We partition the matrix into $c_S$ by $c_T$ squares where $c_S$ and $c_T$ are integers, $c_S, c_T \geq 1$ and $c_S c_T \leq r$. Since $|S|$ and $|T|$ may not be multiples of $\sqrt{|S||T|}/r$, we define $c_S = \left\lfloor |S|/\sqrt{|S||T|}/r \right\rfloor = \left\lfloor |S| r/|T| \right\rfloor$, and $c_T = \left\lfloor |T|/\sqrt{|S||T|}/r \right\rfloor = \left\lfloor |T| r/|S| \right\rfloor$. Note that $c_S$ and $c_T$ can be computed using the ratio $\alpha = |S|/|T|$, i.e., $c_S = \lfloor \sqrt{\alpha r} \rfloor$ and $c_T = \lfloor \sqrt{r/\alpha} \rfloor$ where $1/r \leq \alpha \leq r$. Note that the conditions on $c_S$ and $c_T$ are ensured by the boundaries of $\alpha$.

Figure 3.3 illustrates a partitioning using $c_S$ by $c_T$ squares, each having width $d = \sqrt{|S||T|}/r$. After creating these squares, starting in the upper left corner of the matrix, there might be some cells that are not covered. Let the width of the remaining rows be $e_S$, and length of the remaining columns be $e_T$ such that $|S| = c_S d + e_S$ and $|T| = c_T d + e_T$. We can cover the remaining cells by increasing the length of all squares by $e_T/c_T$, and increasing the width of all squares by $e_S/c_S$. After expansion, each rectangle have width $d + e_S/c_S$ and length $d + e_T/c_T$. Since $e_S, e_T < d$, each rectangle have width less than $(1 + 1/c_S)d$ and length less than $(1 + 1/c_T)d$. Recall that the lower bound for max-reducer-input is $2d$. Therefore, the max-reducer-input found by the partitioning is always less than twice the lower bound since $c_S, c_T \geq 1$. Since both dimensions of the rectangles are
bounded by \(2d\), the area of the rectangles cannot exceed \(4d^2 = 4|S||T|/r\). Therefore, the max-reducer-output found by the partitioning is always less than four times the lower bound. Note that for \(c_S, c_T \geq c\), the max-reducer input does not exceed \((1 + 1/c)\) times the lower bound, and the max-reducer output does not exceed \((1 + 1/c)^2\) times the lower bound.

Consider the remaining two extreme cases where \(\alpha < 1/r\) and \(\alpha > r\). If \(\alpha < 1/r\), we set \(c_S = 1\) and \(c_T = r\). Since \(|S| < |T|/r\), and \(|S| < \sqrt{|S||T|}/r\), i.e. the side length of the optimal square that matches the lower bounds is “taller” than the join matrix. Hence the lower bounds are not tight, because no partition of the matrix can have more than \(|S|\) tuples from input set \(S\). It is easy to see the optimal partitioning of the matrix into \(r\) regions would then consist of rectangles of size \(|S|\) by \(|T|/r\). For the other extreme case where \(\alpha > r\), we set \(c_S = r\) and \(c_T = 1\). Similar to the previous case, the optimal partitioning of the matrix into \(r\) regions consist of rectangles of size \(|S|/r\) by \(|T|\).

The following theorems summarize these results.

**Theorem 3.3.2.** For any \(S \times T\), we can always find a matrix-to-reducer mapping with the following properties. (1) No reducer produces more than \(4|S||T|/r\) output tuples, (2) if \(c_S, c_T \geq c\) then no reducer produces more than \((1 + 1/c)^2|S||T|/r\) output tuples, and (3) No reducer receives more than \(4\sqrt{|S||T|}/r\) input tuples, (4) if \(c_S, c_T \geq c\) then no reducer receives more than \(2(1 + 1/c)\sqrt{|S||T|}/r\) input tuples.

**Theorem 3.3.3.** For \(S \times T\), where \(\alpha < 1/r\) or \(\alpha > r\) our method always finds a matrix-to-reducer mapping with optimal max-reducer-input and max-reducer-output.

### 3.3.1.2 Significance of Analytical Results

Together, Theorems 3.3.1, 3.3.2, and 3.3.3 give us strong guarantees for the near-optimality of our matrix-to-reducer mappings for implementing the cross product. Using only \(|S|\), \(|T|\), and \(r\) as input, we can always find a solution where none of the reducers receives more than four times its “fair share” of the output-related cost.
3.3. THE 1-BUCKET-RANDOM ALGORITHM

practice this bound is usually much better (e.g., if \( c_S \) and \( c_T \) as introduced above are in the order of 10, then no reducer receives more than 1.21 times its fair output share). At the same time, it is guaranteed that none of the reducers receives more than twice its “fair share” of the input-related cost. Similar to output-related cost, in practice this bound is much better in general (e.g., if \( c_S \) and \( c_T \) are in the order of 10, then no reducer receives more than 1.1 times its fair input share).

To summarize our analytical results: We identified matrix-to-reducer mappings that balance reducer output costs within a small constant factor \((1 + \frac{1}{\min\{c_S, c_T\}})^2 \leq 4\) of the lower bound \(|S||T|/r\). At the same time these mappings balance reducer input costs within a small constant factor \(2(1 + \frac{1}{\min\{c_S, c_T\}} < 2\) of the lower bound \(2\sqrt{|S||T|/r}\). The run-time complexity of our algorithm is \(O(r)\).

3.3.1.3 From Mapping to Randomized Algorithm

Turning a matrix-to-reducer mapping into a MapReduce algorithm is tricky. Consider the partitions computed by 1-Bucket-Random in Figure 3.4 for computing a join (using the cross-product of the data sets) using 4 reducers. Since all regions have the same area \((|S||T|/4)\), this would guarantee that each reducer is responsible for its exact fair share of 1/4 of the cross-product space. As illustrated for \(s_i\) and \(t_j\), by assigning each tuple to all intersecting reducers on its row/column, we guarantee to match each pair of \(S\) and \(T\) tuples at exactly one reducer (\(s_i\) and \(t_j\) matches at reducer 2). Ideally, the Map phase should assign 1/2 of the \(S\)-tuples to reducers 1 and 2, and the remaining 1/2 of the \(S\)-tuples to reducers 3 and 4. Similarly, half of \(T\)-tuples should be assigned to reducers 1 and 3, and the other half to reducers 2 and 4. The problem is that nothing is known in advance about the input distribution (except the assignment ratios of each data set) to make row/column assignment based on input records. Each invocation of Map by design sees only a single \(S\) or \(T\) tuple. Therefore, there is no consensus mechanism between mappers to guarantee that the total number of records assigned to reducers in parallel is going to match
CHAPTER 3. PROCESSING THETA-JOINS USING MAPREDUCE

Figure 3.4: Matrix-to-reducer mapping example. The join matrix is partitioned into 4 regions, each assigned to one reducer.

the matrix-to-reducer mapping.

In order to address this problem, we propose to make Map a randomized algorithm as shown in Algorithm 1. It can implement any theta-join of two input sets $S$ and $T$. For each incoming $S$-tuple, Map draws a random integer between 0 and $c_S - 1$, and assigns the tuple to the corresponding row in matrix $M$. For instance, $S$-tuples in Figure 3.4 are assigned to the first row (reducers 1 and 2) if the random number generated is 0, or to the second row (reducers 3 and 4) otherwise. Similarly, $T$-tuples are assigned to the first column (reducers 1 and 3) if the random number generated is 0, or to (reducers 2 and 4) otherwise. Map then creates an output tuple for each region that intersects with this row (augmented by the origin attribute as described earlier). The matrix-to-reducer mapping ($c_S$ and $c_T$) can also be computed before this MapReduce job starts and used by all nodes.

While our randomized algorithm does not guarantee the desired input and output size for each reducer any more, the large data size makes significant variations extremely unlikely. Consider a reducer node $R$ that should receive $n_R$ of the $|S|$ tuples from $S$ based on the matrix-to-reducer mapping. E.g., in Figure 3.4 reducer 1 should receive 1/2 of the $S$-tuples. Let $X_1, X_2, \ldots, X_{|S|}$ be random variables, such that $X_i = 1$ if the $i$-th tuple from $S$ is assigned to this reducer $R$, and 0
3.3. **THE 1-BUCKET-RANDOM ALGORITHM**

Algorithm 1 : Map (Theta-Join)

**Require:** input tuple \( x \in S \cup T, \alpha \)

1. if \( \alpha < 1/r \) then
2. \( c_S = 1, c_T = r \)
3. else if \( \alpha > r \) then
4. \( c_S = r, c_T = 1 \)
5. else
6. \( c_S = \lfloor \sqrt{\alpha r} \rfloor, c_T = \lfloor \sqrt{r/\alpha} \rfloor \)
7. end if
8. if \( x \in S \) then
9. \( \text{row} = \text{randomInteger}(0, c_S) \{ \text{row} \in [0, c_S) \} \)
10. for \( i = 0 \) to \( i < c_T \) do
11. Output (\( \text{row} * c_T + i, (x, "S") \) )
12. end for
13. else
14. \( \text{col} = \text{randomInteger}(0, c_T) \{ \text{col} \in [0, c_T) \} \)
15. for \( i = 0 \) to \( i < c_S \) do
16. Output (\( i * c_T + \text{col}, (x, "T") \) )
17. end for
18. end if

Algorithm 2 : Reduce (Theta-Join)

**Require:** \((\text{ID}, [(x_1, \text{origin}_1), (x_2, \text{origin}_2), \ldots, (x_k, \text{origin}_k)])\)

1. \( \text{Stuples} = \emptyset; \text{Ttuples} = \emptyset \)
2. for all \((x_i, \text{origin}_i)\) in input list do
3. if \( \text{origin}_i = \text{"S"} \) then
4. \( \text{Stuples} = \text{Stuples} \cup \{x_i\} \)
5. else
6. \( \text{Ttuples} = \text{Ttuples} \cup \{x_i\} \)
7. end if
8. end for
9. joinResult = MyFavoriteJoinAlg(Stuples, Ttuples)
10. Output( joinResult )

otherwise. Since \( S \)-tuples are assigned to rows uniformly at random, we obtain \( p_i = \Pr[X_i = 1] = n_R/|S| \). The \( X_i \) are independent random variables, therefore the Chernoff Bound gives us for any \( \delta > 0 \)

\[
\Pr[X \geq (1 + \delta)\mu] \leq e^{-\mu((1+\delta)\ln(1+\delta) - \delta)}.
\]

Here \( X = \sum_i X_i \), i.e., \( X \) is the number of \( S \)-tuples assigned to reducer \( R \). And \( \mu = \mathbb{E}[X] \) is the expected number of tuples assigned to that reducer.
In practice this bound is very tight as long as the reducer with the greatest input share receives on expectation a sufficiently large number of tuples. This is usually the case in practice when dealing with large data sets. For example, let $|S| = 10,000,000$. If reducer $R$ is supposed to receive 10,000 out of 10,000,000 $S$-tuples, the probability that it will receive an extra 5% or more input tuples is $5 \times 10^{-6}$, i.e., virtually zero. Since it is almost impossible for a reducer to receive more than 1.05 times of its target input, its output is practically guaranteed to not exceed $1.05^2 = 1.1$ times its target size. Our experiments support this result.

The alert reader will have realized that considering the cross-product does not seem like the most efficient algorithm for the equi-join example in Figure 3.4. In fact, the three regions in the right example in Figure 3.2 provide a superior partitioning that avoids covering matrix cells that are not part of the join result. We explore this issue in more depth in Sections 3.3.2 and 3.4.

### 3.3.2 Implementing Theta-Joins

Algorithms 1 and 2 can implement any theta-join by selecting the appropriate MyFavoriteJoinAlg for Reduce. While we showed above that this MapReduce implementation is close to optimal for cross-product computation, this result does not necessarily carry over to arbitrary joins. **This section provides strong evidence that even for very selective join conditions, it is often not possible to find a better algorithm than 1-Bucket-Random.** This does not mean that such better algorithms do not exist. They just cannot be identified as correct implementations with the information available at the time when the best implementation is selected for a given join problem, as we show now.

Consider an arbitrary theta-join with selectivity $\sigma$, i.e., it produces $\sigma|S||T|$ output tuples. The lower bound for max-reducer-output is $\sigma|S||T|/r$ join output tuples. While 1-Bucket-Random practically guarantees to balance the cross-product output across reducers, this might not be true for other joins. For example, on some reducer almost all cross-product tuples might satisfy the join condition, while almost
none do so on another. Fortunately, this is very unlikely because of the random-
ization that assigns random samples from $S$ and $T$ to each reducer. Let OUT be
the total output size of a given join problem. For a reducer node $R$, let $s_i$ and $t_j$ be
random variables, and $\delta_{ij}$ be an indicator variable defined as follows:

$$
\begin{align*}
    s_i &= \begin{cases} 
        1 & \text{if } \text{i-th } S\text{-tuple is assigned to } R \\
        0 & \text{otherwise}
    \end{cases} \\
    t_j &= \begin{cases} 
        1 & \text{if } \text{j-th } T\text{-tuple is assigned to } R \\
        0 & \text{otherwise}
    \end{cases} \\
    \delta_{ij} &= \begin{cases} 
        1 & \text{if } (s_i, t_j) \text{ satisfies the join condition} \\
        0 & \text{otherwise}
    \end{cases}
\end{align*}
$$

Let $\text{OUT}_R = \sum_{i,j} s_i t_j \delta_{ij}$ be the output size of reducer $R$. The expected output
size of reducer $R$ is:

$$
\mathbb{E}[\text{OUT}_R] = \mathbb{E}\left[\sum_{i,j} s_i t_j \delta_{ij}\right] = p_S p_T \sum_{i,j} \delta_{ij} 
$$

(3.1)

where $p_S = \Pr[s_i = 1]$, $p_T = \Pr[t_j = 1]$, and $\sum_{i,j} \delta_{ij} = \text{OUT}$. The variance of the
output size of reducer $R$ is:

$$
\text{Var}[\text{OUT}_R] = \mathbb{E}[\text{OUT}_R^2] - \mathbb{E}^2[\text{OUT}_R]
$$

Note that $\text{OUT}_R^2$ is:
\[ \text{OUT}_R^2 = \sum_{i,j} s_i t_j \delta_{ij} \sum_{k,l} s_k t_l \delta_{kl} \]

\[ = \sum_{i,j,k=i,l=j} s_i s_k t_j t_l \delta_{ij} \delta_{kl} + \sum_{i,j,k=i,l \neq j} s_i s_k t_j t_l \delta_{ij} \delta_{kl} + \]

\[ \sum_{i,j,k \neq i,l=j} s_i s_k t_j t_l \delta_{ij} \delta_{kl} + \sum_{i,j,k \neq i,l \neq j} s_i s_k t_j t_l \delta_{ij} \delta_{kl} \]

\[ = \sum_{i,j} s_i^2 t_j^2 \delta_{ij}^2 + \sum_{i,j,l \neq j} s_i^2 t_j t_l \delta_{ij} \delta_{il} + \]

\[ \sum_{i,j,k \neq i} s_i s_k^2 t_j^2 \delta_{ij} \delta_{kj} + \sum_{i,j,k \neq i,l \neq j} s_i s_k t_j t_l \delta_{ij} \delta_{kl} \]

Since each \( s, t, \) and \( \delta \) term can be either 0 or 1, the square of each term can be replaced by that term.

\[ \text{OUT}_R^2 = \sum_{i,j} s_i t_j \delta_{ij} + \sum_{i,j,l \neq j} s_i t_j t_l \delta_{ij} \delta_{il} + \]

\[ \sum_{i,j,k \neq i} s_i s_k t_j \delta_{ij} \delta_{kj} + \sum_{i,j,k \neq i,l \neq j} s_i s_k t_j t_l \delta_{ij} \delta_{kl} \]

The expected value of \( \text{OUT}_R^2 \) is:

\[ \mathbb{E}[\text{OUT}_R^2] = p_s p_T \sum_{i,j} \delta_{ij} + p_s^2 p_T^2 \sum_{i,j,l \neq j} \delta_{ij} \delta_{il} \]

\[ p_s^2 p_T \sum_{i,j,k \neq i} \delta_{ij} \delta_{kj} + p_s^2 p_T^2 \sum_{i,j,k \neq i,l \neq j} \delta_{ij} \delta_{kl} \]

\( \mathbb{E}^2[\text{OUT}_R] \) can be rewritten as follows:
3.3. \textsc{The 1-Bucket-Random Algorithm}

\[ \mathbb{E}^2[\text{OUT}_R] = p_S^2 p_T^2 \sum_{i,j,k,l} \delta_{ij} \delta_{kl} \]

\[ = p_S^2 p_T^2 \left( \sum_{i,j,k=i,l=j} \delta_{ij} \delta_{kl} + \sum_{i,j,k=i,l\neq j} \delta_{ij} \delta_{kl} \right) \]

\[ = p_S^2 p_T^2 \sum_{i,j} \delta_{ij} \left( \delta_{ij} + \sum_{l \neq j} \delta_{il} + \sum_{k \neq i} \delta_{kj} + \sum_{k \neq i,l \neq j} \delta_{kl} \right) \]

\[ = p_S^2 p_T^2 \sum_{i,j} \delta_{ij} \left( 1 + \sum_{l \neq j} \delta_{il} + \sum_{k \neq i} \delta_{kj} + \sum_{k \neq i,l \neq j} \delta_{kl} \right) \]

The variance of the output size of reducer $R$ can be written as:

\[ \text{Var}[\text{OUT}_R] = \mathbb{E}[\text{OUT}_R^2] - \mathbb{E}^2[\text{OUT}_R] \]

\[ = p_S p_T \sum_{i,j} \delta_{ij} + p_S^2 p_T^2 \sum_{i,j,l \neq j} \delta_{ij} \delta_{il} + \\
\quad p_S^2 p_T^2 \sum_{i,j,k \neq i} \delta_{ij} \delta_{kj} - p_S^2 p_T^2 \sum_{i,j,k \neq i,l \neq j} \delta_{ij} \delta_{kl} - \\
\quad p_S^2 p_T \left( \sum_{k \neq i,l \neq j} \delta_{kl} + \sum_{k \neq i} \delta_{kj} + \sum_{l \neq j} \delta_{il} + 1 \right) \]

\[ = p_S p_T \sum_{i,j} \delta_{ij} \left[ 1 + p_T \sum_{l \neq j} \delta_{il} + p_S \sum_{k \neq i} \delta_{kj} - \\
\quad p_S p_T \left( \sum_{k \neq i} \delta_{kj} + \sum_{l \neq j} \delta_{il} + 1 \right) \right] \]

\[ = p_S p_T \sum_{i,j} \delta_{ij} \left( 1 - p_S p_T + p_T (1 - p_S) \sum_{l \neq j} \delta_{il} + \\
\quad p_S (1 - p_T) \sum_{k \neq i} \delta_{kj} \right) \]

Without loss of generality, assume that $p_S \leq p_T$. We derive an upper bound of the variance by removing the negligible terms as follows:
\[
\text{Var}[\text{OUT}_R] \leq p_S p_T \sum_{i,j} \delta_{ij} \left( p_T \sum_{l \neq j} \delta_{il} + p_S \sum_{k \neq i} \delta_{kj} \right) \\
\leq p_S p_T \text{OUT} (p_T \text{OUT} + p_T \text{OUT}) \\
\leq 2p_S p_T^2 \text{OUT}^2
\]

which gives us the following ratio of the standard deviation and the expected value of \( \text{OUT}_R \):

\[
\frac{\sigma[\text{OUT}_R]}{\mathbb{E}[\text{OUT}_R]} \leq \sqrt{\frac{2p_S p_T^2 \text{OUT}^2}{p_S p_T \text{OUT}}} \\
\leq \sqrt{\frac{2}{p_S}}
\]

Chebyshef’s inequality gives us an upper bound on the output imbalance rate which is the ratio of max-reducer-output and the expected output size of a reducer.

\[
\Pr \left[ |\text{OUT}_R - \mathbb{E}[\text{OUT}_R]| \geq k\sqrt{\frac{2}{p_S} \mathbb{E}[\text{OUT}_R]} \right] \leq \frac{1}{k^2}
\]

For example, for \( p_S = 1/5 \), the probability of the output imbalance being more than 5.7 cannot be greater than 30%. Notice that this is an upper bound that is quite loose in practice. Our experiments show that join output is generally very evenly distributed over the reducers. This is to be expected as long as join output size is large enough so that sampling variance is “averaged out”. Significant variance in output size is only likely when join output size is very small, e.g., below thousands of tuples per reducer. However for those cases the total join output size is so small that even a significant output imbalance has only a small effect on the absolute runtime of phase 2. (Note that MapReduce typically works with data chunks of size 64 megabytes or larger.)

In short, whenever join output size is large enough to significantly affect job completion time, 1-Bucket-Random’s randomized approach balances output very well across reducers. It therefore is very difficult to beat it on output-related costs.
3.3. THE 1-BUCKET-RANDOM ALGORITHM

For another algorithm to achieve significantly lower total job completion time, it
has to have significantly lower input-related costs than 1-Bucket-Random.

**Lemma 3.3.4.** Let $1 \geq x > 0$. Any matrix-to-reducer mapping that has to cover at
least $x|S||T|$ of the $|S||T|$ cells of the join matrix, has a max-reducer-input value
of at least $2\sqrt{x|S||T|}/r$.

*Proof.* If $x|S||T|$ cells of the matrix have to be covered by $r$ regions (one for each
reducer), it follows from the pigeonhole principle that at least one reducer has to
cover $x|S||T|/r$ or more cells. This together with Lemma 3.2.1 implies that at least
$2\sqrt{x|S||T|}/r$ input tuples need to be sent to that reducer.

As we showed in Section 3.3.1.1, 1-Bucket-Random virtually guarantees that
its max-reducer-input value is at most $4\sqrt{|S||T|}/r$, and usually it is much closer
to the lower bound of $2\sqrt{|S||T|}/r$. Hence the ratio between max-reducer-input of
1-Bucket-Random versus any competing join algorithm using a different matrix-
to-reducer mapping is at most

$$\frac{4\sqrt{|S||T|}/r}{2\sqrt{x|S||T|}/r} = \frac{2}{\sqrt{x}}.$$ 

E.g., compared to any competing join implementation whose matrix-to-reducer
mapping has to cover 50% or more of the join matrix cells, 1-Bucket-Random’s
max-reducer-input is at most about 3 times the max-reducer-input of that algorithm.
Notice that this is an upper bound that is quite loose in practice. E.g., when work-
ing with 100 reducer nodes and inputs that are of similar sizes, e.g., where one is
at most 4 times larger than the other, max-reducer-input is closer to $2.5\sqrt{|S||T|}/r$
(instead of $4\sqrt{|S||T|}/r$). Then the worst-case ratio for any mapping covering at
least 50% of the join matrix is only $1.25/\sqrt{0.5} \approx 1.8$.

In summary, unless $x$ is very small, no other matrix-to-reducer mapping is go-
ing to result in significantly lower max-reducer-input compared to using 1-Bucket-
Random. Stated differently, the only way to significantly improve over the job
completion time of 1-Bucket-Random is to find a matrix-to-reducer mapping
that does not assign a significant percentage of the join matrix cells to any reducer, e.g., at least 50%.

Recall that for correctness, every join matrix cell with value \texttt{true} has to be assigned to a reducer (see Section 3.2). This means that for any join that produces a large fraction of the cross-product, 1-Bucket-Random is also guaranteed to be close to optimal in terms of both max-reducer-input and max-reducer-output.

For joins with very selective conditions, usually a matrix-to-reducer mapping will exist that has a significantly lower max-reducer-input than 1-Bucket-Random. To improve over 1-Bucket-Random, we have to find such a mapping. Because of Lemma 3.3.4, a necessary condition for this mapping is that it covers only a relatively small percentage of the join matrix cells, e.g., less than 50%. As the following discussion shows, it is often difficult in practice to find a mapping with this property due to requirements for both input statistics and join condition.

**Input statistics.** Knowing only the cardinality of \(S\) and \(T\), it is not possible to decide for any matrix cell if it is part of the join output or not. Let \((s, t)\) be a pair of an \(S\) and a \(T\) tuple that satisfies the join condition. If the join algorithm assumes that some matrix cell \(M(i, j)\) is not part of the join result, one can easily construct a counter example by creating sets \(S\) and \(T\) where \(s\) and \(t\) are assigned to the \(i\)-th row and \(j\)-th column of \(M\), respectively. To identify matrix cells that do not need to be covered, more detailed input statistics are required.

**Join condition.** If the join condition is a user-defined blackbox function, then we do not know which join matrix cells have value \texttt{false} unless we actually evaluate the join condition for these cells. However, this defeats the purpose of the algorithm: To find an efficient join algorithm, we would actually have to compute the join for all cells we are considering as candidates for not covering them by any reducer. Even if a join condition does not contain user-defined functions, in practice it is often difficult to identify large regions in the join matrix for which the algorithm can be certain that the entire region contains no join result tuple.

Before we explore this issue in more depth, consider the examples in Figure 3.5. Assume that in both cases we compute the same equi-join for the same inputs. The
3.3. THE 1-BUCKET-RANDOM ALGORITHM

partitioning in the left example is better, because it avoids the high input duplication needed for 1-Bucket-Random’s cross-product based partitioning in the right example. This is achieved by not covering large regions of cells that contain no result tuples (lower-left and upper-right corners).

For a set \( C \) of matrix cells of interest, input statistics give us predicates that hold for this set. Consider the left example in Figure 3.5. There the \( S \)-tuple with the \( i \)-th largest value of the join attribute is mapped to row \( i \) (similarly for \( T \)-tuples and matrix columns). For the block of 3 by 4 cells in the lower-left corner, histograms on \( S \) and \( T \) could imply that predicate \((S.A \geq 8 \land T.A \leq 7)\) holds for this matrix region. To be able to not assign any cell in this block to a reducer, the algorithm has to know that none of the cells in the region satisfies the join condition. In the example, it has to show that \( \forall s \in S, t \in T : (s.A \geq 8 \land t.A \leq 7) \Rightarrow \neg(s.A = t.A) \). While this is straightforward for an equi-join, it can be difficult and expensive in general. For each candidate region, the algorithm has to derive the corresponding predicates. It then has to run some satisfiability (SAT)-solver algorithm to prove that these predicates imply that the join condition is not satisfiable. If join result tuples are “scattered” all over the join matrix, then most large matrix regions will contain some output tuples. To identify regions without join tuples, therefore many small regions have to be examined, resulting in high computational cost.

To summarize our results: For selective join conditions, better algorithms than
Algorithm 3: Map (Sampling)

Require: input tuple \( x \in S, n, |S| \)

1: \( \text{rand} = \text{random}(0, 1) \) \( \{\text{rand} \in [0, 1)\} \)
2: \( \text{if } \text{rand} < n/|S| \) then
3: \( \text{Output}(0, (x.jav, x)) \) \( \{\text{Record } x \text{ with join attribute value } x.jav\} \)
4: \( \text{end if} \)

1-Bucket-Random might exist. These algorithms require that a significant fraction of the join matrix cells not be assigned to any reducer. Unfortunately, in practice it can be impossible (insufficient input statistics, user-defined join conditions, complex join conditions) or computationally very expensive to find enough of such matrix cells. For those cases, even if we could guess a better matrix-to-reducer mapping, we could not use it because there is no proof that it does not miss any output tuple. The next section will explore special cases where the join condition admits an efficient algorithm for identifying regions in the join matrix that do not contain any join result tuples.

3.4 Improving Highly Selective Theta-Joins by Exploiting Statistics

In this section, we present algorithms that improve over 1-Bucket-Random for popular join types.

3.4.1 Exploiting Approximate Equi-Depth Histograms

1-Bucket-Random only needed to know the cardinality of the inputs. To improve over it, we showed in the previous section that we need to identify large regions in the join matrix that do not contain any output tuples. This requires more detailed input statistics, which have to be computed on-the-fly if they are not available.

We can compute a one-dimensional approximate equi-depth histogram for both input sets \( S \) and \( T \) with two scans as follows. In the first pass, we sample ap-
3.4. IMPROVING HIGHLY SELECTIVE THETA-JOINS BY EXPLOITING STATISTICS

Algorithm 4: Reduce (Sampling)

Require: \((0, [(x_1, jav, x_1), (x_2, jav, x_2), \ldots, (x_k, jav, x_k)])\), \(n, k\)

1: \(width = n/k\)
2: \(counter = 1\)
3: for all \((x_i, jav, x_i)\) in input list do
4: \(\text{if } counter = width \text{ then}\)
5: \(\text{Output}(x_i)\)
6: \(\text{counter} = 0\)
7: \(\text{end if}\)
8: \(\text{counter} = \text{counter} + 1\)
9: end for

Figure 3.6: Example join matrix computed based on histograms

Approximately \(n\) records from each \(S\) and \(T\) in a MapReduce job. The map phase is executed as described in Algorithm 3. For an input tuple from \(S\), Map decides with probability \(n/|S|\) to output the tuple, otherwise discards it. These \(n\) records are sorted by join attribute and grouped in a single reduce task (Algorithm 4) to compute approximate \(k\)-quantiles, \(k < n\), that are used as the histogram’s bucket boundaries. \((T\) is processed analogously.) In a second MapReduce job, we make another pass on both data sets and count the number of records that fall into each histogram bucket. For join types where it is beneficial to have histogram bucket boundaries between \(S\) and \(T\) line up, we perform the second phases for \(S\) and \(T\) together and use the union of both sets’ bucket boundaries for the final histogram for each set.

Using such a histogram, it is straightforward to identify “empty” regions in the join matrix for the popular equi-joins, band-joins, and inequality-joins. As
discussed earlier, only non-empty matrix regions, i.e., those containing at least one result record, need to be assigned to a reducer. We refer to cells in those regions as **candidate cells**. Figure 3.6 illustrates a join matrix based on one-dimensional histograms of $S$ and $T$. Assume that the darker shaded cells represent the actual join output which is not known in advance. The lighter shaded cells represent the candidate cells computed based on the bucket boundaries and the join condition. Given the join condition $|S.a - T.b| < 20$, it is straightforward to prune the buckets by evaluating the condition on bucket boundaries. For example, the bucket bounded by $100 \leq S.a < 200$ and $T.b < 50$ cannot contain any join results since $50 < S.a - T.b < 200$ and $T.b - S.a < -50$.

For illustration purposes, in Section 3.2, each join matrix entry (cell) $M(i, j)$ represented the join result of $i$-th tuple from $S$ and $j$-th tuple from $T$. In reality, we only store each candidate region to model the join matrix. The number of candidate regions in a join matrix is bounded by $k^2$ where $k$ is the number of histogram buckets used. Therefore, the space complexity of the join matrix is $O(k^2)$. For the popular equi-joins, band-joins, and inequality-joins, the join matrix has the following monotonicity property: If cell $(i, j)$ is not a candidate cell (i.e., it is guaranteed to be false), then either all cells $(k, l)$ with $k \leq i \land l \geq j$, or all cells $(k, l)$ with $k \geq i \land l \leq j$ are also false and hence not candidate cells. (Which case holds for cell $(i, j)$ is easy to determine based on the join predicate.) By using this property, we can represent each join matrix column with two candidate regions only: the first and the last candidate regions in the column. All regions between those two are guaranteed to be candidate regions and the rest of the regions are pruned. Therefore, the space complexity of the join matrix for equi-joins, band-joins, and inequality-joins is $O(k)$.

### 3.4.2 M-Bucket-I and M-Bucket-O Algorithms

For input-size dominated joins, we want to find a matrix-to-reducer mapping that minimizes max-reducer-input. Finding such an optimal cover of all candidate re-
3.4. IMPROVING HIGHLY SELECTIVE THETA-JOINS BY EXPLOITING STATISTICS

Algorithm 5: M-Bucket-I

Require: maxInput, r, M

1: row = 0
2: while row < M.noOfRows do
3:   (row, r) = CoverSubMatrix(row, maxInput, r, M)
4:   if r < 0 then
5:     return false
6:   end if
7: end while
8: return true

Regions in general is a hard problem, hence we propose a fast heuristic. We refer to the corresponding algorithm as M-Bucket-I, because it needs more detailed input statistics (Multiple-bucket histogram) and minimizes max-reducer-Input.

The pseudo-code of M-Bucket-I is shown in Algorithm 5. Given the number of reducers (r), the maximal input size allowed for each reducer (maxInput), and a join matrix M, the algorithm divides the covering problem into sub-problems of covering sub-regions of the join matrix in each step. In order to preserve similar sub-problems and avoid producing join output duplicates at different reducers, M-Bucket-I only considers horizontal fragments of the matrix. Figure 3.7 illustrates how Algorithm 5 deals with horizontal fragments of the matrix. Starting from the top row of cells (in the first bucket of the S histogram) as illustrated in Figure 3.7a, it tries to cover all candidate cells in a block of consecutive rows. Note that the algorithm may find a covering that splits a candidate region to multiple reducers. Given the number of rows covered at each reducer and the size of the candidate region, M-Bucket computes the ratio of records that need to be assigned to each reducer. Once a good covering is found for a block, the covering is fixed (Figure 3.7b). The same process repeats starting at the next row that is not covered yet (Figure 3.7c). It continues to cover blocks of consecutive rows until it has either covered all candidate cells, or it exhausted the r reducers without being able to cover all candidate cells (Figure 3.7d). The details of how to pick horizontal fragments and number of reducers to be used at each step are presented next.

During each execution of the while-loop, Algorithm 5 covers a block of rows
CHAPTER 3. PROCESSING THETA-JOINS USING MAPREDUCE

Figure 3.7: Illustration of the M-Bucket-I Algorithm

using Algorithm 6. Based on the shape of the candidate cells, the size of the horizontal fragment having a good covering may change at each iteration. For instance, if the candidate cells form a tall slim shape, a taller horizontal fragment would give better covering. Algorithm 6 searches for horizontal fragments that result in good covering at each iteration. Let’s assume for now that there is an algorithm that covers (or fails to cover) a horizontal fragment of $M$ between $row_s$ and $row_e$ using $r$ reducers. Using this algorithm, given starting row $row_s$, for all number of reducers $r_i$ where $1 \leq r_i \leq r$, it is possible to perform a binary search on $row_e$ and test if $r_i$ reducers can cover horizontal fragments bounded by $row_s$ and $row_e$. The highest scoring covering among different blocks (from $row_s$ to $row_i$ where $row_i \leq row_e$) is fixed and the next uncovered row is returned to Algorithm 5. The score is defined as the average area of candidate cells covered by each reducer in the block. (Intuitively, we want to cover as many candidate cells with as few reducers as possible.)

Note that the size of a horizontal fragment that can be covered in a single step is not limited in the binary search. The remainder of this section presents the details of the algorithm used by Algorithm 6 which finds a covering on a horizontal fragment of a given size using specified number of reducers.

Given the first row $row_f$ and the last row $row_l$ of a particular horizontal fragment of cells in $M$, and the number of reducers $r_i$, M-Bucket-I assigns candidate cells within the fragment column-by-column as shown in Algorithm 7. It starts by creating a new reducer with an initial input capacity of $maxInput$. M-Bucket-I iterates through each column $c_i$ and assigns all candidate cells within $c_i$ (between
3.4. IMPROVING HIGHLY SELECTIVE THETA-JOINS BY EXPLOITING STATISTICS

Algorithm 6: CoverSubMatrix (M-Bucket-I)

Require: row_s, maxInput, r, M
1: maxScore = -1, rUsed = 0
2: minRow = row_s, maxRow = M.noOfRows
3: for i = 1 to r do
4:   if maxScore > 0 then
5:     minRow = bestRow + 1
6:   end if
7:   while minRow ≤ maxRow do
8:     row_e = (minRow + maxRow)/2
9:     R_i = CoverRows(row_s, row_e, maxInput, i, M)
10:    if R_i = ∅ then
11:       {CoverRows unsuccessful}
12:       maxRow = row_e - 1
13:    else
14:       area = totalCandidateArea(row_s, row_e, M)
15:       score = area / i
16:       if score ≥ maxScore then
17:         bestRow = row_e
18:         rUsed = i
19:         minRow = row_e + 1
20:       else
21:         maxRow = row_e - 1
22:       end if
23:     end if
24:   end while
25: end for
26: r = r - rUsed
27: return (bestRow+1, r)

row_f and row_l) to the reducer as long as the input capacity is not exceeded. In order

row to search candidate regions efficiently, each column is designed as a linked-list of
candidate regions. The next candidate region to be processed in the current column

row can be retrieved in constant time. If adding all candidate cells in a column c_i would

row result in the reducer exceeding its input limit, a new reducer is created and c_i is

row added to the new reducer if possible. The algorithm continues until it covers all

row columns or fails to cover. If all columns of the block are covered without exceed-

row r_i, Algorithm 7 returns the set of cover reducers created (empty set is returned

row otherwise). Figure 3.8 illustrates how Algorithm 7 works. Assume that the hor-
Algorithm 7: CoverRows (M-Bucket-I)

Require: row_f, row_l, maxInput, r, M

1: Reducers = ∅; nr=newReducer(); r = r - 1;
2: for all c_i in M.getColumns do
3:     if r < 1 then
4:         return ∅;
5:     end if
6:     if nr.cap < c_i.candidateInputCosts then
7:         Reducers = Reducers ∪ nr
8:         nr=newReducer()
9:         r = r - 1
10:    end if
11:    nr.Cells = nr.Cells ∪ c_i.candidateCells
12: end for
13: return Reducers

Horizontal fragment between row_f and row_l needs to be covered using two reducers with a maximum capacity of five each. M-Bucket-I assigns candidate cells within the first column c_f to the first reducer r_1 in Figure 3.8a. r_1 has remaining input capacity of two since two rows and one column are assigned so far. Since there are no new rows that need to be covered in the second column, the input capacity is reduced by one. r_1 reaches its maximum input capacity of 5 after the third column is assigned (Figure 3.8b). Therefore, the algorithm continues on the next column with a new reducer, r_2. As shown in Figure 3.8c, the fourth column contains three candidate cells between row_f and row_l. Therefore, the input capacity of r_2 is reduced by four. Covering the last column by r_2 requires only input capacity of one since all the rows between row_f and row_l on the last column are already covered by r_2. Therefore, M-Bucket-I successfully covers the horizontal fragment between row_f and row_l using two reducers having maximum capacity of five.

We use M-Bucket-I in a binary search to find the smallest value maxInput for which M-Bucket-I can find a cover that uses at most r reducer tasks. The upper and lower bound of maxInput for the binary search are |S| + |T| and 2√number-of-candidate-cells/r, respectively. The former is obvious, because we can cover the entire matrix with a rectangle of |S| rows by |T| columns. The latter follows from Lemma 3.2.1 and the fact that max-reducer-output is at least
3.4. IMPROVING HIGHLY SELECTIVE THETA-JOINS BY EXPLOITING
STATISTICS

Figure 3.8: Illustration of Algorithm 7 on a horizontal fragment bounded by \( r_{ow_f} \)
and \( r_{ow_l} \) using 2 reducers having maxInput = 5

number-of-candidate-cells/r.

Recall that M-Bucket-I was designed to minimize max-reducer-input. For
output-size dominated joins, one should minimize max-reducer-output instead. For
this problem, we developed a heuristic called M-Bucket-O. It proceeds like M-
Bucket-I, but instead of working with an input-size limit \( maxInput \), it limits
reducers by area, i.e., number of cells contained in a reducer.

Notice that M-Bucket-I can take better advantage of input histograms than M-
Bucket-O, because it knows exactly how many input tuples from each data set be-
long to each bucket. On the other hand, the actual output size of a bucket could be
anything between zero and the area of the bucket (bucket count \( S \) times the bucket
count \( T \)). Hence M-Bucket-I can reliably balance input-related costs even with
fairly coarse-grained histograms, while M-Bucket-O can show significant output-
cost imbalance even for very fine-grained histograms (e.g., where each bucket con-
tains an average of five distinct attribute values). Our experiments support this
observation.

M-Bucket-I algorithm described in this section can be used for any theta-join.
Recall from Section 3.4.1 that for various types of joins such as equi-joins, band-
joins and inequality joins the join matrix has monotonicity property. M-Bucket-I
can therefore process join matrix faster by pruning the search space based on this
monotonicity property.
CHAPTER 3. PROCESSING THETA-JOINS USING MAPREDUCE

Algorithm 8: Map (Theta-Join) using M-Bucket

Require: input tuple \( x \in S \cup T \)

Ensure: \{Matrix to reducerID mapping produced by M-Bucket is loaded into a lookup table during initialization of mapper\}

1: \( \text{rndVal} = \text{random}(0,1) \)
2: \text{if} \( x \in S \) \text{then}
3: \hspace{1em} \text{for all} \ \text{reducerID in lookup.getReducers("S", \text{rndVal}) do}
4: \hspace{2em} \text{Output} (\text{reducerID}, (x, "S"))
5: \hspace{1em} \text{end for}
6: \text{else}
7: \hspace{1em} \text{for all} \ \text{reducerID in lookup.getReducers("T", \text{rndVal}) do}
8: \hspace{2em} \text{Output} (\text{reducerID}, (x, "T"))
9: \hspace{1em} \text{end for}
10: \text{end if}

3.4.3 Using M-Bucket in MapReduce

Using matrix-to-reducer mappings produced by M-Bucket algorithms in MapReduce is similar to how 1-Bucket-Random algorithm is used as described in Section 3.3.1.3.

Each incoming tuple has to be assigned to the correct histogram bucket first. Then, we can randomly select a row/column inside the bucket boundaries if M-Bucket algorithms created multiple rows/columns inside the boundary. Map draws a random number between 0 and 1, and assigns the tuple to the corresponding row in matrix \( M \) that is identified by the ratio of the row size to the corresponding histogram bucket size. \( T \)-tuples are assigned to columns in \( M \) similarly. The pseudo-code of Map phase is given in Algorithm 8. Note that the matrix-to-reducer mapping is computed before this MapReduce job starts and all nodes are initialized with the mapping in the setup function of the mapper class. Since M-Bucket is deterministic, it can be also applied in the setup function in parallel before Map starts.

3.4.4 The Overall Algorithm

Given two input sets \( S \) and \( T \), and a join condition \( \theta \), we can often choose between different MapReduce implementations. For equi-joins, there are the repartition join
(Section 2.2), the M-Bucket algorithms, and 1-Bucket-Random. For any other arbitrary joins, there are the latter two, M-Bucket algorithms, and 1-Bucket-Random.

Depending on the join condition, we consider all applicable algorithms. From their corresponding matrix-to-reducer mapping, we can estimate max-reducer-input and max-reducer-output for each algorithm. Then we can apply traditional cost estimation techniques from databases, because the job completion time is determined by the reducer (i.e., single processor) that receives the greatest input and the reducer that generates the greatest output. Local reducer computation is directly amenable to traditional cost analysis involving CPU and I/O cost. For DFS data transfer we can approximate cost through a disk-like model of average latency and transfer time.

### 3.5 Experimental Evaluation

We discuss representative results for our algorithms for joining real and synthetic data. All experiments were performed on a 12-machine cluster running Hadoop 0.20.203 [1]. One machine served as the head node, while the other 11 were the worker nodes. Each machine has a single quad-core Xeon 2.4GHz processor, 8MB cache, 8GB RAM, and two 250 GB 7.2K RPM hard disks. All machines are directly connected to the same Gigabit network switch. In total, the cluster therefore has 44 cores with 2GB memory per core available for Map and Reduce tasks. Each core is configured to run one map and one reduce task concurrently. The distributed file system block size is set to 64MB and all machines participate as storage nodes for the DFS.

We present results for the following data sets:

**Cloud**: This is a real data set containing extended cloud reports from ships and land stations [38]. There are 382 million records, each with 28 attributes, resulting in a total data size of 28.8GB.

**Cloud-5-1, Cloud-5-2**: These are two independent random samples of 5 million records each from Cloud. They are used for experiments with output-size
dominated joins.

**Synth-α**: For a fixed $\alpha$, this is a pair of data sets (one for each join input). Both contain 5 million records, each record being a single integer number between 1 and 1000. For one data set, numbers are drawn uniformly at random from this range. For the other data set, we use the Zipf distribution for the same range. Skew is adjusted by choosing a value between 0 (uniform) and 1.0 (skew) for $\alpha$, which is the usual Zipf skew parameter.

### 3.5.1 Equi-Joins: 1-Bucket-Random vs. Repartition join

Figure 3.9 reports output imbalance for computing an equi-join on Synth-α for various values of $\alpha$. Since all experiments have significantly larger output than input, we report output imbalance. Imbalance is computed as max-reducer-output divided by average-reducer-output. We compare the load imbalance of 1-Bucket-Random against the repartition join implementation from the literature.

As we discussed in Sections 3.3.1.1 and 3.3.2, 1-Bucket-Random is virtually guaranteed to achieve an excellent balancing of both input and output tuples because of its randomized approach. As predicted, the repartition join algorithm’s approach of creating reduce jobs based on join attribute values suffers from increasing skew. There is no significant difference between the two algorithms when both inputs are uniform (Synth-0). However, the node responsible for the most frequent value is overloaded by a factor of 3.5 to 6.3 for skewed data. This could
be much worse for a join attribute with smaller cardinality even for small data set size. Since total output size is approximately the same for each experiment, output imbalance directly reflects max-reducer-output. And as predicted by our qualitative cost analysis, greater max-reducer-output leads to greater job completion time.

### 3.5.2 Input-Size Dominated Joins

We study M-Bucket-I for the following selective self-join on the large Cloud data set:

```sql
SELECT S.date, S.longitude, S.latitude, T.latitude
FROM Cloud AS S, Cloud AS T
WHERE S.date = T.date AND S.longitude = T.longitude
    AND ABS(S.latitude - T.latitude) <= 10
```

This join produces 390 million output tuples, a much smaller set than the total of almost 800 million input records.

The statistics for M-Bucket-I are approximate equi-depth one-dimensional histograms of different granularities (i.e., number of buckets) for Cloud. The approximate $k$-quantiles are computed using a 1% random sample of the input. The 1-dimensional sort key for the histogram is the combined (date, longitude, latitude)-vector, using alphabetical sorting with date as the most significant component. Notice that for a 1-bucket histogram, M-Bucket-I degenerates to a version of 1-Bucket-Random. Instead of using the square-only approach of 1-Bucket-Random (which was needed to achieve the bounds in Theorem 3.3.2, but often does not use all reducers), we run the heuristic described in Section 3.4.2 to find the matrix-to-reducer mapping.

For histogram granularities 1 and 10, max-reducer-input size turned out to exceed main memory size of the reducer nodes. Therefore, we used our memory-aware version of M-Bucket-I for experiments with buckets 1 and 10. Note that this version will create $c \cdot r$ reduce tasks for $r$ reducers and some appropriately chosen integer $c \geq 1$, just large enough to make each reducer input fit in memory.
Figure 3.11a shows how input imbalance, computed as (max-reducer-input / avg-reducer-input), of M-Bucket-I changes with varying histogram granularity. (Notice that all 44 reducers were used in all cases, hence avg-reducer-input is comparable across algorithms.) The graph shows that our heuristic achieves its goal of balancing input evenly across reducers.

Figure 3.11b compares max-reducer-input for M-Bucket-I as we vary the granularity of the input statistics. Here the data point for 1 bucket again corresponds to the version of 1-Bucket-Random described above. It is obvious that even though 1-Bucket-Random almost perfectly balances input load, it has a lot more to balance than M-Bucket-I with more fine-grained histograms. The reason is that 1-Bucket-Random covers the entire join matrix, while M-Bucket-I can avoid covering a large fraction of the matrix based on the available statistics and properties of the band-join. As predicted by our cost analysis, Figure 3.11c shows for this input-size dominated join that MapReduce job completion time tracks the trend of
max-reducer-input almost perfectly.

Figure 3.11c does not include M-Bucket-I’s pre-processing costs. The detailed cost breakdown and true total job completion times are shown in Table 3.1. In addition to performing the MapReduce job for the join, M-Bucket-I first has to compute the statistics (unless they are already available) by finding the approximate quantiles (single pass to compute random sample on each data set) and then counting the number of records per quantile range (another sequential pass on each data set). The heuristic for finding the best matrix-to-reducer mapping is run on a single node, followed by the actual join. As we can see, at some point more fine-grained statistics improve join cost only minimally, while especially the preprocessing step to compute input statistics and heuristic for finding the matrix-to-reducer mapping becomes more expensive as more (and smaller) candidate regions have to be considered by M-Bucket-I. Note that the heuristic is more expensive using 1 bucket compared to using 10 buckets because it satisfies the memory limit constraint with a greater number of partitions. The memory limit constraint is satisfied with 2068 and 176 partitions respectively and this increase in the number of partitions results in slower partitioning.

In summary, this experiment highlights the Achilles heel of 1-Bucket-Random—its potentially high input duplication. Input duplication was worse than the number of reducers (set to 44) suggests due to the extra partitioning performed in order to satisfy memory constraints of reduce tasks. In chapter 4, we propose novel optimization techniques to reduce the amount of data transferred between mappers and reducers which is applicable to our memory-aware join algorithm. For smaller input data or when running the version with local I/O (and hence fewer
partitions), input duplication would be significantly lower. The experiments also show that whenever available statistics and properties of the join condition enable us to avoid mapping a large fraction of the join matrix to any reducer, input-related costs can be reduced significantly.

### 3.5.3 Output-Size Dominated Joins

We study the following moderately selective join on the smaller Cloud-5 real data sets:

```sql
SELECT S.latitude, T.latitude
FROM Cloud-5-1 AS S, Cloud-5-2 AS T
WHERE ABS(S.latitude - T.latitude) <= 2
```

This join produces 22 billion output tuples, a much larger set than the total of 10 million input records.

This experiment mirrors the one reported in the previous section, but now for a join that is output-size dominated. As with M-Bucket-I, M-Bucket-O for a histogram with 1 only bucket degenerates to a version of 1-Bucket-Random. It tries to find a partitioning of the join matrix where all reducers have approximately the same area (number of cells covered), but no reducer is left without work (as could happen for the original 1-Bucket-Random algorithm, whose primary goal was to guarantee the result in Theorem 3.3.2).

Figure 3.12a shows the output-size imbalance, computed as (max-reducer-output / avg-reducer-output), for different granularities of the histogram. (Notice that all 44 reducers were used in all cases, hence avg-reducer-output is comparable across algorithms.) 1-Bucket-Random, as expected, achieves almost perfect output balancing due to its randomized approach. However, for other histogram granularities, output imbalance is much greater than the corresponding numbers for the input imbalance for M-Bucket-I for the previous experiment. Even though input sizes can be estimated well, even from comparably coarse-grained histograms (see Figure 3.11a), the number of output tuples in a reducer is difficult to estimate be-
3.5. EXPERIMENTAL EVALUATION

Figure 3.12: Experimental Results for Output Size Dominated Joins

Table 3.2: M-Bucket-O cost details (seconds)

<table>
<thead>
<tr>
<th>Step</th>
<th>Number of Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Quantiles</td>
<td>0</td>
</tr>
<tr>
<td>Histogram</td>
<td>0</td>
</tr>
<tr>
<td>Heuristic</td>
<td>0.06</td>
</tr>
<tr>
<td>Join</td>
<td>802.8</td>
</tr>
<tr>
<td>Total</td>
<td>802.86</td>
</tr>
</tbody>
</table>

Because join selectivity can vary significantly in different reducers, 1-Bucket-Random is practically immune to this problem, because its randomization “shuffles” tuples around randomly, hence tuples from reducers with high result density and those from reducers with low result density in the join matrix get intermixed. M-Bucket-O cannot do this, because a tuple can only be assigned to the appropriate histogram bucket, not any random bucket.

Since the total join output size is independent of the number of histogram buckets, Figure 3.12b looks exactly the same as Figure 3.12a, just with all numbers scaled by average-reducer-output. And since the join is output-size dominated, the
job completion time numbers in Figure 3.12c for the MapReduce join implementation closely track the values for max-reducer-output. This highlights that minimizing max-reducer-output for this problem is the right approach. The true total job completion times are listed in detail in Table 3.2.

In summary, for the output-size dominated join problem, 1-Bucket-Random performed better than M-Bucket-O, except when very detailed statistics were available for the latter. Notice that there are 5951 different latitude values in the data set, hence the histogram with 5951 had a single bucket per occurring latitude value. This allowed M-Bucket-O to compute exact output sizes for any region in the join matrix. With fewer buckets, estimation error increased significantly, resulting in worse output balancing.

### 3.6 Conclusions

In this chapter, we proposed algorithms for implementing any arbitrary theta-join as a single MapReduce job. This implementation is achieved without any modifications to the MapReduce framework.

Starting with the goal of minimizing total job completion time, we showed how to define a great variety of join implementations using appropriate join matrix-to-reducer mappings. We proposed 1-Bucket-Random, a randomized algorithm whose matrix-to-reducer mapping we showed to be provably close to optimal for any join with significantly larger output size than its input size. For more selective join conditions we showed that even though there might be faster algorithms than 1-Bucket-Random, in practice it might often not be possible to identify these algorithms as usable without performing expensive analysis or without knowing the join result in advance.

We proposed the M-Bucket-I algorithm that can improve runtime of a popular class of non-equi joins compared to 1-Bucket-Random by exploiting input statistics to exclude large regions of the join matrix with a comparably lightweight test, and thus reduce input-related costs.
CHAPTER 4

Optimizing MapReduce Programs using Anti-Combining

4.1 Introduction

The join algorithms we propose in Chapter 3 are compatible with any local join implementation provided by the user. For in-memory join algorithms, we further partition the join matrix in order to generate larger number of reduce tasks with smaller max-reducer-input size in order to satisfy memory constraints of nodes in the cluster. The disadvantage of this approach is the extra data replication which results in larger amount of data transferred between mapper and reducer nodes.

Often the shuffle-and-sort phase, when data is transferred from mappers to reducers, represents the bottleneck of a MapReduce job execution. There are several reasons for this. (1) During the shuffle-and-sort phase, often massive amounts of data are grouped, sorted, and moved across the network [34, 80, 87]. (2) This data transfer is inherent to enable parallel execution. (3) And while shared-nothing environments (on which MapReduce tends to be executed) make it easy to increase CPU, memory, and disk resources by adding more machines, this is difficult for the network. Network links and switches are in fact shared resources in the sense that the same link or switch is on the path between many pairs of machines. Hence reducing network load is essential for increasing throughput in highly utilized environments.
Network load can be reduced through the use of a Combiner as explained in Section 2.1. A Combiner attempts to decrease mapper-to-reducer data transfer by applying some of the reducers’ work on the mappers, replacing individual records with more compact aggregate data. Unfortunately, combining essentially is limited to applications that compute distributive or algebraic [36] aggregates. And even if combining is possible, it will only be effective if many Map output records in the same map task have the same key. The same applies to the in-mapper combining design pattern [56].

Whenever combining is not possible or not effective, the programmer is left with the option to choose one of the compression techniques that usually come with MapReduce implementations. While they differ in computational cost and compression rate, they all follow the same pattern: Mapper output is compressed on the mapper machine. Then the compressed data chunks are sent to the appropriate reducers, where they are decompressed before merging and processing. While conceptually simple and elegant, this use of general-purpose compression can add a significant overhead for compression and decompression of large data sets.

Our approach is based on the following observation that trivially holds for any MapReduce program: Consider an input record \( i \). For this input record the Map
function might produce zero or more output records \( o_1, o_2, \ldots \). Some of these output records will be assigned to the same reduce task, others will end up in different reduce tasks. Figure 4.1 shows an example. Whenever multiple or large Map output records are assigned to the same task, e.g., \( o_1 \) and \( o_2 \) to Reducer 1, there is an opportunity for data reduction. Starting with this simple observation, we chose the following two design goals:

**Simple encoding/decoding functions.** We want to keep the overhead for data reduction low by only using encode and decode functions (see Figure 4.1) with low CPU cost. Furthermore, to avoid the need for large buffer space, each encode (and corresponding decode) function call will only be applied to some subset of the output records of a single Map call.

**Fine-grained adaptive optimization.** To achieve good compression, the choice of encode (and corresponding decode) should be driven by the data. In the example in Figure 4.1, we might choose a different encoding for \( \{o_1, o_2\} \) than for \( o_3 \). In particular, for \( o_3 \) it might be best to simply leave the record alone and transmit it unaltered. Hence the encoding decision has to be made adaptively at runtime. Similarly, the decision might be different for the output of different Map calls.

To meet these design goals, we propose **Anti-Combining**. As will become clear later, the encode/decode functions we propose for Anti-Combining reduce the total data transfer from mappers to reducers by shifting some of the mapper-side processing to the reducers. In that sense it does the opposite of a Combiner, which performs reducer-side work on the mappers.

Anti-Combining can be enabled in any given “plain” MapReduce program through purely syntactic program transformations. This makes it possible to enable it automatically even in programs written in expressive, usually Turing-complete, languages such as Java or C++. Hence it can be also applied to compiler-generated MapReduce code produced by systems such as Pig and Hive, or to other statically optimized MapReduce programs, e.g., those produced by database-style scan-sharing and multi-query optimization [76, 8, 34, 54, 60].
Like Combiners, Anti-Combining will not always result in significant cost savings. Fortunately, in our experience there is a large and diverse spectrum of applications that can significantly benefit from it. **Join processing** for instance, relies on input replication in the map phase in order to compute multi-way joins [7], complex join predicates [61] as described in Chapter 3, similarity joins [6, 77], and k-nearest neighbor joins [58, 86]. For many **graph algorithms** including PageRank [65], Hyperlink-Induced Topic Search [48], and social network analysis, the Map function processes a node by emitting output records for each outgoing edge in the node’s adjacency list. As graphs tend to be very skewed, Anti-Combining’s adaptive approach can significantly reduce cost for nodes with high out-degree, while leaving those with low out-degree alone. Furthermore, all previously proposed **multi-query optimization** techniques such as scan-sharing [8, 34, 60, 79] are a perfect target for Anti-Combining because a single record produced by the shared operator might have to be duplicated many times in order to forward it to the downstream operators of the queries involved.

In this chapter, we first discuss properties of MapReduce programs that enable optimization opportunities for reducing the amount of data transferred in the shuffle-and-sort phase and provide motivating examples including theta-joins in Section 4.2. Then, we propose Anti-Combining based on encoding and decoding techniques that exploit sharing opportunities in Section 4.3. We demonstrate the effectiveness of Anti-Combining on theta-joins and query log processing using real and synthetic data in Section 4.4.

### 4.2 Motivating Examples and Optimization Opportunities for Anti-Combining

#### 4.2.1 Query Suggestion using Log Processing

Consider a typical commercial search engine, which returns the best matching Web pages for a given search query. To aid the user in composing a query, most search
4.2. MOTIVATING EXAMPLES AND OPTIMIZATION OPPORTUNITIES FOR ANTI-COMBINING engines propose possible query completions as the user is typing. For example, after entering “icd” the search engine might suggest “icde”, “icde 2014”, and “icde acceptance rate”. For realtime suggestions, these expansions of a given prefix have to be pre-computed. While the algorithm for selecting suggested expansions is more complex (and usually a trade secret), one of its crucial inputs is the popularity of queries starting with the prefix typed by the user so far. For illustration purposes, we will therefore consider the following version of the Query-Suggestion problem: 

We are given a log of search queries. For any string \( P \) that occurred as a prefix of some query in the log, pre-compute the five most frequent queries in the log starting with prefix \( P \).

This is a comparably simple query involving grouping, aggregation, and top-k selection. Hence Query-Suggestion is perfectly suitable for parallel computation using MapReduce. The natural way of implementing Query-Suggestion in MapReduce is as follows. For a query string \( S \), the Map function emits intermediate key/value pairs \((P, S)\) for each prefix \( P \) of \( S \). By using the prefix as the key, the MapReduce environment guarantees that the Reduce call for some prefix \( P \) will have all queries with prefix \( P \) in its input list. It can then easily determine the most frequent queries for \( P \). (We use the secondary sort pattern [80] so that Reduce’s list of input values is sorted by query, greatly simplifying the Reduce function.)

In Figure 4.2, we revisit the data-flow in MapReduce using Query-Suggestion problem. It illustrates how a single query string will result in multiple Map output records, so that this query will be taken into account for each of its prefixes. The Map output is collected in a buffer which is spilled to disk when it fills up. Before writing them to disk, the intermediate key/value pairs are assigned to partitions corresponding to different reduce tasks by the Partition function. Records in each partition are sorted by key. Before the map task is finalized, the spill files are merged on disk, preserving the sort order for each partition. Then, each partition is transferred, usually over the network, to the machine responsible for the corresponding reduce task. In the example, keys “m”, “man” and “mango” are assigned to reduce task 1. This task processes the records in increasing key order, calling the
reduce function once for each key and determining the top-5 queries for it. Notice that in practice each query comes with additional features, e.g., on which search result the user clicked. These features can be included in the value component of a record, but are omitted here for simplicity.

For a search query of length $n$, the Map function will generate $n$ output records. Since each output record contains the query itself, each Map function call’s output is quadratic in its input size. This results in high cost of the shuffle-and-sort phase, including for transferring the data over the network, which is a shared resource and hence tends to become the bottleneck. When that happens, the MapReduce programmer has two options for decreasing the amount of data transferred from mappers to reducers: Combiners and compression.

First, if she is lucky and the goal is to compute a distributive or algebraic aggregate [36], she can add a Combiner. A Combiner applies aggregate computations that would normally take place in the Reduce function, already in the map task. For the Query-Suggestion problem, a Combiner could replace $m$ occurrences of the same pair (key, value) in the output of a map task by “aggregate” record (key, (value, $m$)). Unfortunately, our experiments show that the Combiner approach is not very effective for Query-Suggestion due to the large number of distinct query strings in each map task input batch (see Section 4.4.3). For other problems, e.g., holistic aggregates or non-aggregation problems, the Combiner approach is not applicable at all.

The second option for reducing the amount of data transferred from mappers to reducers is to use compression. Map output compression is easy to enable in
4.2. **MOTIVATING EXAMPLES AND OPTIMIZATION OPPORTUNITIES FOR ANTI-COMBINING**

Hadoop, but can add a significant computational overhead. As we will see soon, our approach is usually much more effective because it exploits MapReduce properties and is also computationally light-weight. Furthermore, our techniques can be used in combination with compression to greater effect.

### 4.2.2 Processing In-Memory Theta-Joins

As seen in Chapter 3, processing joins in MapReduce requires input replication. We provided near-optimal join processing algorithms that distribute multiple replicas of input tuples across \( r \) reduce tasks. As observed in Section 3.5.2, these algorithms may create partitions where the size of a reducer input is too big to fit in memory. In this case, the job would fail if the reduce function provided assumes that the reduce input fits in memory. Our memory-aware algorithm avoids this situation by partitioning the join matrix into \( p \) partitions where \( p > r \), which increases the map input replication rate.

For instance, Figure 4.3a illustrates a join matrix to be partitioned by our M-Bucket algorithms. Let \( S_i \) and \( T_j \) input chunks be equal sized and sum of each pair of chunks equals to the memory limit \( MEM \). Figure 4.3b illustrates an input-balanced solution of our memory-aware join algorithm on this join matrix. Each reduce group is labelled with an integer (numbered 1 to 16). Note that each group satisfies the given memory limit constraint, i.e., can be executed in memory since total input size of each group does not exceed \( MEM \). Assume that the number of available nodes in the cluster is 4. Therefore, these groups are going to be assigned to 4 reduce tasks. The labels \( P1, P2, P3, \) and \( P4 \) identify which partitions, i.e. reduce tasks these 16 groups are assigned to. For example, groups identified by 1, 2, 3, and 4 are assigned to partition \( P1 \). Note how this distribution for the memory-aware join algorithm sends the same input chunks to the same partitions for different reduce groups. For instance, input chunk \( S_1 \) is replicated three times since it is used in groups 1, 2, and 3. All these reduce groups are computed by the same task that this partition \( P1 \) is assigned to. Therefore, a single copy of \( S_1 \) would
be sufficient for $P_1$, where three copies are assigned in this specific example.

Note that Combiner is not applicable for processing joins.

### 4.2.3 Optimization Opportunities

We propose strategies that exploit data sharing opportunities based on the output produced by a single Map function call. Let $(k_1, v_1)$ and $(k_2, v_2)$ be two intermediate records emitted by Map for some input record $(k, v)$. As Table 4.1 shows, there are four different cases with respect to data sharing opportunities that can be exploited for reducing the cost of the shuffle-and-sort phase.

**Same key, same value.** If $k_1 = k_2 = k'$ and $v_1 = v_2 = v'$, then the two records can be encoded more compactly as $(k', (v', 2))$, where $(v', 2)$ indicates that there are two occurrences of value $v'$. (It is easy to see how this generalizes to a larger number of duplicate records.)

**Same key, different value.** If $k_1 = k_2 = k'$ and $v_1 \neq v_2$, then the two records can be encoded more compactly as $(k', (v_1, v_2))$. In general, a set of intermediate records with the same key will be encoded as a single record with that key and the corresponding list of values.

In both same-key cases, the Reduce call for key $k'$ would receive a list of encoded value lists. Decoding this list of lists is straightforward. Also note that Combiner can also deal with the same-key cases. On the other hand, data reduction through sharing is more challenging for the different-key cases. This is due to the
4.3 ANTI-COMBINING STRATEGIES

Table 4.1: Encoding Strategies for each Sharing Opportunity

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>different</td>
<td>different</td>
</tr>
</tbody>
</table>

fact that intermediate key/value pairs with different keys are processed by different Reduce calls. Hence even if two Map output records \((k_1, v')\) and \((k_2, v')\) share the same value \(v'\), the reducer will need these records separately, one for the input list for the Reduce call for \(k_1\) and the other for the input list for the Reduce call for \(k_2\). While this makes data sharing seemingly impossible for the different-key cases, we briefly summarize optimization opportunities here and then discuss the corresponding strategies in more detail in the following sections.

**Different key, same value.** If \(k_1 \neq k_2\) and \(v_1 = v_2 = v'\), then the two records can be encoded more compactly as \((k_1, \{k_2\}, v')\). This can only be done if keys \(k_1\) and \(k_2\) are assigned by the Partitioner to the same reduce task. And we have to make sure that this record is properly decoded in the reduce task so that the Reduce calls for \(k_1\) and \(k_2\) both see value \(v'\) in their input value-list. This is done by our strategy *EagerSH*.

**Different key, different value.** If \(k_1 \neq k_2\) and \(v_1 \neq v_2\), then there is no obvious data sharing opportunity. However, data reduction can still be achieved by sending the Map input record \((k, v)\) instead of the two output records. This is the basic idea behind our *LazySH* strategy. Making this work in practice requires execution of Map on a reducer and careful management of the output of the reduce-side Map calls.

4.3 Anti-Combining Strategies

In this section, we propose two encoding and decoding based techniques that enable Anti-Combining. In Section 4.3.1, *EagerSH* is presented which is for Map output records with the same value component. Next, *LazySH* is proposed in Section 4.3.2
which can be applied even Map output records have different keys and values, i.e.,
even when traditional compression would not be effective. Finally, we discuss how
these two techniques can be applied together with fine-grained decisions adaptively
at runtime in Section 4.3.4.

4.3.1 Eager Sharing Strategy

Let \((k_1, v')\) and \((k_2, v')\), \(k_1 \neq k_2\), be two intermediate records emitted by Map for
some input record \((k, v)\). Since both output records have the same value field, they
present a data reduction opportunity if both records can “share” the common value.
For instance, the two records can be encoded more compactly as \((k_1, \{k_2\}, v')\).
Unfortunately, sharing across records with different keys is challenging. This is due
to the fact that intermediate key/value pairs with different keys are processed by
different Reduce calls. Hence even if two Map output records \((k_1, v')\) and \((k_2, v')\)
share the same value \(v'\), the reducer will need these records separately: one for the
input list for the Reduce call for \(k_1\) and the other for the input list for the Reduce
call for \(k_2\).

We can still make this type of sharing work, as long as both keys are assigned
to the same reduce task. (This key-to-task assignment happens in the Partitioner.)
Furthermore, even if \(k_1\) and \(k_2\) are assigned to the same reduce task, we still have to
make sure that this record is properly decoded in the reduce task so that the Reduce
calls for \(k_1\) and \(k_2\) both see value \(v'\) in their input value-list. This is done by our
strategy EagerSH. Consider the following example:

\[
\begin{align*}
(k_1, v_1)_{r_1} & \quad (k_1, \{\}, v_1)_{r_1} \\
(k_2, v_1)_{r_2} & \quad (k_2, \{\}, v_1)_{r_2} \\
(k_{in}, v_{in}) & \xrightarrow{map} (k_3, v_2)_{r_2} \xrightarrow{encode} (k_3, \{k_4, k_5\}, v_2)_{r_2} \\
& \quad (k_4, v_2)_{r_2} \quad (k_5, v_2)_{r_2}
\end{align*}
\]

Note that \((k_1, v_1)\) and \((k_2, v_1)\) cannot exploit the common value \(v_1\), because
these records are sent to different reduce tasks \(r_1\) and \(r_2\), respectively. Since the
other three Map output records have the same value \( v_2 \) and keys \( k_3, k_4, \) and \( k_5 \) are assigned to the same reduce task \( r_2, \) *EagerSH* would transmit only a single encoded record for value \( v_2 \) to reduce task \( r_2. \)

### 4.3.1.1 EagerSH Map Phase

We present *EagerSH* using the query suggestion example introduced in Section 4.2.1. Recall that for queries “mango” and “manga”, the original map function generates five key/value pairs per query, each with a different prefix as the key and the same input query as the value. The partition function assigns keys “m”, “man”, and “mango” to reduce task 1 and “ma”, “mang” and “manga” to reduce task 2.

Figure 4.4 shows the encoding of the Map function output for two different calls, one for input “mango” in map task 1 and one for “manga” in another map task 2. Consider the Map call for “mango”. Instead of sending (m, mango), (man, mango), and (mango, mango) separately to reduce task 1, *EagerSH* sends the more compact (m, \((\{man, mango\}, mango))\), thus eliminating the value duplication. To generate the encoded record, *EagerSH*’s map function first executes the original Map on the given input record. It then groups the original Map’s output by value and partition number (as assigned by the Partitioner). For each group, a single record is emitted. Its key is the *smallest* key in the group; all other keys are added to the value component.

Algorithm 9 shows the pseudo-code. Notice that Map produces key/value pairs, hence MapOutput has a key and a value attribute. The Partition function used by
CHAPTER 4. OPTIMIZING MAPREDUCE PROGRAMS USING ANTI-COMBINING

Algorithm 9: *EagerSH*’s Map Function

**Require:** input tuple $I$

1. $\text{MapOutput} = \text{O-map}(I)$ \{Original map\}
2. $\text{result} = \text{SELECT MIN(O.key) AS key, (setOfOtherKeysInGroup(), O.value) AS value FROM MapOutput O GROUP BY getPartition(O.key), O.value}$
3. $\text{for all } r \in \text{result do}$
4. $\text{Emit}(r\.key, r\.value)$
5. $\text{end for}$

Algorithm 10: *EagerSH*’s Reduce Function

**Require:** $<\text{key}_{k}, \text{KVAL} = \text{listOf(key set K, value)}>$

1. $\text{repeat}$
2. $\text{altKey} = \text{Shared.peekMinKey()}$
3. $\text{if } \text{altKey} < \text{key}_{k} \text{ then}$
4. $\text{O-reduce(altKey, Shared.popMinKeyValues())}$ \{Original reduce on values with smaller key altKey\}
5. $\text{end if}$
6. $\text{until } \text{altKey} \geq \text{key}_{k}$
7. $\text{for all } (\text{K, value}) \text{ in KVAL do}$
8. $\text{for all } \text{key in K do}$
9. $\text{Shared.add(key, value)}$ \{Store for later reduce calls\}
10. $\text{end for}$
11. $\text{end for}$
12. $\text{Values} = \text{KVAL.getValues()}$
13. $\text{if } \text{altKey} = \text{key}_{k} \text{ then}$
14. $\text{Values} \cup \text{Shared.popMinKeyValues()}$ \{Append values with same key in Shared\}
15. $\text{end if}$
16. $\text{O-reduce(key}_{k}, \text{Values})$ \{Original reduce\}

the MapReduce job, getPartition, returns the reduce task to which a key is assigned. And setOfOtherKeysInGroup is a function that returns the set of all keys except for the minimal one in the group.

The minimal key is chosen as the “representative” key for the encoded record, because all Reduce calls in a reduce task happen in ascending key order [22]. This way the other keys can be decoded before their Reduce calls are executed.
4.3. ANTI-COMBINING STRATEGIES

4.3.1.2 EagerSH Reduce Phase

The encoded records generated by the mappers have to be decoded on the reducers. In particular, for each of the keys that were transmitted with the value component of an encoded record, the corresponding key/value pair has to be made available before the key’s Reduce call. To avoid modifications to the underlying MapReduce system, we rely on a data structure called **Shared**. It is defined at the level of a reduce task, i.e., is visible to all Reduce function calls in the same task. Hadoop provides a method called **setup** for initializing such data structures before the first Reduce call in a task, and a **cleanup** method that is executed after the last Reduce call in that task completed.

Figure 4.5 illustrates the difference between the original Reduce and EagerSH’s Reduce. In the original MapReduce execution, reduce task 1 receives all the records with the keys assigned to it by the Partitioner, in key order. It then calls Reduce three times, first for key “m”, followed by “man”, and finally “mango” in the example.

EagerSH’s Reduce only receives three encoded records, in this case all those with key “m”. Before executing the original Reduce call for key “m”, EagerSH’s Reduce scans through all records with that key and inserts into Shared the corresponding key/value combinations for all keys encoded in the value component. In the example, records (man, manga), (man, mango), and (mango, mango) are added to Shared. Then the original Reduce is called for key “m”. The other keys are processed the same way: first the input list is scanned to decode and insert into
Shared, then the original Reduce function is executed. For correctness, *EagerSH’s* Reduce conceptually has to work with the merged list consisting of both the “normal” reduce task input buffer and Shared. We achieve this with a merge-sort style approach that reads from normal input buffer and Shared in lockstep.

Algorithm 10 shows the pseudo-code. ‐KVAL is a list of encoded records, each consisting of a list of keys and the value shared by them. KVAL.getValues returns all values in list KVAL; and Shared.getValues returns all values for a given key. Assume the MapReduce environment just started executing Reduce for key \(k\), i.e., \(k\) is the smallest key in the reduce task’s input buffer for which Reduce has not yet been executed. Since previous Reduce calls might have inserted records with smaller keys into Shared, *EagerSH’s* reduce has to make sure that their Reduce calls are processed first. This is done by the repeat-until loop.

The following for-all nested loop scans the list of encoded records with key \(k\) in the reduce task input and inserts the decoded key/value pairs into Shared. (Since encoding used the smallest key as the representative, it is guaranteed that all newly inserted records have keys greater than or equal to \(k\)!) Finally the original Reduce function is executed on the union of all records with key \(k\) from both KVAL (i.e., the “normal” reduce task input) and Shared.

Decoding has to deal with yet another subtle problem. Consider again the example in Figure 4.5. With *EagerSH*, the input buffer for reduce task 1 only contains records with key “m”. Hence the MapReduce system will only call *EagerSH’s* Reduce function for key “m”. The other two keys—“man” and “mango”—only appear in Shared. Hence *EagerSH’s* Reduce would not be called for them by the MapReduce system. To make sure the remaining records in Shared are processed after the last “regular” Reduce call completed, the reduce task’s clean-up function also has to use a similar repeat-until loop to process all remaining records in Shared. (Recall that cleanup is called automatically by the MapReduce system after all Reduce calls of the task have completed.)
4.3.2 Lazy Sharing Strategy

Our second strategy, LazySH, reduces the cost of the shuffle-and-sort phase, even if all keys and values in the output of a Map call are unique. Hence it will work for cases where traditional compression or EagerSH are not effective. Furthermore, LazySH can significantly improve over those techniques even when there are other sharing opportunities. Consider again the Query-Suggestion problem for a query string $Q$ of length $n$. Even if all prefixes of $Q$ happen to be assigned to the same reduce task, EagerSH would still send a total of $O(n^2)$ data to the reducer. (The encoded record contains each prefix, from 1 to $n$ characters long.) LazySH would only send $O(n)$ data for query $Q$.

Instead of sending Map output from mappers to reducers, LazySH simply transfers the Map input record to all reduce tasks that would have received some of the Map output for this record, e.g.:

\[
\begin{align*}
(k_1, v_1)^{r_1} \\
(k_2, v_2)^{r_1} \\
(k_3, v_3)^{r_2} \\
(k_4, v_4)^{r_2} \\
(k_5, v_5)^{r_2}
\end{align*}
\]

Since reduce tasks $r_1$ and $r_2$ would have received Map output records for input $(k_{in}, v_{in})$, those and exactly those two reduce tasks will receive $(k_{in}, v_{in})$. Since Reduce ultimately needs the Map output records, these have to be generated lazily on the Reducer by re-executing Map there.

4.3.2.1 LazySH Map Phase

Figure 4.6 illustrates LazySH for our running example. For input query “mango”, instead of sending $(m, (\{\text{man, mango}\}, \text{mango}))$ to reduce task 1, LazySH transfers only $(m, \text{mango})$. Algorithm 11 shows the pseudo-code. It first computes the output of the original Map call for input record $I$, then finds the minimal key for each
reduce task (i.e., partition) that would have received some of that output. Finally
together record $I$ is emitted for each of these minimal keys.

The SQL statement in Algorithm 11 highlights the difference to *EagerSH*’s
map in Algorithm 9. Since *LazySH*’s Map function groups the original map output
only by partition (and not also by value), there are more data reduction opportu-
nities. And by using map input $I$ as the value, it does not need to transmit all the
other keys. This is crucial for asymptotic (in Map input size) data reduction for the
Query-Suggestion problem.

To make the differences between the original MapReduce program, *EagerSH*,
and *LazySH* more tangible, consider the following example of the real query
“watch how i met your mother online”. If the Partitioner assigns all its prefixes
to the same reduce task, then *EagerSH* would transmit $34 \cdot 35/2 + 34 = 629$
characters, significantly improving over the original program’s output of size
$34 \cdot 35/2 + 34 \cdot 34 = 1751$. However, *LazySH* would do even better, requiring
Algorithm 12: LazySH’s Reduce Function

Require: \(< k_v, VAL = listOf(map input I)>\)

1: repeat
2: altKey = Shared.peekMinKey()
3: if altKey < key \(k_v\) then
4: O-reduce(altKey, Shared.popMinKeyValues()) \{Original reduce on values with smaller key altKey\}
5: end if
6: until altKey \(\geq\) key \(k_v\)
7: for all \(I\) in VAL do
8: MapOutput = O-map\(I\) \{Original map\}
9: for all (key, value) in MapOutput do
10: if getPartition(key) = this.partitionNumber then
11: Shared.add(key, value)
12: end if
13: end for
14: end for
15: O-reduce(key \(k_v\), Shared.popMinKeyValues()) \{Original reduce\}

Figure 4.7: LazySH’s Reduce Phase for Query-Suggestion

only \(1 + 34 = 35\) characters to be transmitted, using “w” as the key and the complete query as the value component.

4.3.2.2 LazySH Reduce Phase

The reduce tasks of LazySH receive Map input, not output, therefore decoding in the reducer requires re-execution of the original Map function. Decoded records are stored in a reduce-task level data structure Shared to allow data transfer between individual Reduce calls, as discussed for EagerSH’s Reduce. Since not all outputs
of a given Map call might be assigned to the current reduce task, the Partition function has to be used to determine those that are. Algorithm 12 shows the pseudo-code for LazySH’s Reduce. It essentially is identical to EagerSH’s Reduce, except for the decoding process that calls the original Map and getPartition functions.

For simplicity, we illustrate the algorithm with an example in Figure 4.7. Similar to EagerSH’s reduce, all values with minimal key “m” are present in the input buffer of reduce task 1. When the MapReduce system calls LazySH’s Reduce for prefix “m”, the original Map function is applied to generate all original Map output pairs. For each output record the Partition function is applied to identify those records that belong to reduce task 1. E.g., when input record (m, manga) is processed in this Reduce call, only (m, manga) and (man, manga) are inserted into Shared.

4.3.3 The Shared Data Structure

The Shared data structure used in the reduce phase is designed to efficiently manage decoded key-value pairs and return all records that have the minimal key. It maintains the minimal key using a min-heap, and an in-memory hash-table that maps keys to their corresponding values. As the memory reserved for Shared fills, the data is spilled to local disk in sorted key order. This is done by repeatedly removing the minimal key from the min-hash and then removing the list of values for it from the hash-table, writing it sequentially to disk. this mirrors what happens during the Map phase of the original MapReduce program. For each spill, the minimal key is recorded in Shared. If the number of spill files exceed the merge threshold, they are merged, again mirroring the standard map phase processing [80].

As shown in Algorithms 10 and 12, Shared provides three functions: peekMinKey(), popMinKeyValues(), and add(key, value). As detailed in Section 4.3.1.2, Anti-Combining only reads key-value pairs stored in Shared having minimal key. The min-heap is used to find the minimal key at constant cost when peekMinKey() is called. All values associated with the minimal key are retrieved
4.3. ANTI-COMBINING STRATEGIES

using popMinKeyValues(). If the values are in the in-memory hash-table, they are found through a lookup using the minimal key. If values with the minimal key are spilled to disk, since the spill files are sorted by key, Shared performs a buffered sequential read on the relevant spill files, never needing random accesses. Add(key, value) is performed on the in-memory hash-table in constant amortized running time. Inserting the key into the min-heap requires logarithmic time.

Using Combine in the Reduce Phase. For MapReduce programs that admit the use of a Combiner during the map phase, the provided Combine function can also be leveraged during the reduce phase of *EagerSH* and *LazySH*. The Combine function summarizes a set of Map output records that have the same key. Hence it can be applied to reduce the amount of data managed in Shared. Instead of adding each decoded key-value pair separately, Shared can immediately Combine values and maintain a single record for each unique key. Our experiments show that Combine reduces the size of Shared significantly, sometimes allowing in-memory processing without spilling to disk. Combining during the Reduce phase is highly effective, because records with the same key end up in the same reduce task. Hence even for applications where Combining is not effective in the Map phase, leveraging the Combine function can be highly effective for reducing the size of Shared in the reduce phase.

4.3.4 Enabling Adaptive Runtime Optimization

In this section we discuss how to enable Anti-Combining through syntactic transformations of a given MapReduce program.

4.3.4.1 Program Transformation

The original mapper class is replaced by a new mapper class called AntiMapper as shown in Figure 4.8. This is done by modifying the class name in the statement that sets the mapper class for the program. Notice that we do not need to modify the original class. Instead, AntiMapper is automatically generated to contain an object
Figure 4.8: Syntactic rewrite of original mapper class to enable Anti-Combining of the original mapper class and an extended context object. The former enables the use of the original Map functionality. The latter extends Hadoop’s context class and is needed because in Hadoop mappers emit their output to the context object. The extended context intercepts the original Map output and replaces it by the encoded version. Notice that MapReduce implementations other than Hadoop would have to rely on a similar mechanism for collecting Map output, enabling a similar interception approach.

For each individual Map call, the AntiMapper has to adaptively choose between EagerSH and LazySH to pick the encoding strategy that decreases data transfer the most. In fact, the encoding decision is made independently for each partition (i.e., reduce task) the output records of the Map call are assigned to. There are two reasons for this: First, since different tasks cannot share data, the encoding decision for one partition does not affect the choices for the others. Second, the greater flexibility enables greater data reduction compared to enforcing the same decision for all partitions. Notice that the original program’s unencoded output is a
4.3. ANTI-COMBINING STRATEGIES

special case of *EagerSH* when the set of keys included in the value component is empty. For proper decoding in the reducers, a flag is added to the encoded record’s value component to indicate which strategy was used.

As discussed for *EagerSH* and *LazySH*, AntiMapper’s map function first executes the original program’s Map function on input record (key, val) (through the o_mapper object) and then partitions the output, which was intercepted by the extended context object, using the original program’s partitioner (accessed through the context, as usual). After these steps, the **exact execution cost of the original map and getPartition for this input record (key, val) are known**. (We currently measure cost in terms of CPU time, but one could similarly use measures that include I/O cost.) We also know to how many reduce tasks (key, val)’s output records will be sent. From this we can compute the total cost of re-executing o_mapper.map and getPartition if *LazySH* was used for encoding. To deal with expensive map and getPartition cases, Anti-Combining uses a cost threshold $T$ that disables *LazySH* if
total re-execution cost exceeds the threshold (see figure 4.8). The “ideal” setting for $T$ depends on user preference. Smaller $T$ limits the overhead from “duplicate” Map and getPartition executions, but limit Anti-Combining’s encoding choices. Hence larger $T$ enables greater decrease of network transfer at the cost of higher CPU load. In the extreme, if $T$ is set to $\infty$, then Anti-Combining chooses freely between EagerSH and LazySH the one that minimizes data transfer cost. Setting $T = 0$ forces Anti-Combining to only use EagerSH, i.e., completely avoid any duplicate Map and getPartition calls.

As for the mapper class, we also replace the original reducer class with our AntiReducer in the program statement that sets the reducer class. AntiReducer, as shown in Figure 4.9, is also automatically generated from the given original reducer class. It essentially performs the decoding work as introduced in Alg. 10 and 12. There are some non-trivial technical challenges to enable reading from the union of the regular reducer input and from Shared, which contains decoded records. In particular, for the current minimal key, there could be EagerSH- and LazySH-encoded records in the regular input buffer, in addition to already decoded records (from previous reduce calls) in Shared. To correctly deal with such cases, AntiReducer’s reduce function first iterates over all values in the reducer’s input in order to decode them. For eagerSH-encoded records such as (key1, (key2, . . . , val)), it also inserts (key1, val) into Shared. Hence after the “for all val in values” loop, Shared contains all key-value pairs that the Reduce call in the original program would have received for that key.

Notice that Shared.popMinKeyValues removes the key from Shared’s min-heap and all its associated values from the hash-table, passing an iterator for the removed values to the o_reducer.reduce call. The grouping comparator is used to determine key equality, ensuring that Shared’s behavior is consistent with the original MapReduce program when the user provides a grouping comparator that is different from the regular key comparator, e.g., for secondary sort [80]. (Notice that records are removed from Shared in key order, hence the values passed to o_reducer.reduce are in key order.)
Notice that a Combiner is defined as a reducer class. Hence we apply the same syntactic transformation to it. Like for mapper and reducer, in the given program, the statement setting the Combiner class then can be changed to select the Anti-Combining enabled version. Since the Combiner is optional, Anti-Combining has a second parameter (in addition to $T$), a flag $C$ that lets the user choose to disable the Combiner by setting $C = 0$.

### 4.3.4.2 Anti-Combining in Practice

Anti-Combining can be enabled for any given MapReduce program, because it treats the original units of functionality (map, reduce, setup, cleanup, Combiner, Partitioner) as blackboxes. However, there are cases that require a more careful analysis.

**Non-determinism.** Non-determinism affects the $LazySH$ strategy because the re-executed Map and getPartition function might return different results. Hence whenever non-determinism in Map or getPartition can affect the Map output keys or their assignment to reduce tasks, then $LazySH$ has to be disabled. This is done by the user who would simply set threshold $T = 0$ when she suspects such effects of non-determinism. In practice we have not encountered examples for this type of application.

**Programs without Combiner.** If a program has no Combiner, then Anti-Combining can be safely enabled. $EagerSH$ in the worst case, i.e., when there are no sharing opportunities, would add an insignificant overhead due to the additional bits needed to indicate the encoding of a record. All critical steps—write Map output to local disk, sort/merge it locally, transfer it to reducers, merge it on reducer, read it in Reduce call—become less costly thanks to the smaller encoded Map output. The additional cost on the reducers for decoding and managing Shared is the same or less compared to the reduction in cost on the mappers for writing sorted Map output to local disk and merging it there. Our experiments support this analysis. For $LazySH$ the analysis is similar, except that expensive Map and getPartition
calls can result in significantly higher CPU cost. As our experiments show, the user can effectively control the tradeoff between data size reduction and CPU cost increase through parameter $T$.

**Combiner on or off.** For programs with Combiner, the user has the choice to leave it on or turn it off by setting Anti-Combining flag $C$ accordingly. If the Combiner results in small data reduction in the original program, e.g., less than 10%, then it should be turned off as it essentially undoes some of Anti-Combining's encoding gains without delivering significant data reduction. Somewhat surprisingly, if a Combiner is highly effective, e.g., reduces data transfer by a factor of 10, then it will also benefit from Anti-Combining. Our experiments show this for Word Count. Intuitively the reason is that Map writes smaller encoded output (compared to the original program) and the Combiner reads these smaller data. The Shared data structure remains small because the effective Combiner reduces data size as it decodes. In general the decision about turning the Combiner off can be made by running the program with and without Combiner on a sample of input file splits, choosing the winner based on this sample run.

**Partitioner.** Careful design of a Partitioner can increase the impact of Anti-Combining by assigning records with commonalities to the same reduce task. Defining an appropriate Partitioner is the responsibility of the programmer who needs to analyze the statistical properties of the Map output. This is part of the MapReduce program design process already, because the effectiveness and scalability of parallel programs in general depend on a good partitioner to appropriately distribute records over reduce tasks. Automatically finding a good partitioner is beyond the scope of this paper since different data properties and applications lead to different sophisticated partitioning techniques [87, 61, 37, 49]. However, in our experiments we demonstrate that even easy-to-design Partition functions already lead to significant cost savings, which can be further improved by careful analysis of data and program properties.

**Total cost versus running time.** Anti-Combining focuses on reducing network transfer to lower the stress on this crucial shared resource. As a side-effect, writing
and reading the smaller encoded data locally on mappers and reducers often also decreases local I/O and CPU cost. Hence we often see improvements not only in total cost, but also in running time. The main exception are cases where Anti-Combining introduces additional skew because of $\textit{LazySH}$ encoding. In particular, a reducer dealing with many $\textit{LazySH}$ encoded records might receive a large share of additional CPU and local I/O cost (for spilling $\textit{Shared}$ to disk). This skew is not a concern when the goal is to optimize for throughput. While the overloaded machine delays job completion, other machines finish early and can be used by other jobs in the cluster. Also, by choosing a smaller threshold $T$, the user can control how aggressively she wants to optimize for lower cost (and hence higher throughput) at the cost of potentially longer job completion time.

### 4.4 Experimental Evaluation

We discuss representative results for Anti-Combining on real data. All experiments were performed on a 12-machine cluster running Hadoop 1.0.3 [1]. One machine served as the head node, while the other 11 were worker nodes. Each machine has a single quad-core Xeon 2.4GHz processor, 8MB cache, 8GB RAM, and two 250 GB 7.2K RPM SATA hard disks. In total, the cluster therefore has 44 worker cores with 2GB memory per core available for Map and Reduce tasks. All worker cores are used in the reduce phase resulting in 44 reducers. The distributed file system block size is set to 64MB and all machines participate as storage nodes for the DFS.

All machines are directly connected to the same Gigabit network switch. Notice that this configuration of comparably few machines connected to a fast network and single switch is a challenging setup for Anti-Combining, which focuses on reducing network cost. In larger data centers with more machines and multi-hop communication between them, Anti-Combining will deliver even more benefits.

In our experiments we compare the performance of the original program written by an experienced MapReduce programmer (Original) against the transformed version with Anti-Combining. For Anti-Combining, we compare a version that
only uses EagerSH, another that only uses LazySH, and the adaptive version introduced in Section 4.3.4 (AdaptiveSH). The following data sets were used:

QLog contains 140 million real queries issued to a commercial search engine between 03/01/2011 and 04/30/2011. Each input record consists of an anonymous user identifier, the search query, and two query features (total number of occurrences of the query in search logs, total number of resulting links users browsed). The average search query string consists of 19.07 characters and the total data size is 4.3GB.

Cloud is a real data set containing extended cloud reports from ships and land stations [38]. There are 382 million records, each with 28 attributes, resulting in a total data size of 28.8GB. Note that this is the same dataset used in Section 3.5.

RandomText is a synthetic data set containing randomly generated text records with around 360GB total size.

4.4.1 Anti-Combining Overhead Analysis

We measure the overhead of Anti-Combining for workloads where it is not applicable or ineffective. This is done by executing the Hadoop Sort program on RandomText. Sorting emits a single Map output record for each input record to be sorted in Reduce. Therefore, it is a representative workload where Anti-Combining is not beneficial. The adaptive algorithm automatically chooses EagerSH encoding without any shared keys, which degenerates to the original record plus a few bits that are needed to flag the type of encoding used.

AdaptiveSH results in only 0.2% more total disk read/write than Original program due to extra bits used for the encoding type flag. This also results in 0.15% more total data transfer cost. The total CPU time spent using AdaptiveSH is 7.8% more than the original program which is the overhead of looking for sharing opportunities among Map output records. The total runtime increased by 1.7% when AdaptiveSH is used. This supports our claim that Anti-Combining incurs little overhead even when it is applied to a job that does not benefit from it.
4.4.2 Query-Suggestion

We study Anti-Combining for the Query-Suggestion problem on QLog and explore the effect of the choice of Partitioner by comparing three alternatives. The first corresponds to an inexperienced programmer who simply relies on the standard hash Partitioner (Hash). The second assumes a programmer who understands the behavior of Map and realizes that maximal sharing is possible when all keys with the same first letter are sent to the same partition (Prefix-1). (Figuring this out is actually not that difficult, because sharing opportunities are determined by the Map and Partition functions and do not require complex program analysis.) Finally, Partitioner Prefix-5 uses the first five characters. It addresses the concern that Prefix-1 might generate only a small number of reduce tasks, equal to the number of distinct first characters of search queries in QLog. Note that none of these functions are specially designed based on input data statistics which may be easily constructed by a programmer. More careful data analysis may even lead to better partition functions that increase sharing opportunities.

Figure 4.10 shows the total Map output size for each strategy and partition function. Note that Original produces the same output size with all partition functions since it does not exploit sharing opportunities. On the other hand, EagerSH and LazySH effectively reduce the amount of data transferred for all partition functions, achieving reduction up to a factor of 27. Our adaptive strategy, AdaptiveSH, achieves the best result in all cases (except for Prefix-1) as expected since it adaptively picks the most suitable encoding type per record. For Prefix-1, AdaptiveSH encodes all map output records using LazySH encoding but produces slightly larger output than pure LazySH due to the extra bits used for identifying the encoding type used.

4.4.3 Query-Suggestion With Combiner

We repeated the same experiments, but now for the case that the original program came with a Combiner. A Combiner can reduce data transfer cost for both the
original program and when used together with Anti-Combining (Section 4.3.3).

Figure 4.11 shows the total Map output size, indicating that Original achieves about 12% reduction in Map output size compared to the no-Combiner result in Figure 4.10. EagerSH experiences about the same improvement, while LazySH does not benefit from a Combiner in the map phase since it cannot apply the Combine function to encoded map input records.
4.4. EXPERIMENTAL EVALUATION

Figure 4.12: Total Map Output Size for Query-Suggestion using Combiner and Compression

Table 4.2: Total Cost Breakdown of Compression Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Deflate</th>
<th>Gzip</th>
<th>Bzip2</th>
<th>Snappy</th>
<th>Anti-Combining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Disk Read(GB)</td>
<td>65</td>
<td>65</td>
<td>56</td>
<td>105</td>
<td>15</td>
</tr>
<tr>
<td>Total Disk Write(GB)</td>
<td>82</td>
<td>82</td>
<td>70</td>
<td>133</td>
<td>21</td>
</tr>
<tr>
<td>Total Map Output Size(GB)</td>
<td>18</td>
<td>18</td>
<td>15</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>Total CPU Time(sec)</td>
<td>126.893</td>
<td>125.244</td>
<td>332.386</td>
<td>77.359</td>
<td>27.950</td>
</tr>
</tbody>
</table>

4.4.4 Query-Suggestion With Compression

We provide results for the same set of experiments, but now with both Combiner and Map output compression enabled.

We tried the following compression algorithms: Deflate, Gzip, Bzip2, and Snappy. The detailed cost breakdown of these algorithms and Anti-Combining for the Query Suggestion task can be found in Table 4.2.

As Figure 4.12 shows, compression works very well in reducing the total amount of data transferred for all strategies. Anti-Combining still performs better than Original for all partition functions. The results support that Anti-Combining works well in combination with compression. Note that the compression algorithm reported in the Figure is Gzip which achieved significantly lower total CPU time with top- or near-top compression rate for the Original program for all applications.

The total runtime of the experiments also support that our approach is light-
weight compared to compression. *EagerSH* reduces the total runtime of *Original* by a factor of up to 4; *LazySH* achieves up to a factor of 16 improvement.

### 4.4.5 Effect on Disk I/O and CPU

The cost breakdown for total disk read/write and total CPU time of the Query Suggestion experiments using *Prefix-5* are shown in Table 4.3. Suffixes “-CB” and “-CP” refer to the experiments when the algorithms are executed using combiner and compression respectively.

*AdaptiveSH* effectively reduces the total amount of disk I/O, achieving reduction up to a factor of 3.8 and 4.1 for total amount of reads and writes respectively. Although Anti-Combining performs extra map calls in the Reduce phase, it also reduces the total CPU time by a factor of 5.5. We believe that *Original* suffers from low CPU utilization because of the large amount of disk I/O and network transfer. Note that total CPU time measured for *Original-CP* is also lower than *Original* although CPU intensive compression is performed on the map output buffer.

The Shared data structure in *AdaptiveSH* spills the data to disk 1575 times. On the other hand, *AdaptiveSH-CB* effectively applies Combine on the Shared data structure in the Reduce phase and manages to keep all unique keys and aggregated values in-memory. This reduces the total amount of disk reads and writes by a factor of 2.4 and 2.1 respectively.

### 4.4.6 CPU Intensive Workloads

We analyze the performance of Anti-Combining for CPU intensive workloads by adding extra CPU intensive calls to the Map function. As explained in Sec-
4.4. EXPERIMENTAL EVALUATION

AdaptiveSH performs runtime cost-based optimization and switches to EagerSH encoding in order to avoid re-executing expensive Map calls in the Reduce phase. We measure the performance of Anti-Combining using two extreme runtime thresholds. Adaptive-0, with runtime threshold \( T = 0 \) that results in EagerSH encoding for all map output records and Adaptive-\( \infty \) with \( T = \infty \) that does not restrict the choice of eager vs lazy. We also provide results for Adaptive-\( \alpha \) which uses 400 microseconds runtime threshold. We modified the Map function of Query-Suggest task in order to add extra CPU intensive work by computing Fibonacci numbers.

Figure 4.13 shows the effect of increasing Map function call cost on the total CPU time for all algorithms. The x-axis represents the amount of busy work added. When \( x_i \) extra work is added, each map call computes the first \( 25000 \times x_i \) Fibonacci numbers. For low Map call cost, Adaptive-\( \infty \) achieves lower total CPU time as expected by optimizing for map output size. As the Map call cost increases, executing Map function in the Reduce phase for lazy encoded records increases as well, resulting in higher total CPU time increase for Adaptive-\( \infty \). The area between Adaptive-0 and Adaptive-\( \infty \) plots represents the space of runtime thresholds that could be assigned by an optimizer. As expected, Adaptive-\( \alpha \) uses lazy encoding where beneficial for low cost map calls and converges to Adaptive-0 as the map function gets more expensive. The experiments show that Anti-Combining can be optimized for the dominating cost of a MapReduce job. In the next section, we also show effectiveness of Anti-Combining on two different CPU intensive applications.

4.4.7 Processing Joins using Anti-Combining

In this section, we study the performance of Anti-Combining for join processing.

We execute the following join query on the Cloud data set:

```
SELECT S.date, S.longitude, S.latitude, T.latitude
FROM Cloud AS S, Cloud AS T
WHERE S.date = T.date AND S.longitude = T.longitude
    AND ABS(S.latitude - T.latitude) <= 10
```
CHAPTER 4. OPTIMIZING MAPREDUCE PROGRAMS USING ANTI-COMBINING

For our experiments, we use the memory-aware version of the 1-Bucket-Random algorithm as detailed in Section 4.2.2. This algorithm ensures that data chunks are just small enough so that each local join task on a reducer can be executed in-memory. Optimization opportunities arise because each input record might be assigned to multiple chunks to enable parallel processing.

Figure 4.14 compares the total Map output size produced by each algorithm. Results for LazySH are not reported, because AdaptiveSH ended up choosing LazySH encoding for all map output records. The creation of data chunks for parallel join processing causes an average data replication by a factor of 67 when comparing total map phase output to input. Since the join does not admit the use
of a Combiner, this huge amount of data replication directly affects Original, producing 926 GB of Map output. The adaptive Anti-Combining technique reduces Map output by a factor of 9.5. Note that if we had not used the in-memory join version, i.e. amount of input for each reduce group found by 1-Bucket-Random had fit reducer memory, the map input replication rate would be 6.44.

Since a Combiner was not an option for the join, we repeated the same experiments using compression on Map output data (denoted by suffix “-CP”). All techniques result in similar compression rates. Note that Map output size of Original with compression is still significantly larger than for Anti-Combining without compression.

Figure 4.14 also compares the runtime for each technique, showing that Anti-Combining improved over Original by a factor of 9.6 and 6 for the no-compression and compression scenarios, respectively. Notice that the 1-Bucket-Random algorithm achieves almost perfect load balancing between the different worker nodes. Hence the runtime improvement tracks closely the reduction in Map output size.

4.4.8 Anti-Combining on Different Join Workloads

We study the behavior of Anti-Combining techniques on 1-Bucket-Random using a selective join query by varying the max-reducer-input size limit that is used to satisfy the memory constraints of nodes.

In Chapter 3, we have shown both analytically and empirically that 1-Bucket-Random can only be improved for selective joins. The goal of Anti-Combining is to reduce the load on the network and eliminate the negative effects of 1-Bucket-Random’s high input replication rate, especially when it performs extra data replication due to memory constraints forced by the join executed.

We execute the same join query above, on a smaller sample of Cloud data set with a total input size of 53 million records. For the experiments, the original 1-Bucket-Random and Anti-Combining enabled version is compared for different memory constrains (max-reducer-input size limit). For each experiment, the total
CHAPTER 4. OPTIMIZING MAPREDUCE PROGRAMS USING ANTI-COMBINING

Figure 4.15: Memory Size Limit Effect on 1-Bucket-Random and Anti-Combining

amount of data transferred between Map and Reduce phases is compared. Note that
the memory-aware join algorithm uses a pair of integers as key. The first value is the
partition number that is found by the original 1-Bucket-Random. The second value
is the sub-partition number that is found by applying the memory-aware M-Bucket
algorithm to the partition found by 1-Bucket-Random. The Partition function uses
the first value to partition key/value pairs created in the Map phase.

Figure 4.15 compares the total Map output size produced by each algorithm.
As expected, the total Map output size of 1-Bucket-Random increases as the mem-
ory limit decreases, since more (smaller) partitions are generated by 1-Bucket-
Random, resulting in greater input replication. On the other hand, the AdaptiveSH
enabled 1-Bucket-Random algorithm manages to reduce the total Map output of
1-Bucket-Random more aggressively as the memory limit decreases (as 1-Bucket-
Random replicates more data). When the memory size limit is 8 million records,
Anti-Combining reduces 1-Bucket-Random’s total Map output size by a factor
of 1.6. When the memory size limit is 1 million records, Anti-Combining re-
duces 1-Bucket-Random’s total Map output size by a factor of 11.3. This is an
expected behaviour of Anti-Combining because more data sharing opportunities
become available when the amount of data replication in the Map phase increases
for 1-Bucket-Random. Note that the smallest total Map output size achieved by
1-Bucket-Random, if there is no memory constraint, is 6.4 GB.
5.1 Introduction

In this chapter, we present Scolopax, a data exploration tool we developed that helps scientists find interesting patterns in scientific data sets [62]. Scolopax has a simple user interface to specify users’ pattern search preferences. Scolopax provides scalable algorithms for exploratory search including model training, model summary generation, correlation and relationship search between summaries, and summary ranking based on user preferences. All of these algorithms are scalable and run in parallel using MapReduce. Correlation and relationship search is performed using our join algorithms described in this dissertation.

Across many disciplines, scientists have to cope with a flood of data. One of the grand challenges of data-driven science is to find interesting patterns in massive high-dimensional data sets that may lead to new hypotheses. This process is currently limited by the large amount of required human effort and the high computational cost. Scalable exploration tools are required to make pattern search in big data (almost) as easy as searching the Web, while hiding technical data management challenges.

Scolopax’ functionality is demonstrated for a real data set that was extracted from eBird reports [59] in collaboration with the Cornell Lab of Ornithology, one
of the world’s leading research institutes for birds. eBird is a citizen-science project that takes a crowdsourcing approach, harnessing the power of birders around the world to collect observations of birds with broad scale and fine resolution. It has amassed one of the largest biodiversity data resources in existence and is rapidly growing, at a rate of millions of observations monthly. Since all reports contain location and time, the eBird data is joined with high-quality environmental and geographic data sets, e.g., climate, habitat, and human census data.

The following examples illustrate patterns that could be found and the corresponding hypotheses proposed by the domain scientists. Figure 5.1a is a 1-dimensional summary that shows how the estimated probability of observing the Acorn Woodpecker in California varies with local human population density. After seeing the strong relationship between increasing human population density and lower woodpecker probability, the scientists hypothesized that the bird’s reliance on dying tree branches for storing acorns conflicts with properties of habitats in more densely settled areas.

The plots in Figure 5.1b represent the annual estimated observation probability of two bird species in Northern New England. The strikingly complementary biannual cycles for Purple Finch and Common Redpoll hint at habitat competition, possibly driven by the availability of local food sources.

Broad-scale migratory behavior can be explored through analysis of similar univariate summaries describing how local occurrence probabilities vary through-
5.1. INTRODUCTION

Figure 5.2: Average cluster trajectories for spring migration and cluster map of Tree Swallow

out the year. Each summary indicates the time of the year that the species is likely to be observed in the given locality. By clustering local occurrence trajectories across broad extents, the geographic patterns of movement that characterize migrations can be revealed. Figure 5.2 presents an example for spring migration flyways of the Tree Swallow. In Figure 5.2a, the average spring cluster trajectories are shown; Figure 5.2b presents the visualization of the regions and to which of two clusters they belong. Based on this image, scientists hypothesize that an early initial wave of migrants is flying through the dark Piedmont/Southern Appalachian region and the larger wave of migrants moves north a little later through the light region.

Supporting discovery of such patterns is challenging. Complex prediction models need to be trained to address issues such as noise, sparseness, and skew in the raw data. Summaries need to be extracted from the models and raw data in order to find intelligible patterns [45]. An interesting hypothesis could be found in some “slice” or “dice” of the data space, e.g., a certain trend might show only in some small region or a combination of habitat features and elevation ranges. Hence to broadly explore possible hypotheses, a huge number and variety of such summaries, both one- and multi-dimensional, needs to be generated. To find correlations, e.g., habitat competition between two species or variables with similar effect on a species, joins with complex predicates are needed, e.g., using inequality conditions. As some promising patterns are found, scientists want to interactively post-process the result set, e.g., by refining a geographic selection through drill-
Scolopax achieves high performance and scalability by relying on novel data management techniques for parallel model-summary generation and parallel processing of theta-joins presented in this thesis. In Section 5.2, we present how model-summary generation is achieved efficiently in Scolopax. Then, in Section 5.3 we describe how our efficient theta-join algorithms are used in Scolopax for correlation analysis. The system architecture is presented in Section 5.4. Finally, we present the Scolopax interface and how it can be used for interesting pattern discovery in high-dimensional data sets in Section 5.5.

5.2 Model-Summary Generation

As the example in Figure 5.1a illustrate, intelligible low-dimensional summaries are the basic ingredient for the patterns that motivated a variety of different types of hypotheses. For various reasons, most of them related to data quality and sparseness\footnote{Note that sparseness is still a major issue for Big Data due to the large number of attributes.}, many scientists prefer to work with sophisticated prediction models that are trained on the collected data. Summaries are then derived from the model, essentially capturing the most important low-dimensional relationships contained in the complex (high-dimensional) model.

Generating a single summary is fairly expensive, requiring a large number of data records to be processed by the model. For broad exploration, scientists want to include many possible summaries. Hence summaries are generated for many combinations of variables, for different geographical regions, but also for a variety of other partitionings, e.g., by combinations of habitat and climate features. (This enables discovery of more complex interactions, e.g., where some habitat change only affects bird populations in certain climate types.)

Based on a careful analysis of workload and model properties, sophisticated techniques have been proposed for speeding up computation of massive model-summary collections by one or more orders of magnitude [66]. The authors model
the problem as a database-style query plan and show how it can be decomposed into
two much cheaper steps without affecting result quality. This approach relies on
memoization (which short-circuits expensive model evaluation processes), pushing
materialized aggregates into model evaluation, and effective bulk-computation that
exploits common properties of different summaries. In addition to reducing total
cost, Scolopax also facilitates a parallel implementation in MapReduce that shows
near-optimal speedup.

5.3 Correlation Analysis using Theta-Joins

Once a large collection of summaries is generated, Scolopax explores and ranks
patterns by processing these summaries using relational operators such as selec-
tion, grouping and join, but also data mining techniques like clustering. The join
operator is of particular interest, because scientists are very interested in discover-
ing various types of relationships and correlations. For instance, the plots in Fig-
ure 5.1b represent the annual estimated observation probability of two bird species
in Northern New England.

Exploring various types of relationships and correlations require flexible join
predicates, not just equality. For example, the following query is executed in
Scolopax to search for habitat competition between species (similar to Figure 5.1b):

```
SELECT S.plotData, T.plotData, S.sumAttr
FROM Summaries AS S, Summaries AS T
WHERE S.species ≠ T.species
AND S.sumAttr = T.sumAttr
AND S.region = T.region
AND |\phi(S.plotData, T.plotData)| > \epsilon
```

This query is a self-join to find summaries of different species having the same
summary attribute (x-axis of the plot) and complementary trends in the same region
where \(\phi\) is a user-defined distance function and \(\epsilon\) is a threshold parameter set by the
user. A variety of other non-equality conditions are necessary for scenarios where
scientists may investigate correlations among summaries with different summary attributes in various regions.

To compute this type of general theta-join, Scolopax uses our novel theta-join processing techniques presented in this thesis.

### 5.4 Scolopax System Architecture

Scolopax is designed as a data-driven Web application that allows users to interact with the system through simple user interfaces. The UI runs in any standard Web browser, connecting to an application server. All the heavy lifting is performed in parallel on a cluster running the Hadoop MapReduce system. The Hadoop and HBase powered backend performs tasks such as parallel training of models that accurately predict species observation probabilities, generation of huge collections of model and raw data summaries, join processing for correlation analysis, and clustering. The system architecture and interaction between the components are illustrated in Figure 5.3.

The distributed HBase database stores previously executed queries and their results. For each incoming query, the server first checks with the HBase table if the query is previously executed. If so, the results are submitted to the client. If not, the query is translated into a MapReduce job and executed on the cluster. The output of the MapReduce job is directly written to the HBase table and submitted to the client.

### 5.5 Discovery of Interesting Patterns using Scolopax

This section shows how Scolopax can be employed for discovery of interesting patterns in real data about the occurrence of different species of birds. The data set was introduced in Section 5.1 and is based on a subset of the eBird observations made by citizen scientists in the lower 48 states of the USA between 2000 and 2011 [59]. All the functionalities described in this chapter are executed using a
5.5. DISCOVERY OF INTERESTING PATTERNS USING SCOLOPAX

Scolopax comes with three different interfaces: “Summary Ranker”, “Correlation Finder”, and “Cluster Ranker”, all discussed in the remainder of this section.

5.5.1 Summary Ranker

The summary ranker interface allows users to make discoveries in a real data set that is actively studied by biologists and ecologists world-wide. Users can search for interesting model summaries that may help identify threats to bird species. Users select any set of species and can interactively pick regions of interest by zooming and sub-partitioning on a map as shown in Figure 5.4. To support broad exploration, Scolopax will then generate all possible one-dimensional summaries (for each of the input data attributes) for each of the regions the user selected. The corresponding summaries are ranked by a user-selected rank measure. While Scolopax can support any function computable on a summary (including its meta-data), for

Figure 5.3: Scolopax System Architecture
CHAPTER 5. SCOLOPAX: EXPLORATORY ANALYSIS OF SCIENTIFIC DATA

Figure 5.4: The Scolopax summary generator input screen allows the user to select species of interest, explore areas of interest on the interactive map and create partitions on latitude and longitude.

simplicity the interface currently only offers commonly used ones. “Max-Min” and “Standard Deviation” help identify important variables by ranking summaries showing greater changes in output value above those closer to “flat lines”. Slope-based measures are used to identify summaries showing strong trends, e.g., declining species probability with increasing human presence.

After the query is submitted, Scolopax returns a ranked list of patterns that best match the users’ interest as shown in Figure 5.5. Hovering over each plot reveals the region the summary belongs to on the map. Since the initial summary generation is intentionally broad to include many potentially interesting patterns, the result usually contains highly-ranked summaries that turn out to be of little interest. The user can then interactively filter results by selecting attributes she wants to further investigate or remove. At any point, the user can go back and modify the initial selections to execute another query. Scolopax automatically stores executed query results in order to be able to respond rapidly to previously executed queries.
5.5. DISCOVERY OF INTERESTING PATTERNS USING SCOLOPAX

Figure 5.5: The Scolopax summary ranker output screen allows the user to interactively examine the ranked list of summaries, see the corresponding region on the map, filter results for further investigation, and resubmit a refined query.

5.5.2 Correlation Finder

Users can also search for habitat competition and other correlations, e.g., variables with similar effects on a single species. Similar to the summary ranker, users first select species and region(s) of interest on the map. They also select on which data attributes and region properties an equality or inequality condition should be enforced as shown in Figure 5.6. Furthermore, the user can specify if she is interested in correlation or anti-correlation and choose from a variety of data manipulations that should be applied before computing the correlation score, e.g., shifting or scaling. This allows a great variety of possible join conditions to be expressed. The join
CHAPTER 5. SCOLOPAX: EXPLORATORY ANALYSIS OF SCIENTIFIC DATA

Figure 5.6: The Scolopax correlation finder input screen allows the user to select species of interest, explore areas of interest on the interactive map, create partitions on latitude and longitude, and express a variety of join conditions.

results are presented ranked by strength of correlation and the user can filter and post-process them to home in on the most interesting ones as shown in Figure 5.7.

5.5.3 Cluster Ranker

Users can search for spatial patterns, including migration using Scolopax. Similar to the previous scenarios, a set of species and regions of interest are selected interactively. In addition, attributes of interest are selected, e.g., the day of the year for migration patterns or human population density for exploring regional effects on the association between bird occurrence and human population density. Scolopax computes the corresponding set of summaries and clusters them based on a similarity function, cluster algorithm, and clustering parameters also selected by the user. These clusters are visualized on the map together with the average cluster trajectories (see Figure 5.8). The user can also examine the individual summaries generated by hovering over the corresponding area on the map. If the user suspects an interesting migration pattern on the map, she can further explore the area by zooming in and re-partitioning the geographic space.
5.5. DISCOVERY OF INTERESTING PATTERNS USING SCOLOPAX

Figure 5.7: The Scolopax correlation finder output screen allows the user to interactively examine the ranked list of summary pairs, see the corresponding region on the map, filter results for further investigation, and resubmit a refined query.

Figure 5.8: The Scolopax cluster ranker output screen allows the user to interactively examine the cluster of summaries with the average cluster trajectories, and each individual summary in a cluster.
CHAPTER 6

Related Work

6.1 Join Processing

Graefe [35] classified parallel join algorithms in two categories: symmetric partitioning and fragment-and-replicate. Fragment-and-replicate join algorithm (FR) partitions one relation to multiple sites and broadcasts the other relation to all sites [32]. One major drawback of FR is the total communication cost depending on the number of sites and the replicated relation size [74]. Note that FR is a special case of 1-Bucket-Random when the join matrix is partitioned into only horizontal or vertical stripes.

Join algorithms using the symmetric partitioning method partition both inputs and perform a local join at each site. DeWitt et al. [23] extended grace hash join [47] and hybrid hash join [25] algorithms to a multiprocessor architecture. Both algorithms are shown to provide linear increases in throughput with corresponding increases in processor and disk resources. Similarly in a shared-nothing multiprocessor environment, grace hash join and hybrid hash join are shown to be superior for equi-joins for uniform input sets [71]. Most previous work in the MapReduce context [84, 13, 7, 63, 67] also adopted hash based partitioning approach to implement equi-joins as described in Section 2.2.

The major drawback of these algorithms is that they are vulnerable to skew [57, 69, 52, 83]. Data skew was shown to cause both poor query cost estimates and sub-linear speedup [24]. Redistribution skew (RS) occurs when the distribution of
records received by nodes are imbalanced [78]. These algorithms are also vulnerable to join product skew (JPS) where the join selectivity at each node differs leading to imbalanced join output at each node [78]. Previous load balancing efforts in shared-nothing architectures for join processing have focused on handling RS in the build relation [27, 46, 40]. DeWitt et al. [27] developed skew-handling parallel equi-join algorithms and concluded that hybrid hash join is the winner in lower skew or no skew cases. For high skew cases, virtual processor range partitioning algorithm is the winner. This algorithm performs range partitioning on the building relation where the number of partitions is a multiple of the number of processors. The algorithm is successful in handling RS but may suffer from JPS [73].

Research by Poosala and Ioannidis showed that attribute value skew in the probe relation and JPS can have significant impact on the performance of parallel join execution [69]. Previous research also suggests composite hash functions in order to collect more information about data distributions which is used by a scheduler to minimize imbalance [83, 81]. Again, JPS is very difficult to be detected without very detailed statistics [73]. Moreover, join attribute value skew cannot be eliminated using hashing since the same join attribute values are always grouped together.

Large scale data analysis platforms try to address skew in computation times through speculative execution for tasks which are expected to dominate end-to-end latency [22, 41]. This approach does not handle data skew in joins, because the excessively large tasks would just be executed on another machine, but not broken up. Kwon et al. [51] attempt to minimize computational skew in scientific analysis tasks with blackbox functions. We demonstrated that we can handle join attribute value skew, RS and JPS not only for equi-joins, but any arbitrary join conditions.

Hash-based join algorithms cannot be used to execute non-equijoin operations. DeWitt et al. [26] studied parallel execution of band joins. The proposed algorithm performs range partitioning on one of the relations and the second relation is distributed to all overlapping partitions. Therefore, RS on both relations and JPS cannot be handled effectively by this algorithm. It is also not clear how this algo-
rithm can be generalized to other non-equijoin operations. Symmetric Fragment Replicate Join (SFR) algorithm partitions both relations symmetrically and replicate them among sites so that all record pairs of the input relations are compared in one of the sites [74]. The replication rate of input sets are decided using heuristics to minimize total communication cost. In order to handle RS, tuples are sent to each site in a round-robin fashion. JPS is not studied which may dominate the job completion time as shown in this dissertation. Partitioning found by the proposed heuristics can be integrated into our approach and used in MapReduce.

Map-Reduce-Merge [84] extends the MapReduce model in order to be able to support other join types and different implementations. Our approach does not require any change to the MapReduce model, but still supports any theta-join in a single MapReduce job. Hence it is possible to integrate our approach with high-level programming languages on top of MapReduce [64, 2, 68].

Afrati et al. [7] studied multiway equi-join processing in MapReduce, optimizing for throughput by selecting a query plan with the lowest input replication cost in the Map phase. The reduce tasks are generated using join attribute value, similar to repartition join, which does not generalize to other join conditions. Vernica et al. [77] present an in-depth study of a special type of similarity join in MapReduce. Some clever techniques for dealing with memory limits are proposed. To the best of our knowledge, our work is the first to study all arbitrary joins and explore optimality properties for them in MapReduce-based systems.

## 6.2 Query Optimization using MapReduce

Multi-query optimization [72] aims at reducing query cost by sharing data and computation. It has been studied extensively for databases [19, 33, 39, 89].

Data sharing was demonstrated to be beneficial also for MapReduce workloads. Multiple jobs that share the same input are merged into a single job in Pig to avoid redundant I/O [34]. Hive [76] follows a similar approach in order to share input among multiple jobs. Agrawal et al. [8] studied how to schedule scans of large
data files when there are many simultaneous requests to a common set of files. The goal is to schedule pending jobs to execute shareable jobs as aggressively as possible without making individual jobs suffer from starvation. CoScan [79] is another scheduling framework in order to merge multiple jobs working on the same datasets while trying to meet individual job deadlines. Wolf et al. [82] proposes an alternative scheduler where jobs are decomposed into sub-jobs and cyclic piggybacking is performed instead of batching for sharing scans. Prior work also focused on improving MapReduce by sharing data and computation among iterative tasks. Haloop [17] provides a MapReduce based framework that improves execution of iterative MapReduce jobs by supporting data re-use. Restore [30] also supports data re-use among MapReduce jobs in a workflow to speed up future workflows executed in the system. Our work is complementary to any of these data sharing approaches. In fact, they create additional opportunities for Anti-Combining because shared data has to be transmitted for multiple queries.

MRShare [60] proposes cost-based optimization for data sharing in MapReduce. In addition to input sharing, map output sharing is also studied. However, intermediate data sharing is only limited to overlapping parts of the map output generated from shared input of multiple jobs. Therefore, cases where the Map function produces multiple key/value pairs for a single job cannot be addressed. In addition, our techniques enable sharing of non-overlapping parts of the Map output by pushing the Map operator to the reduce phase. YSmart [54] translates SQL queries into MapReduce jobs and exploits correlations among the operators in the query plan. In addition to input sharing, intermediate data sharing is also studied. However, intermediate data is shared only among operators having the same key. Therefore, this approach also does not solve the problem introduced in our work. Jahani et al. [43] provides a static analysis-style mechanism for automatic detection of selection, projection, and data compression optimizations in MapReduce programs. Proposed data compression optimization aims to work on directly compressed data where applicable. Delta-compression is also used which is only applicable to numeric datasets where sequential data values change slightly.
6.2. QUERY OPTIMIZATION USING MAPREDUCE

Zhou et al. [88] focus on minimizing the number of partitioning operations while executing complex queries. The Scope optimizer aims to improve data shuffling efficiency by making use of several partition techniques to generate efficient query plans [87]. However, none of these approaches exploit data sharing opportunities that is explored by Anti-Combining.
Conclusions and Future Work

Join is one of the fundamental relational operations and it is frequently used to combine information from multiple sources. Joins are also essential for many large-scale data analysis tasks. Moreover, different join conditions need to be supported by large-scale data processing platforms for many applications. In this dissertation, we described how to efficiently process arbitrary join conditions in distributed shared-nothing systems and analyzed the performance of our techniques using MapReduce.

Starting with the goal of minimizing total job completion time, we showed how to define a great variety of join implementations using appropriate join matrix-to-reducer mappings. In order to support arbitrary joins, we first proposed 1-Bucket-Random, a randomized algorithm that can compute any subset of the cross-product, i.e., any theta-join. We showed that the matrix-to-reducer mapping achieved by 1-Bucket-Random is provably close to optimal for any join with significantly larger output size than its input size. For more selective join conditions we showed that even though there might be faster algorithms than 1-Bucket-Random, in practice it might often not be possible to identify these algorithms as usable without performing expensive analysis or without knowing the join result in advance.

For a popular class of joins such as equi-joins, inequality joins and band-joins, we improved the runtime achieved by 1-Bucket-Random using our M-Bucket algorithms. These algorithms achieve improved runtime by exploiting input statistics with a comparably lightweight test, and thus compute selective join conditions
efficiently. Our join model enables us to estimate max-reducer-input and max-reducer-output for each algorithm. An optimizer can apply traditional cost estimation techniques from databases, because the job completion time is determined by the reducer (i.e., single processor) that receives the greatest input and the reducer that generates the greatest output. Local reducer computation is directly amenable to traditional cost analysis involving CPU and I/O cost.

The join algorithms proposed in this dissertation are compatible with any local join implementation provided by the user. For in-memory join algorithms, the join matrix is further partitioned in order to generate a larger number of reduce tasks with smaller max-reducer-input size in order to satisfy memory constraints of nodes in the cluster. One drawback of this approach is the extra data replication which results in larger amount of data transferred between mapper and reducer nodes. In order to address this problem, we proposed Anti-Combining.

Anti-Combining for MapReduce is a novel approach for reducing the amount of data transferred between mappers and reducers. It can be enabled in any given MapReduce program by applying the appropriate (purely syntactic) transformations, but will be most effective for problems where the shuffle-and-sort phase dominates the overall cost. Anti-Combining dynamically and adaptively shifts mapper-side processing to the reducers and is much more lightweight than general compression techniques.

Anti-Combining can be used together with existing database-style optimizations such as sharing of scans and intermediate results. Since it does not need to understand the semantics of a given MapReduce program, it is perfectly suitable for adding dynamic optimizations to statically optimized MapReduce programs generated by compilers for high-level languages such as PigLatin and HiveQL.

Finally, we presented Scolopax, a data exploration tool we developed which enables scientists to create hypothesis search queries through a user-friendly interface. Scolopax combines the novel techniques proposed in this dissertation to achieve high performance and scalability for various queries used by scientists to compute relationships and correlations in high-dimensional scientific data.
The join algorithms proposed in this dissertation achieve impressive performance for a broad class of join problems. A direction for future work is to develop a complete optimizer for selecting the best MapReduce implementation for any given join problem. Recall that 1-Bucket-Random can be improved by exploiting input statistics for selective joins. We achieved this for equi-joins, inequality joins, and band-joins using equi-depth histograms and M-Bucket algorithms. In order to be able to use M-Bucket algorithms for other joins where the join matrix does not have monotonicity property, exploring different types of histograms is another area for future work.

Supporting joins with multiple conditions using M-Bucket algorithms is limited when equi-depth histograms are used. It is possible to generate a join matrix using an equi-depth histogram for each attribute in the join, and choose the one that results in the best matrix-to-reducer mapping. However, an alternative preprocessing strategy may result in a better matrix-to-reducer mapping by exploiting combinations of conditions in the join problem.

Anti-Combining is applied at the end of each map call to eliminate redundant I/O as early as possible in the MapReduce data-flow. Since the output of the map function is not immediately pulled by the reducers, it would be possible to search for encoding opportunities among several map call output as well. Developing an optimizer to find the best encoding strategy is an interesting future work.

Anti-Combining focused on reducing the network transfer cost of MapReduce programs by re-executing some of the mapper work at the reducers. One interesting direction for future work is to explore the use of more comprehensive cost-based optimization to deal with MapReduce programs whose map- and partition-function related costs are a significant fraction of the total cost.

Recall that careful design of a partitioner would increase the impact of Anti-Combining by assigning records with commonalities to the same reduce task. Note that the effectiveness and scalability of parallel programs in general depend on a good partitioner for even distribution of records among machines. Automatically finding a good partitioner remains another interesting area for future work.
Bibliography


BIBLIOGRAPHY


