Abstract

Many data-driven applications perform computations on large volumes of data that do not fit on a single computer. These applications typically must use parallel shared-nothing distributed software systems to perform their computations. This thesis addresses challenges in large-scale distributed data processing with a particular focus on two primary areas: (i) theoretical foundations for understanding the costs of distribution; and (ii) processing large-scale graph data.

The first part of this thesis presents a theoretical framework for the MapReduce system, to analyze the cost of distribution for different problems domains, and for evaluating the “goodness” of different algorithms. We identify a fundamental tradeoff between the parallelism and communication costs of algorithms. We first study the setting when computations are constrained to a single round of MapReduce. In this setting, we capture the cost of distributing a problem by deriving a lower-bound curve on the communication cost of any algorithm that solves the problem for different parallelism levels. We derive lower-bound curves for several problems, and prove that existing or new one-round algorithms solving these problems are optimal, i.e., incur the minimum possible communication cost for different parallelism levels. We then show that by allowing multiple rounds of MapReduce computations, we can solve problems more efficiently than any possible one-round algorithm.

The second part of this thesis addresses challenges in systems for processing large-scale graph data, with the goal of making graph computation more efficient and easier to program and debug. We focus on systems that are modeled after Google’s Pregel framework for large-scale distributed graph processing. We begin by describing an open-source version of Pregel we developed, called GPS (for Graph Processing
System). We then describe new static and dynamic schemes for partitioning graphs across machines, and we present experimental results on the performance effects of different partitioning schemes. Next, we describe a set of algorithmic optimizations that address commonly-appearing inefficiencies in algorithms programmed on Pregel-like systems. Because it can be very difficult to debug programs in Pregel-like systems, we developed a new replay-style debugger called Graft. In addition, we defined and implemented a set of high-level parallelizable graph primitives, called HelP (for High-level Primitives), as an alternative to programming graph algorithms using the low-level vertex-centric functions of existing systems. HelP primitives capture several commonly appearing operations in large-scale graph computations. We motivate and describe Graft and HelP using real-world applications and algorithms.
Acknowledgments

I want to start by thanking Prof. Jennifer Widom, for being an excellent PhD advisor. I realize that it takes a lot of effort to take a student with no background in research and prepare him to teach and advise his own students in other institutions. My approach to research, my writing, speaking, and presentation skills all bear deep marks from her. I am very grateful for the unique role she has played in my education.

Prof. Jeff Ullman, along with Prof. Foto Afrati, initiated and advised the theory part of this thesis. Some of the most memorable moments of my PhD were the times I would try to solve a problem together with Jeff in front of his whiteboard. I want to thank him for giving me the opportunity to work with him. It was a true privilege for me.

Prof. Christopher Ré has advised me in our work on multiround join algorithms. Since he joined the Stanford faculty, he has opened my eyes to amazing problems in databases, from generalized hypertree decompositions to worst-case and beyond worst-case join algorithms. His research will deeply influence the research that I will do after I leave Stanford. I also want to thank him for spreading my work to other database researchers in every opportunity that he found.

I want to thank Prof. Hector Garcia-Molina for making life in InfoLab a lot of fun with his photography and movie nights. I also want to thank him for being on my orals and quals committees, and helping me with the conference presentations I gave throughout my PhD.

I have collaborated very closely with Prof. Foto Afrati on the theory part of my thesis. Foto has been a great mentor and I hope to continue our collaborations in the years to come.
I want to thank five other professors: Prof. Ram Rajagopal for chairing my defense; Prof. Gio Wiederhold for continuing the tradition and being with us every Friday in InfoLab lunches; and Prof. Ashish Goel, Tim Roughgarden, and Luca Trevisan for inspiring me to learn more theory in the first few years of my PhD.

I thank Marianne Siroker and Jam Kiattinant for helping me with countless bureaucratic troubles I had throughout my six years at Stanford. Their presence at Gates brings cheer to the CS department. I also want to acknowledge Andrej Krevl for helping me solve numerous systems issues in my projects.

My research was done in collaboration with many exceptional colleagues: Anish Das Sarma, Manas Joglekar, Jaeho Shin, Sungpack Hong, Robert Ikeda, Vikesh Khanna, Ba Qu Truang, Prof. Kunle Olukotun, Firas Abouzaid, and Anand Rajaraman. I thank all of them for all their contributions to this thesis.

Most importantly, I want to thank my parents for encouraging me to go to the United States to study and enduring my absence for so long. They, more than any one else, have shaped me as a person, and I hope I can follow their steps in life. Finally, I want to thank Basak, my sister and brother and their families, and my friends for their love and support.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstract</strong></td>
<td>iv</td>
</tr>
<tr>
<td><strong>Acknowledgments</strong></td>
<td>vi</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>1.1 MapReduce Overview</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Theoretical Foundations For Distributed Data-Processing Systems</td>
<td>5</td>
</tr>
<tr>
<td>1.2.1 Outline of Thesis Chapters on Theoretical Foundations</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Pregel Overview</td>
<td>9</td>
</tr>
<tr>
<td>1.4 Distributed Graph Processing Systems</td>
<td>11</td>
</tr>
<tr>
<td>1.4.1 Efficiency</td>
<td>12</td>
</tr>
<tr>
<td>1.4.2 Ease of Programming and Debugging</td>
<td>13</td>
</tr>
<tr>
<td>1.4.3 Outline of Thesis Chapters on Distributed Graph Processing Systems</td>
<td>16</td>
</tr>
<tr>
<td><strong>2 A Theoretical Framework For MapReduce</strong></td>
<td></td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>17</td>
</tr>
<tr>
<td>2.1.1 Reducer Size As a Measure of Parallelism Level</td>
<td>18</td>
</tr>
<tr>
<td>2.1.2 An Example of the Tradeoff: All-to-All Problem (A2A)</td>
<td>19</td>
</tr>
<tr>
<td>2.1.3 How the Tradeoff Can Be Used</td>
<td>21</td>
</tr>
<tr>
<td>2.1.4 Outline of the Chapter</td>
<td>22</td>
</tr>
<tr>
<td>2.2 The Framework</td>
<td>22</td>
</tr>
<tr>
<td>2.2.1 Problem Representation: Bipartite Input-Output Graphs</td>
<td>22</td>
</tr>
<tr>
<td>2.2.2 Algorithm Representation: Mapping Schemas</td>
<td>24</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Cost Model</td>
</tr>
<tr>
<td>2.2.4</td>
<td>A Proof Template for Deriving Tradeoff Curves</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Criteria for Good MR Algorithms</td>
</tr>
<tr>
<td>2.2.6</td>
<td>Summary of Our Results</td>
</tr>
<tr>
<td>2.3</td>
<td>Dense Matrix Multiplication</td>
</tr>
<tr>
<td>2.3.1</td>
<td>The Tradeoff Curve</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Algorithms</td>
</tr>
<tr>
<td>2.4</td>
<td>Hamming Distance 1</td>
</tr>
<tr>
<td>2.4.1</td>
<td>The Tradeoff Curve</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Algorithms</td>
</tr>
<tr>
<td>2.5</td>
<td>Multiway Equijoins</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Fractional Edge Coverings</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Vertex Packing and Max-Out Distribution</td>
</tr>
<tr>
<td>2.5.3</td>
<td>The Tradeoff Curve</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Algorithms</td>
</tr>
<tr>
<td>2.6</td>
<td>Related Work</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Models of MapReduce</td>
</tr>
<tr>
<td>2.6.2</td>
<td>MapReduce Algorithms</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Other Related Work</td>
</tr>
<tr>
<td>3</td>
<td>Fuzzy Joins With Hamming and Edit Distance d &gt; 1</td>
</tr>
<tr>
<td>3.1</td>
<td>Problem Definition</td>
</tr>
<tr>
<td>3.2</td>
<td>The Naive Algorithm</td>
</tr>
<tr>
<td>3.3</td>
<td>Generalized Anchor Points Algorithm and Covering Codes</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Cost Analysis</td>
</tr>
<tr>
<td>3.4</td>
<td>Constructing Explicit Codes By Cross Product</td>
</tr>
<tr>
<td>3.5</td>
<td>Edit-Distance Covering Codes</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Elementary Lower Bounds</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Summary of Results</td>
</tr>
<tr>
<td>3.6</td>
<td>Explicit Construction of Edit-Distance Covering Codes</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Insertion-1 Covering Codes</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>3.6.2</td>
<td>O(1/a^2)-size Deletion-1 Covering Codes</td>
</tr>
<tr>
<td>3.6.3</td>
<td>O(1/a^3)-size Deletion-2 Covering Code For Shorter Strings</td>
</tr>
<tr>
<td>3.7</td>
<td>Existence of O(log(n)/n)-size Deletion-1 Covering Codes</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Step 1: Run Patterns</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Step 2: Converting Run Patterns into Binary Strings</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Step 3: Partitioning Strings Based on Safe Bit Counts</td>
</tr>
<tr>
<td>3.7.4</td>
<td>Step 4: Existence of a Deletion-1 Code Covering (1-1/n) Fraction of HS</td>
</tr>
<tr>
<td>3.7.5</td>
<td>Proof of Lemmas</td>
</tr>
<tr>
<td>3.7.6</td>
<td>Steps 5 and 6: Existence of O(log(n)/n)-size Deletion-1 Codes</td>
</tr>
<tr>
<td>3.8</td>
<td>Related Work</td>
</tr>
<tr>
<td>4</td>
<td>Multiround MapReduce</td>
</tr>
<tr>
<td>4.1</td>
<td>Dense Matrix Multiplication</td>
</tr>
<tr>
<td>4.2</td>
<td>Multiway Equijoins</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Summary of GYM and Main Results</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Outline for the Rest of the Chapter</td>
</tr>
<tr>
<td>4.3</td>
<td>Generalized Hypertree Decompositions</td>
</tr>
<tr>
<td>4.4</td>
<td>Distributed Yannakakis</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Serial Yannakakis Algorithm</td>
</tr>
<tr>
<td>4.4.2</td>
<td>DYM-n</td>
</tr>
<tr>
<td>4.4.3</td>
<td>DYM-d</td>
</tr>
<tr>
<td>4.5</td>
<td>GYM</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Overview of GYM</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Analysis of GYM</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Example Execution of GYM</td>
</tr>
<tr>
<td>4.6</td>
<td>Constructing O(log(n))-depth GHGs</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Extending D'</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Unique-Child-And-Grandchild Vertices</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Two Transformation Operations</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Log-GTA</td>
</tr>
</tbody>
</table>
6.2.5 Weakly Connected Components (WCC) .......................... 167
6.3 Finishing Computations Serially (FCS) ............................. 167
  6.3.1 High-level Description ........................................ 168
  6.3.2 Implementation .............................................. 169
  6.3.3 Cost Analysis ............................................... 169
  6.3.4 FCS for Backward-Traversal Phase of SCC (FCS-BT) ........ 170
  6.3.5 FCS for SCC (FCS-SCC) ..................................... 172
  6.3.6 FCS for GC (FCS-GC) ....................................... 173
6.4 Storing Edges At Subvertices (SEAS) in MSF ....................... 174
  6.4.1 High-level Description ....................................... 174
  6.4.2 Implementation ............................................. 174
  6.4.3 Cost Analysis ............................................... 175
  6.4.4 Experiments .................................................. 175
6.5 Edge Cleaning On Demand ............................................. 176
  6.5.1 High-level Description ....................................... 176
  6.5.2 Implementation for MWM ..................................... 177
  6.5.3 Cost Analysis for MWM ....................................... 178
  6.5.4 Implementation for MSF ..................................... 179
  6.5.5 Cost Analysis for MSF ....................................... 179
  6.5.6 Experiments .................................................. 181
6.6 Single Pivot Optimization (SP) ....................................... 182
  6.6.1 High-level Description and Implementation .................... 182
  6.6.2 Cost Analysis ............................................... 183
  6.6.3 Experiments .................................................. 183
6.7 Related Work .......................................................... 184

7 Debugging Graph Algorithms For Pregel-like Systems ............. 187
  7.1 The Graft Debugging Tool ....................................... 189
    7.1.1 Capture: The DebugConfig File and Graft Instrumenter .... 189
    7.1.2 Visualize: Graft GUI ....................................... 191
    7.1.3 Reproduce: Context Reproducer ............................ 194
# List of Tables

2.1 Inputs and outputs of the A2A problem. .......................... 20  
2.2 Summary of one-round MR results. ............................... 29  
3.1 Hamming distance covering codes. ............................... 53  
3.2 Summary of edit-distance covering codes. ...................... 59  
4.1 Example queries and their edge covering and edge packing numbers. 87  
5.1 Data sets. ................................................. 135  
6.1 Algorithms. ............................................... 154  
6.2 Optimization Techniques. ...................................... 154  
6.3 Graph datasets. ............................................. 157  
7.1 Graph datasets for demonstration. .............................. 197  
7.2 Graph datasets for performance experiments. .................. 202  
7.3 DebugConfig configurations. .................................. 202  
8.1 HelP Primitives. ............................................ 210  
8.2 Implemented Algorithms. .................................... 212  
8.3 Inputs to aggregateNeighborValues. ............................ 219  
8.4 Inputs to propagateAndAggregate. ............................. 220  
8.5 Inputs to updateSelfUsingOneOtherVertex. .................... 225  
8.6 Inputs to formSupervertices. .................................. 232
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>MapReduce Overview</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>WordCount in MapReduce</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Tradeoff curves</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Algorithms</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>Pregel Overview</td>
<td>10</td>
</tr>
<tr>
<td>1.6</td>
<td>PageRank in Pregel</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>Bipartite input output graphs for representing problems</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>Tradeoff curve and algorithms for the A2A problem</td>
<td>28</td>
</tr>
<tr>
<td>2.3</td>
<td>Input/output relationship for the matrix-multiplication problem</td>
<td>30</td>
</tr>
<tr>
<td>2.4</td>
<td>Tradeoff curve and algorithms for matrix multiplication</td>
<td>31</td>
</tr>
<tr>
<td>2.5</td>
<td>Input/output relationship for HD-1</td>
<td>33</td>
</tr>
<tr>
<td>2.6</td>
<td>Tradeoff curve and algorithms for HD-1</td>
<td>36</td>
</tr>
<tr>
<td>2.7</td>
<td>Fractional edge coverings and vertex packings of queries</td>
<td>39</td>
</tr>
<tr>
<td>2.8</td>
<td>Example queries and their edge covering numbers</td>
<td>40</td>
</tr>
<tr>
<td>3.1</td>
<td>Algorithms for Hamming distance $d &gt; 1$</td>
<td>50</td>
</tr>
<tr>
<td>3.2</td>
<td>Input/output relationship for ED-2</td>
<td>52</td>
</tr>
<tr>
<td>3.3</td>
<td>Simulation of insertions of symbols $0$ and $(a-1)$ into strings</td>
<td>62</td>
</tr>
<tr>
<td>3.4</td>
<td>Pairs that sum to $i$ modulo 6</td>
<td>66</td>
</tr>
<tr>
<td>4.1</td>
<td>Performance of GYM on Chain-n with different GHGs of Chain-n</td>
<td>82</td>
</tr>
<tr>
<td>4.2</td>
<td>The two-round Partial-sums algorithm</td>
<td>83</td>
</tr>
<tr>
<td>4.3</td>
<td>Matrix multiplication algorithms and the one-round lower bound</td>
<td>85</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>4.4</td>
<td>Fractional edge packing</td>
<td>87</td>
</tr>
<tr>
<td>4.5</td>
<td>The hypergraph and the GHD for the join in Example 4.3.1</td>
<td>93</td>
</tr>
<tr>
<td>4.6</td>
<td>Acyclic queries with different depth GHDs</td>
<td>101</td>
</tr>
<tr>
<td>4.7</td>
<td>GYM</td>
<td>102</td>
</tr>
<tr>
<td>4.8</td>
<td>Width-3 GHD of Chain-16</td>
<td>105</td>
</tr>
<tr>
<td>4.9</td>
<td>After Computing IDBs of the GHD of Figure 4.8</td>
<td>105</td>
</tr>
<tr>
<td>4.10</td>
<td>Width-1 GHD of Chain-16</td>
<td>105</td>
</tr>
<tr>
<td>4.11</td>
<td>Unique-child-and-grandchild vertices</td>
<td>108</td>
</tr>
<tr>
<td>4.12</td>
<td>Leaf Inactivation</td>
<td>110</td>
</tr>
<tr>
<td>4.13</td>
<td>Unique-c-gc inactivation</td>
<td>112</td>
</tr>
<tr>
<td>4.14</td>
<td>Log-GTA</td>
<td>114</td>
</tr>
<tr>
<td>4.15</td>
<td>Log-GTA simulation</td>
<td>115</td>
</tr>
<tr>
<td>4.16</td>
<td>Performance of GYM on Chain-n with different GHDs of Chain-n</td>
<td>119</td>
</tr>
<tr>
<td>5.1</td>
<td>GPS Architecture</td>
<td>129</td>
</tr>
<tr>
<td>5.2</td>
<td>Connected components in GPS</td>
<td>131</td>
</tr>
<tr>
<td>5.3</td>
<td>A simple k-means like graph clustering algorithm</td>
<td>132</td>
</tr>
<tr>
<td>5.4</td>
<td>Counting the number of crossing edges with vertex.compute()</td>
<td>133</td>
</tr>
<tr>
<td>5.5</td>
<td>Clustering algorithm using master.compute()</td>
<td>134</td>
</tr>
<tr>
<td>5.6</td>
<td>Network I/O, different partitioning schemes</td>
<td>138</td>
</tr>
<tr>
<td>5.7</td>
<td>Run-time, different partitioning schemes</td>
<td>140</td>
</tr>
<tr>
<td>5.8</td>
<td>Slowest worker, number of messages</td>
<td>141</td>
</tr>
<tr>
<td>5.9</td>
<td>Fixing workload imbalance of METIS-default</td>
<td>142</td>
</tr>
<tr>
<td>5.10</td>
<td>Performance of LALP</td>
<td>143</td>
</tr>
<tr>
<td>5.11</td>
<td>Performance of PageRank with and without dynamic repartitioning</td>
<td>147</td>
</tr>
<tr>
<td>6.1</td>
<td>SCCMaster's code for switching phases</td>
<td>159</td>
</tr>
<tr>
<td>6.2</td>
<td>Skeleton code for SCCVertex.compute()</td>
<td>159</td>
</tr>
<tr>
<td>6.3</td>
<td>Original Coloring algorithm for computing SCCs</td>
<td>160</td>
</tr>
<tr>
<td>6.4</td>
<td>SCCVertex subroutines for the Forward-Traversal phase</td>
<td>162</td>
</tr>
<tr>
<td>6.5</td>
<td>Parallel version of Boruvka's MSF algorithm</td>
<td>162</td>
</tr>
<tr>
<td>6.6</td>
<td>Example of a conjoined-tree</td>
<td>163</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Performing complex analyses on large-scale data is becoming one of the core challenges of many application domains such as web search, social networks, and genetic analysis. As the volume of data grows, and the speed with which new data is generated increases, these applications need to perform their computations on large computer clusters using highly-parallel and shared-nothing distributed software systems. Starting with the advent of Google’s MapReduce system [36] and its open-source version Hadoop [9], the last decade has seen a proliferation of new distributed systems for processing large-scale data. Examples include distributed dataflow systems [67, 98], new distributed relational database systems [14, 121], graph processing systems [8, 87], machine learning systems [50, 90], and others.

This thesis makes contributions in two primary areas:

• The first part of the thesis (Chapters 2–4) starts by providing a theoretical foundation for large-scale distributed systems, with the goal of being able to answer two questions: (i) How costly is it to parallelize different problems? (ii) How should we measure the “goodness” of existing algorithms for a given problem? We present a theoretical framework to answer these two questions rigorously within the context of the MapReduce system. Using our framework, we answer these two questions for several problems, including fuzzy and equi-joins of record-oriented data, and multiplying two dense matrices, and we present new algorithms that have provable guarantees.
The second part of the thesis (Chapters 5–8) studies systems for processing large-scale graph data, such as social networks and the web graph. We describe an open-source platform for scalable graph processing, algorithms for distributing graphs across machines, and extensions to existing graph processing systems with the goal of making them more efficient and easier to program. The material in these chapters mainly focuses on systems that are modeled after Google’s proprietary Pregel system [87] for processing large graphs.

In the remainder of this introductory chapter, we first give an overview of MapReduce (Section 1.1) and provide a summary of our contributions on theoretical foundations for large-scale data-processing systems (Section 1.2). We then give an overview of Pregel (Section 1.3) and provide a summary of our contributions on distributed graph processing systems (Section 1.4). Related work on the topics of this thesis is discussed at the end of each chapter.

1.1 MapReduce Overview

MapReduce [36] was introduced by Google in 2004 to support several of Google’s data-intensive applications that process data in volumes too large for a single machine, such as indexing and sorting of web pages, and clustering of Google News articles and Google Shopping products. Soon after in 2005, the Hadoop system [9] was developed as an open-source version of MapReduce and saw wide adoption amongst enterprises and researchers. Figure 1.1 shows a high-level overview of MapReduce. The system runs on a cluster of machines that are grouped logically into mapper machines and reducer machines. For simplicity, we will refer to these machines as mappers and reducers, respectively. The input data, represented as a set of records \( r_1, ..., r_m \) in the figure, is initially partitioned across the mapper machines. The programmer expresses the desired computation by implementing two functions: a \texttt{map()} and a \texttt{reduce()} function. The system performs the computation in two stages:

- **Map Stage:** The \texttt{map()} function is applied to each input record in parallel and generates a set of intermediate \(<\text{key}, \text{value}>\) pairs. We refer to the intermediate
Figure 1.1: MapReduce Overview.

keys generated by the map() function as reduce-keys. The system sends all pairs with the same reduce-key to the same reducer and groups their values. For simplicity, we assume that each reducer gets only one reduce-key.

- **Reduce Stage:** Each reducer \( r \) applies the reduce() function to the reduce-key and bucket of values that were sent to \( r \), and produces some outputs.

In the most general setting, a MapReduce algorithm can consist of multiple rounds of map and reduce stages, each executing possibly different map() and reduce() functions, and using the output of the previous round as its input.

**Example 1.1.1.** We illustrate the classic WordCount problem in a one-round MapReduce algorithm. We are given as input a corpus of words. The goal is to find the number of times each word occurs in the corpus. Figure 1.2 shows the map() and
// Maps each word \( w \) to \( \langle w, 1 \rangle \).

```java
Pair<String, Integer> map(String word) { output(new Pair(word, 1); }
```

// Outputs for each \( \langle w, \text{bucket-of-ones} \rangle \) to \( \langle w, \text{bucket-of-ones.size} \rangle \).

```java
Pair<String, Integer> reduce(String word, List<Integer> values) {
    output(new Pair(word, values.size); }
```

Figure 1.2: WordCount in MapReduce.

\( \text{reduce}() \) functions that solve the WordCount problem. The \( \text{map}() \) function simply takes a word \( w \) and produces the key-value pair \( \langle w, 1 \rangle \). After grouping by reduce-key, each word \( w \) has a bucket of values containing as many 1s as \( w \) appears in the input corpus. The \( \text{reduce}() \) function simply counts the number of 1s in each word \( w \)’s bucket.

Three properties of MapReduce have made it very attractive to programmers as a system for processing large-scale data:

- **Transparent Parallelism:** The system automatically parallelizes the work to mapper and reducer machines in the cluster, and guarantees that the intermediate key-value pairs are routed correctly to reducer machines. The programmer does not write any code to parallelize the computation.

- **Transparent Fault Tolerance:** The system automatically recovers if a mapper or reducer machine fails during the computation. MapReduce’s fault recovery is based on simple checkpointing of intermediate outputs to disk and rescheduling the tasks of the failed machines to healthy machines.

- **Simple Programming API:** The \( \text{map}() \) and \( \text{reduce}() \) functions focus the user’s attention on the computational problem, rather than how the computation will be executed on the actual machines of the cluster. The MapReduce API has also proved to be capable of expressing numerous data-processing tasks across many application domains, including web search [36], graph algorithms [36], natural language processing [83], biology [89], and many others.
These three fundamental properties of MapReduce have been a guideline for many systems that were built subsequently. Most of these systems [8, 14, 27, 45, 50, 134, 135] have adopted MapReduce’s transparent fault-tolerance and parallelism. Many of them [8, 27, 50, 87, 135] also have MapReduce-like APIs that are tailored for specific application domains. For example, as we will describe in Section 1.3, similar to MapReduce’s map() and reduce() functions, Pregel’s API is based on implementing a vertex.compute() function that is tailored for graph computations.

1.2 Theoretical Foundations For Distributed Data-Processing Systems

A routine decision that programmers of distributed data-processing systems need to make is to pick a parallelism level for their computations. We will make the notion of “parallelism level” concrete in Chapter 2. For the purpose of this introductory chapter, parallelism level intuitively corresponds to the number of machines that are used in the computation. For example, a MapReduce algorithm could map each input record of a problem $P$ to the same reduce-key, thus effectively solving $P$ serially inside a single machine. In contrast, another algorithm could use thousands of reduce-keys, solving problem $P$ using thousands of machines in parallel. One natural question that arises when picking an appropriate parallelism level for solving a problem is: What is the cost of parallelizing a computation to more machines? More specifically, the two central questions motivating the first part of this thesis are:

1. How costly is it to parallelize different problems?
2. How should we measure the “goodness” of different parallel algorithms solving a particular problem?

In general, the cost of parallelizing problems is increased communication between the machines in the cluster. In MapReduce, the communication corresponds to the amount of data that is sent from the mappers to the reducers. Thus, there is an inherent tradeoff between the parallelism and communication costs of algorithms when
solving problems in MapReduce. Driven by this observation, we present a theoretical framework to answer our two questions in the limited setting when computations are constrained to a single round of MapReduce.

1. We represent the cost of parallelizing a problem $P$ by deriving a tradeoff curve between parallelism and communication for solving $P$. Specifically, this tradeoff curve is a lower-bound curve that shows the minimum amount of communication that is needed to solve $P$ by any algorithm for different parallelism levels. Figure 1.3 shows an abstract example of what these tradeoff curves might look like for two hypothetical problems 1 and 2, shown by the red and orange curves, respectively. The X-axis is the parallelism level, decreasing to the right, and the Y-axis is the communication cost. For both problems, as we increase the parallelism of algorithms, i.e., move left on the X-axis, the communication cost increases. In this hypothetical example, problem 2 is more costly to parallelize than problem 1 because problem 2’s curve is higher than problem 1’s. In other words, for each parallelism level, algorithms need to incur more communication cost to solve problem 2 than problem 1.
Similarly, we measure how “good” an algorithm is for solving a problem by calculating its parallelism level and communication cost. We depict the performance of algorithms as points on the same parallelism level versus communication cost graphs. For example, in Figure 1.4 a hypothetical algorithm $A$ for problem 1 is depicted as a single rectangular point. If an algorithm is parameterized and can run at multiple different parallelism levels, we depict its performance as a scatter plot. For example, algorithm $B$, which can run at multiple parallelism levels, is depicted as a scatter plot of round points in the figure. Ideally we want to design algorithms whose communication costs match the lower-bound tradeoff curve exactly, such as algorithm $C$, shown as the triangular point in Figure 1.4. We call algorithm $C$ “Pareto optimal” because no other algorithm can incur less communication cost than $C$ for the parallelism level at which $C$ executes. In this framework, the ultimate goal of algorithm designers is to have a parameterized algorithm that is Pareto optimal at every parallelism level.

Using our framework, we derive tradeoff curves and analyze the performance of algorithms that execute in a single round of MapReduce for several problems, including dense matrix multiplication, fuzzy-joins of a set of strings, and equi-joins of multiple
relational tables. For some of the problems we either show that existing algorithms are Pareto optimal, or we describe new algorithms that are Pareto optimal or better than existing ones. We also show that by running multiple rounds of map and reduce stages, we can solve problems more efficiently than in a single round.

1.2.1 Outline of Thesis Chapters on Theoretical Foundations

The thesis chapters covering our work on theoretical foundations of distributed data-processing systems are organized as follows:

- Chapter 2 describes our theoretical framework. We present formalisms to represent computational problems and one-round MapReduce algorithms, and a proof template for deriving tradeoff curves between communication and parallelism for different problems. We then use our framework to analyze three different problems, and algorithms for these problems: (i) dense matrix multiplication; (ii) fuzzy-joins of bit strings with Hamming distance 1, i.e., finding pairs of bit strings from a corpus that are at Hamming distance 1; and (iii) equi-joins of multiple relational tables.

- Chapter 3 presents a one-round MapReduce algorithm called the Generalized Anchor Points (GAP) algorithm for computing fuzzy-joins with Hamming and edit distance greater than 1. Although we cannot derive a tradeoff curve for solving fuzzy-joins with Hamming and edit distance greater than 1, we show that the GAP algorithm is provably better than existing algorithms. The GAP algorithm relies on finding a set of strings called a covering code for a corpus of strings. The main technical contribution of this chapter is the construction of the first covering codes for the edit-distance problem.

- Chapter 4 studies multiround MapReduce algorithms. We show that by running a sequence of MapReduce jobs, we can solve problems more efficiently than any possible one-round algorithm. We give examples from the problems of dense matrix multiplication and equi-joins of multiple relational tables. The main technical contribution of this chapter is an algorithm called GYM, for
Generalized Yannakakis in MapReduce, for the problem of equi-joins of multiple relational tables.

1.3 Pregel Overview

The second part of this thesis studies distributed systems for processing large-scale graphs: graphs that contain up to billions of vertices and edges, such as the web graph and social networks. Soon after the advent of MapReduce and Hadoop, several systems and libraries were built on top of Hadoop [27, 74, 135] specifically for processing large-scale graphs. However, the MapReduce setting was not found suitable for processing graphs, for two reasons:

1. Many graph algorithms, such as PageRank [25], iteratively compute a value for each vertex of the graph. In MapReduce, these algorithms need to be executed in multiple rounds by running a separate MapReduce job for each iteration of the algorithm. The inputs of each MapReduce job are the entire graph data and the output of the previous job; the output is the latest values of vertices. MapReduce implementations store data in a distributed fashion, such as in network file systems or distributed databases. Reading the inputs from and writing the output to distributed storage in between each MapReduce job incurs very high I/O costs.

2. Many graph algorithms are naturally expressed as vertices communicating with each other. Expressing these algorithms in MapReduce is difficult and yields clumsy `map()` and `reduce()` functions to simulate vertex-to-vertex communications.

Google’s Pregel system addresses these two shortcomings while maintaining the three fundamental properties of MapReduce (Section 1.1). Similar to MapReduce, Pregel is transparently parallel, fault-tolerant, and has an API based on implementing simple user-specified functions. Unlike MapReduce, Pregel is inherently iterative, and it has a graph-specific API that is based on passing messages between vertices.
Figure 1.5 shows the high-level overview of the system. The computational framework of Pregel is based on Valiant’s *Bulk Synchronous Parallel (BSP)* model [123]. The input is a directed graph. Initially the vertices of the graph are distributed across the machines in the cluster. Computation consists of iterations called *supersteps*. In each superstep, a user-specified `vertex.compute()` function is applied to each vertex in parallel. Inside `vertex.compute()`, each vertex reads its incoming messages, updates its value, sends messages to other vertices to be used in the next superstep, and sets a flag indicating whether this vertex is ready to stop computation. At the end of each superstep, all machines synchronize before starting the next superstep. The computation terminates when all vertices are ready to stop the computation.

**Example 1.3.1.** Consider computing the PageRank of vertices of a web graph. In PageRank, each vertex is assigned an initial PageRank value. In each iteration, the PageRank value of each vertex is updated by aggregating the PageRank values of its incoming neighbors. Figure 1.6 shows the `vertex.compute()` code implementing the PageRank algorithm that executes for 50 iterations. In the first superstep, each vertex sends its initial PageRank value to its outgoing neighbors as a message. In later
public void compute(Iterable<Msg> msgs) {
    if (getSuperstepNo() == 1) {
        updateValue(0.1);
    } else if (getSuperstepNo() < 50) {
        double sum = sumMsgs(msgs);
        double newPR = sum*0.85 + 0.15;
        updateValue(newPR);
        sendMsgToAllNbrs(newPR);
    } else { voteToHalt(); }
}

Figure 1.6: PageRank in Pregel.

supersteps, each vertex first aggregates the messages that it has received to update its own PageRank value, and then sends its new PageRank value to its outgoing neighbors as a message. Computation terminates in superstep 50, when all vertices indicate that they are ready to terminate computation by calling the voteToHalt() function.

Pregel is more efficient than MapReduce for iterative graph computations because the graph and vertex values remain in memory throughout the computation, avoiding large I/O costs. It is also easier to express graph algorithms inside Pregel’s graph-specific vertex.compute() function than the map() and reduce() functions of MapReduce.

1.4 Distributed Graph Processing Systems

Our first contribution is the description an open-source version of Pregel we built from scratch, called GPS, for Graph Processing System. During our initial development of GPS and experience with implementing algorithms on the system, we observed two main problems with the Pregel framework: (1) Algorithms running on Pregel can be inefficient incurring high communication costs or executing for a very large number of supersteps. (2) The vertex.compute() functions implementing some algorithms can be very difficult to program and debug. In response to these shortcomings, we make contributions to both efficiency and ease of programming and debugging. Our contributions have sometimes led to new features that we integrated into our GPS
system, or new systems projects that we built on other distributed graph systems, including another open-source version of Pregel called Apache Giraph [8]. For simplicity of terminology, we refer to systems that are modeled after Pregel, such as GPS and Apache Giraph, as Pregel-like systems.

1.4.1 Efficiency

Our work to address inefficiencies in graph algorithms on Pregel-like systems is in two categories: (1) sophisticated graph partitioning schemes; and (2) specific algorithmic optimizations.

Sophisticated Graph Partitioning Schemes

By default Pregel, GPS, and Apache Giraph distribute the input graph across the compute nodes randomly, or using a round-robin scheme. For many graph algorithms the messages between vertices are sent along the edges of the graph. As a result, under random partitioning, a large fraction of the messages are sent across the network, incurring high network I/O costs. Our first contribution under the theme of efficiency is a study of the graph partitioning question: Can algorithms perform better if we partition the graph using a more sophisticated scheme before the computation starts? Will they perform better if we dynamically repartition the graph during the computation? We have developed two new partitioning schemes, and we have experimented with domain-based partitioning of web pages and the popular METIS graph partitioning algorithm [91]. We demonstrate that schemes that are able to maintain a good workload balance across the machines can improve the performance of algorithms significantly. Our first new partitioning scheme, called LALP, for large adjacency-list partitioning, partitions the edges of high-degree vertices across all compute nodes. Our second new partitioning scheme starts with an initial partitioning of the graph and dynamically repartitions the vertices of the graph to decrease the number of messages sent across the network. Both schemes have been fully implemented and integrated into our GPS system.
Algorithmic Optimizations

Graph algorithms running on Pregel-like systems sometimes incur unnecessary inefficiencies such as slow convergence or high communication or computation costs, typically due to structural properties of the input graphs, such as large diameters or skew in component sizes. For example, in the standard Pregel implementation of breadth-first traversal of a graph, a source vertex sends a message to its neighbors in the first superstep, which is then propagated by the receiving vertices to their neighbors, until all of the vertices receive a message. This computation can take a large number of superstep executions on graphs with large diameters. Can we avoid some of these superstep executions? Our second contribution under the theme of efficiency is a set of algorithmic optimization techniques to address such commonly appearing inefficiencies in algorithms.

1.4.2 Ease of Programming and Debugging

Our initial implementation of a suite of algorithms on GPS revealed that it can be very difficult to program and debug some algorithms that are written inside `vertex.compute()`. This thesis makes three contributions to tackle the challenge of programming and debugging algorithms on Pregel-like systems: (1) an extension to Pregel’s original API called `master.compute()`; (2) a new debugger called *Graft*; and (3) a set of high-level parallelizable graph primitives, called *HelP*, as an alternative to programming algorithms within `compute()` functions.

**Master.compute()**

Implementing a graph computation inside `vertex.compute()` is ideal for algorithms that can be performed in a fully “vertex-centric” and parallel fashion. Examples include the PageRank algorithm from Figure 1.6, finding shortest paths in a graph, or finding connected components. However, many algorithms consist of both vertex-centric (parallel) and global (sequential) computations. Such algorithms cannot be easily and efficiently implemented within `vertex.compute()`. Our first contribution is an extension to the original Pregel API called `master.compute()`
that enables efficient implementation of algorithms composed of one or more vertex-centric computations, combined with sequential computations. The ability to easily express sequential computations inside `master.compute()` has also been critical in our most applicable algorithmic optimization, called *Finishing Computations Serially*. `Master.compute()` has also been adopted by Apache Giraph.

**Graft Debugging Tool**

Debugging programs written for Pregel-like systems is very challenging. A series of interviews we conducted with several Apache Giraph and GPS programmers revealed that programmers typically go through three manual steps in the debugging process: (1) adding print statements to `compute()` functions; (2) examining the printed statements; and (3) inspecting the `compute()` functions to infer the lines of code that must have executed to generate the printed statements. In distributed systems, reasoning about the printed information produced by possibly hundreds of machines can be very tedious and time-consuming. To simplify programmers’ debugging cycles, we developed a new replay-style debugger called Graft, implemented for the Apache Giraph system. Graft’s features are motivated by programmers’ three manual steps, which we refer to as *capture*, *visualize*, and *reproduce*. In Graft:

- **Capture**: Programmers programmatically describe the vertices they are interested in gathering information about. Typically these vertices are expected to be a relatively small number of “suspicious” vertices.

- **Visualize**: Programmers visualize the behavior of the captured vertices in a graph-specific and superstep-based GUI. The GUI allows the programmers to replay how the vertex values and messages are changing across supersteps.

- **Reproduce**: Programmers do line-by-line debugging, using any local debugger, to see the lines of the `vertex.compute()` function that executed for a captured vertex in a specific superstep.

Graft similarly enables debugging of the `master.compute()` function. We built Graft for Apache Giraph because it was the most widely used Pregel-like system at
the time of our development. However, with only minor modifications Graft could be used for other Pregel-like systems, such as GPS.

**HelP Primitives**

The APIs of Pregel-like systems have drawn philosophically from MapReduce. As we discussed earlier, the benefit of these APIs is that programmers concentrate on implementing a few specific functions, and the framework automatically parallelizes the computation across machines. However, this approach has two shortcomings in the Pregel setting. First, the `compute()` function can be too low-level or restricted for certain algorithmic operations, such as grouping vertices together to form “supervertices”—an operation that appears in several algorithms as we will discuss in Chapter 8. Second, custom code is required for commonly appearing operations across algorithms, such as initializing vertex values or checking for convergence of computations. Similar shortcomings of the MapReduce API were addressed by widely-used languages such as *Pig Latin* [98] and *Hive* [121], which express data analysis tasks using higher-level constructs and primitives, such as filters and grouping. Programs written in these languages then compile to the lower-level `map()` and `reduce()` functions of the Hadoop open-source MapReduce system. In this thesis, we take a similar approach for graph processing and introduce a set of six high-level graph primitives, called *HelP* (for *High-Level Primitives*), which capture the commonly appearing operations in distributed graph computations. We will describe the primitives we identified, our implementation of these primitives on the *GraphX* graph processing system [130], and the implementation of a large suite of graph algorithms using these primitives. Our experience has been that implementing algorithms using our primitives is more intuitive and faster than using the APIs of existing distributed systems.
1.4.3 Outline of Thesis Chapters on Distributed Graph Processing Systems

The thesis chapters covering our work on distributed graph processing systems are organized as follows:

- Chapter 5 describes our GPS system. We describe the implementation of GPS, GPS’s `master.compute()` API extension, LALP partitioning, and dynamic repartitioning. We also present extensive experiments on the effects of different ways of partitioning the graph on the performance of algorithms.

- Chapter 6 describes a set of algorithmic optimizations to address some of the commonly appearing inefficiencies in algorithms that execute on Pregel-like systems.

- Chapter 7 describes the Graft debugging tool for the Apache Giraph system.

- Chapter 8 describes our HelP primitives and their implementation on the GraphX system.
Chapter 2

A Theoretical Framework For MapReduce

2.1 Introduction

Two decisions that users of highly-parallel shared-nothing systems routinely make when processing their data is to pick an algorithm and a parallelism level for their computations. We will make the notion of “parallelism level” concrete momentarily, but intuitively it corresponds to the number of machines that are used in the computation. There are two benefits to processing data at increasing parallelism levels. First, each machine usually gets a smaller fraction of the data. So, once the data is partitioned appropriately, each machine does less work and completes its computation in less time. Second, applications can use cheaper machines with less RAM and disk space, as each machine is processing less data. On the other hand, the costs of parallelization are less obvious. In this chapter, we present a theoretical framework for answering the following two questions in a mathematically rigorous way:

1. How costly is it to parallelize different problems in a MapReduce (MR) setting?

2. What are the appropriate metrics to compare the “goodness” of different MR algorithms that solve a particular problem?
The main theme of this chapter is that there is an inherent tradeoff between parallelism and communication cost when solving problems in distributed systems. Specifically, the cost of parallelization in MR is an increased communication cost between mappers and reducers. In light of this observation, we will measure how costly it is to parallelize a problem $P$ in MR by calculating the minimum amount of communication that any MR algorithm has to incur to solve $P$ at different parallelism levels. Similarly, we will measure how good an algorithm that solves $P$ is by measuring the algorithm’s communication cost and level of parallelism. We note that the focus of this chapter and Chapter 3 is one-round MR computations. In Chapter 4, we will consider multiround computations.

In the remainder of this section we first make the notion of “parallelism level” concrete. We then demonstrate the tradeoff between communication cost and parallelism on a running example called the all-to-all problem from reference [2]. We will use this problem to motivate our framework in Section 2.2. We then discuss how we can use this tradeoff in practice to pick an appropriate parallelism level for solving problems.

### 2.1.1 Reducer Size As a Measure of Parallelism Level

One way to characterize the parallelism level of a MR algorithm is to look at the number of reducers that it uses. Recall from Chapter 1 that a reducer, in the sense we use the term in this thesis, is a reducer machine that gets a single reduce-key, i.e., one of the keys that can appear in the output of the mappers, together with its list of associated values. However, the number of reducers can be a deceptive measure of an algorithm’s parallelism level. An algorithm can generate a very large number reducers, yet intentionally or unintentionally solve a problem effectively in a serial fashion in only one of the reducers. For example, suppose we want to join two 1 TB-size relational tables $R(A, B)$ with $S(B, C)$ by hashing on their $B$ attribute. That is, we use, say, 1000 reducers and randomly hash each $R$ and $S$ tuple on its $B$ value to one of the reducers. At first, this hash-join algorithm might seem like a highly parallel algorithm. However, if the input data is skewed and contains a single
B value, then a single reducer would receive the entire 2TB of input data and solve the problem in a purely serial fashion.

A better way to characterize the parallelism level of an algorithm is to look at its *reducer size*—the maximum amount of data mapped to a single reducer. This is a more robust measure of parallelism, because by getting less data, each reducer solves a smaller part of the problem and therefore algorithms need more reducers to solve a problem: decreasing the reducer size forces algorithms to increase their parallelism.

The hash-join algorithm above for skewed inputs would have a reducer size of 2TB, and therefore is essentially a serial algorithm. In Section 2.5, we will present an algorithm called the *Grouping* algorithm that can perform the same join with a reducer size of 2GB on any input, but that uses one million reducers. It is therefore a much more parallel algorithm than the hash-join algorithm. We also demonstrate a sample execution of the Grouping algorithm in Section 2.2.2 when we describe our framework.

With this justification in mind, we will use maximum reducer size as the measure of parallelism of algorithms, and study a fundamental tradeoff that exists between reducer size (or parallelism) and communication cost when solving many problems in MR.

### 2.1.2 An Example of the Tradeoff: All-to-All Problem (A2A)

Suppose we are given a set of $n$ inputs, $e_1, \ldots, e_n$, and we want to run a user-defined function (UDF) on all pairs of inputs. This problem has been called the all-to-all (A2A) problem in reference [2]. Table 2.1 shows the inputs and outputs for the A2A problem. As an example, consider the *drug interactions* problem from reference [122]. In this problem, each input is a record consisting of information about the medical history of patients who had taken a particular drug. The records for each pair of drugs are needed to be compared using a sophisticated UDF in order to determine whether a particular side effect is more common among patients using both drugs than those using only one or neither. More generally, the A2A problem appears in *similarity* or *fuzzy join* problems, in which the goal is to find pairs of inputs that have
a distance score less than a threshold. When the distance function is a black box UDF, then similarity join problems also reduce to the A2A problem.

Here are two one-round MR algorithms solving this problem at two extreme parallelism levels. In our analysis of these algorithms, we assume that each input is of the same size and we take the number of inputs as are unit of communication and memory. The first algorithm, PS, for purely serial, solves this problem in a purely serial fashion by using a single reducer and sending the entire input to that reducer. The communication cost of PS is equal to \( n \) and the sole reducer of the algorithm gets all of the \( n \) inputs. The second algorithm, MP, for max parallelism, uses \( \binom{n}{2} \) reducers \( r_{1,2}, r_{1,3}, \ldots, r_{n-1,n} \), one for each pair of inputs. MP maps each input \( e_i \) to all \( n-1 \) reducers that contain \( i \) in its index. The communication cost of MP is \( n(n-1) \), but each reducer now gets only 2 items.

Notice that there is a tradeoff between communication cost \( (C) \) and reducer size \( (M) \) as we switch from PS to MP for solving the A2A problem. \( C \) increases from 2 to \( n(n-1) \) and \( M \) decreases from \( n \) to 2. This tradeoff is actually an inherent property of the A2A problem and not due to a poor design of the PS or MP algorithms. On the contrary, one can show that both algorithms are incurring the minimum communication cost that is necessary to solve the A2A problem for their respective reducer sizes. Specifically, we will show in Section 2.2 that any algorithm that uses reducers of size \( M \) needs to incur a communication cost of at least \( f(M) = \frac{n(n-1)}{M-1} \).

Replacing the reducer sizes of PS and MP algorithms, we can see that both algorithms incur the optimal communication cost for the parallelism levels at which they execute.

The goal of the framework we present in this chapter is to derive such tradeoff

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Input₁</th>
<th>Input₂</th>
<th>UDF(Input₁, Input₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_1 )</td>
<td>( e_1 )</td>
<td>( e_2 )</td>
<td>UDF(( e_1 ), ( e_2 ))</td>
</tr>
<tr>
<td>( e_2 )</td>
<td>( e_1 )</td>
<td>( e_3 )</td>
<td>UDF(( e_1 ), ( e_3 ))</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( e_n )</td>
<td>( e_{n-1} )</td>
<td>( e_n )</td>
<td>UDF(( e_{n-1} ), ( e_n ))</td>
</tr>
</tbody>
</table>

(a) Input | (b) Output

**Table 2.1:** Inputs and outputs of the A2A problem.
functions between $C$ and $M$ for different problems and to analyze the performance of algorithms in light of these tradeoff curves.

### 2.1.3 How the Tradeoff Can Be Used

Suppose we have determined that the tradeoff for a problem $P$ between $C$ and $M$ is $C \geq f(M)$ for some function $f$. This tradeoff function is essentially a lower bound function on the communication cost of any MR algorithm solving $P$ at different parallelism levels $M$. Readers can look ahead to Fig. [2.2] for an example of what such a function $f$ might look like. In particular, the best algorithms solving $P$ will have a communication cost of $f(M)$ at parallelism level $M$, where $f(M)$ is an increasing function of $M$. Suppose such optimal algorithms exist for each parallelism level $M$.

When we try to solve an instance of $P$ on a particular machine cluster, we must determine the true costs of execution. For example, if we are running on EC2 [7], we pay particular amounts for communication and for rental of virtual processors. The price we pay for communication $p_c(C)$ is a function of $C$ that depends on the rate EC2 charges for communication. The price we pay for renting processors, $p_m(M)$, is some function of $M$. Notice that we need to rent $\frac{f(M)}{M}$ processors for each parallelism level because the communication $C = f(M)$ is the total amount of data sent to reducers, and each reducer holds $M$ of this data. Then, in order to minimize our budget for solving $P$ we would try to pick an algorithm at a parallelism level $M$ that minimizes $p_c(C) + p_m(M) \frac{f(M)}{M}$, which is equal to the following expression by substituting $f(M)$ for $C$:

$$p_c(f(M)) + p_m(M) \frac{f(M)}{M}$$

(2.1)

In a more detailed analysis we can also add another term $z(M)$ to Expression [2.1] for the amount of computation done by each reducer to estimate how long we will rent each processor. In general, different problems will have different tradeoff functions $f(M)$, and they will also have different functions of $M$ that measure the computation cost. Therefore, depending on the price of communication $p_c$, processors $p_m$, the tradeoff function $f(M)$, and the computation cost $z(M)$, we might optimize our
budgets differently by picking a different parallelism level to solve \( P \). Sometimes there may not be algorithms designed for each parallelism level. Then we would simply pick the algorithm that minimizes Expression 2.1 among the available parallelism levels \( M \).

### 2.1.4 Outline of the Chapter

In the rest of this chapter we first describe our framework (Section 2.2). Then, we demonstrate how to use our framework to analyze three different problems and algorithms for these problems: (i) dense matrix multiplication (Section 2.3); (ii) fuzzy joins with Hamming distance-1 (Section 2.4); and (iii) multiway equijoins (Section 2.5). In Chapter 3 we study the fuzzy joins problem with Hamming and edit distance \( d > 1 \) separately, because our algorithm there is significantly more complex than the algorithms presented in this chapter and we believe is interesting in its own right.

### 2.2 The Framework

Our framework is simple, yet powerful enough to develop interesting and realistic insights into the range of possible MR algorithms for a problem. It consists of: (1) formal representations of a computational problem (Section 2.2.1) and an MR algorithm (Section 2.2.2); (2) a cost model for algorithms (Section 2.2.3); and (3) a proof template for deriving tradeoff curves between communication cost and reducer sizes for solving a problem in one round of MR (Section 2.2.4). In Section 2.2.5 we also discuss the criteria for good MR algorithms in light of the tradeoff curve of a problem, and introduce the notion of Pareto optimality of algorithms. We end this section with a summary of our results (Section 2.2.6), elaborated in subsequent sections.

#### 2.2.1 Problem Representation: Bipartite Input-Output Graphs

A problem in our framework consists of:

1. A set of inputs and outputs.
2. A mapping from outputs to sets of inputs. The intent is that each output depends on the set of inputs it is mapped to.

A problem is essentially represented by a bipartite graph of inputs and outputs. For some problems, the inputs and outputs represent the actual inputs and outputs. In this case, each input $i$ will have some associated data that is required to generate the outputs that depend on $i$. For other problems, inputs and outputs represent the possible inputs and outputs. In this case, each instance of the problem will contain a subset of the inputs. The corresponding output for that instance will be the subset of possible outputs that have all of their dependencies present in the input. We give two examples to demonstrate each case.

Example 2.2.1. Figure 2.1a shows the representation of the A2A problem. Each $e_i$ is an actual input. Each output is a triple $<e_i, e_j, UDF(e_i, e_j)>$ that depends on the inputs $e_i$ and $e_j$.

Example 2.2.2. Figure 2.1b shows the representation of the natural join of three relations $R_1(A_1, A_2)$, $R_2(A_2, A_3)$, and $R_3(A_3, A_4)$, which we call the Chain-3 join problem. The inputs are possible tuples in $R_1$, $R_2$, and $R_3$, and the outputs are tuples with schema $(A_1, A_2, A_3, A_4)$. For simplicity, we assume that each attribute $A_i$ is from a finite domain with size 100. Each output tuple $(a_{1,i}, a_{2,j}, a_{3,k}, a_{4,l})$ depends on the $(a_{1,i}, a_{2,j})$ tuple from $R_1$, the $(a_{2,j}, a_{3,k})$ tuple from $R_2$, and the $(a_{3,k}, a_{4,l})$ tuple from $R_3$. 
When inputs represent possible as opposed to actual inputs, we will sometimes be interested in solving a distribution of instances. That is, rather than solving the problem for any instance, we will identify a subset of the instances. For example, for the Chain-3 join, we might be interested in solving the problem over the distribution of instances that contain, say, $X$ tuples from each relation.

### 2.2.2 Algorithm Representation: Mapping Schemas

A MR algorithm for a problem in our framework is represented as a mapping schema. A mapping schema is an assignment of the inputs to a set of reducers, such that for every output, there is at least one reducer that is assigned all of the inputs for that output. We say such a reducer covers the output. This reducer need not be unique, and it is, of course, permitted that these same inputs are assigned also to other reducers. We note that there is no restriction to the number of reducers a mapping schema can use.

**Example 2.2.3.** The PS algorithm for the A2A problem from Section 2.1.2 is a mapping schema that uses a single reducer and maps each input to that reducer. Trivially, the inputs of every output $\langle e_i, e_j, UDF(e_i, e_j) \rangle$ are mapped to this single reducer together. Similarly in the MS algorithm, which uses $\binom{n}{2}$ reducers, each output $\langle e_i, e_j, UDF(e_i, e_j) \rangle$ is covered by reducer $r_{i,j}$.

**Example 2.2.4.** Consider the following algorithm, called the Grouping algorithm, for solving Chain-3 problem over all instances that contain $X$ tuples. We will generalize the Grouping algorithm in Section 2.5.4. The algorithm divides each relation into $g$ groups of size $\frac{X}{g}$ tuples. For simplicity, we assume that with each tuple there is an index between 1 and $X$ that allows us to perform this grouping. Otherwise, we can randomly assign each tuple to one of the $g$ groups. The algorithm uses $g^3$ reducers $r_{1,1,1}, \ldots, r_{g,g,g}$. The tuples in group $i$ for relation $R_j$ are assigned to all reducers that contain $i$ in its $j$th index, for $i \in \{1, 2, \ldots, g\}$ and $j \in \{1, 2, 3\}$. Now consider any output $\langle a, b, c, d \rangle$ of the given instance and suppose that $(a, b)$ belongs to group $i$ of $R_1$, $(b, c)$ belongs to group $j$ of $R_2$, and $(c, d)$ belongs to group $k$ of $R_3$. Then $\langle a, b, c, d \rangle$ would be covered by reducer $r_{i,j,k}$. 
2.2.3 Cost Model

Our framework focuses on two costs of mapping schemas:

1. **Reducer size (M):** Suppose a mapping schema \( A \) uses \( p \) reducers and assigns \( M_i \) inputs to reducer \( r_i \). If the inputs are possible and not actual, then we suppose that \( A \) assigns at most \( M_i \) inputs to \( r_i \) on any instance of the problem. The reducer size cost of \( A \) is defined to be \( M = \max_{i \in [1,p]} M_i \). We note that if \( A \) solves a distribution of problem instances, \( M_i \) is taken as the *worst case* or maximum number of inputs assigned to \( r_i \) in any of the instances in the distribution. As we discussed in Section 2.1.1, the reducer size characterizes the parallelism level of the algorithm, where algorithms with smaller reducer sizes are more parallel. Also, the reducer size captures the amount of memory that is required in the machines of a cluster to execute the algorithm. In practice, if the machines of a cluster have less memory than the reducer size cost of an algorithm, then one of the machines would run out of the main memory while executing the algorithm on some problem instance.

2. **Communication (C):** The communication cost of a mapping schema is the total amount of data that it maps to its \( p \) reducers in the worst case or \( C = \sum_{i=1}^{p} M_i \). The communication cost captures the amount of data that is shuffled between mappers and reducers during the computation.

For simplicity, we take each input to be of the same length and therefore our unit of data is not bits but the number of inputs.

**Example 2.2.5.** For the PS algorithm both the reducer size and communication cost are equal to \( n \). In contrast, the MS algorithm has a reducer size cost of \( 2 \) and a communication cost of \( n(n - 1) \), since each input is mapped to \( n - 1 \) reducers.

**Example 2.2.6.** In the Grouping algorithm from Example 2.2.4, no matter what the actual input is, each reducer \( r_{i,j,k} \) gets \( \frac{X}{g} \) tuples from each relation. Therefore the reducer size cost of the Grouping algorithm is \( \frac{3X}{g} \). The communication cost, which is equal to the reducer size times the number of reducers, is \( g \cdot \frac{3X}{g} = 3Xg^2 \). For example if \( g \) is \( 1 \), then both the reducer size and communication cost is \( 3X \). If \( g \) is \( X \), then the reducer size is \( 3 \) and the communication cost is \( 3X^3 \).
2.2.4  A Proof Template for Deriving Tradeoff Curves

While upper bounds on the communication cost for all problems are derived using constructive algorithms, there is a generic technique for deriving tradeoff curves, i.e., lower bounds on the communication cost for different values of $M$. We next outline the argument that we use to derive all of the lower bounds in this chapter.

1. **Deriving $g(M)$:** First, find an upper bound, $g(M)$, on the number of outputs a reducer can cover if $M$ is the number of inputs it is given.

2. **Number of Inputs and Outputs:** Count the total numbers of inputs $|I|$ and outputs $|O|$. If we are solving a distribution of problem instances, we take this to be the maximum number of inputs and outputs. However, to simplify our proofs in this thesis, we will consider distributions where each instance has the same number of inputs and outputs.

3. **The Inequality:** Assume there are $p$ reducers, each receiving $M_i \leq M$ inputs and covering at most $g(M_i)$ outputs. Together they must cover all of the outputs. That is:

\[
\sum_{i=1}^{p} g(M_i) \geq |O| \tag{2.2}
\]

4. **Replication Rate:** Manipulate the inequality from Equation 2.2 to get a lower bound on the communication cost, which is $\sum_{i=1}^{p} M_i$.

Note that the last step above may require clever manipulation to factor out the communication cost. We have noticed that the following “trick” is effective in Step (4) for all problems we have considered. First, arrange to isolate a single factor $M_i$ from $g(M_i)$; that is:

\[
\sum_{i=1}^{p} g(M_i) \geq |O| \Rightarrow \sum_{i=1}^{p} M_i \frac{g(M_i)}{M_i} \geq |O| \tag{2.3}
\]
Assuming $\frac{g(M_i)}{M_i}$ is monotonically increasing in $M_i$, we can use the fact that $\forall M_i: M_i \leq M$ to obtain from Equation 2.3:

$$\sum_{i=1}^{p} M_i \frac{g(M)}{M} \geq |O|$$

(2.4)

Now, we move $\frac{g(M)}{M}$ to the right side of Equation 2.4 to get a formula with the communication cost on the left:

$$C = \sum_{i=1}^{p} M_i \geq \frac{M|O|}{g(M)}$$

(2.5)

Equation 2.5 gives us a lower bound on $C$.

**Example 2.2.7.** We apply the above 4-step proof template to derive a lower bound on the communication cost of algorithms solving the A2A problem.

- **Deriving $g(M)$**: With $M$ inputs, a reducer can cover at most $g(M) = \binom{M}{2}$ pairs of outputs.

- **Number of Inputs and Outputs**: There are $n$ inputs, i.e., $|I| = n$ and $\binom{n}{2}$ outputs, i.e., $|O| = \binom{n}{2}$.

- **$\sum_{i=1}^{p} g(M_i) \geq |O|$ Inequality**: Substituting for $g(M)$ and $|O|$ from above:

$$\sum_{i=1}^{p} \frac{M_i(M_i - 1)}{2} \geq \frac{n(n - 1)}{2}$$

- **Lower Bound On $C$**: Finally we employ the manipulation trick to derive a lower bound on $C$.

$$\sum_{i=1}^{p} \frac{M_i(M_i - 1)}{2} \geq \frac{n(n - 1)}{2} \Rightarrow \sum_{i=1}^{p} \frac{M_i(M - 1)}{2} \geq \frac{n(n - 1)}{2}$$

$$C = \sum_{i=1}^{p} M_i \geq \frac{n(n - 1)}{M - 1}$$

The orange line in Figure 2.2 shows this tradeoff curve graphically.
2.2.5 Criteria for Good MR Algorithms

There are two desired properties of algorithms in light of the lower bounds on the communication costs of algorithms for different reducer sizes:

- **Pareto Optimality**: We want algorithms that incur the minimum possible communication cost for different parallelism levels. Borrowing from the economics literature, we call an algorithm *Pareto optimal* if it incurs the minimum communication cost for its reducer size. In economics, an allocation of resources is Pareto optimal if it is not possible to make any individual better off without making someone else worse off. Similarly, we call an algorithm Pareto optimal if it is not possible to reduce its communication cost without increasing its reducer size or vice-versa. For example, both the PS and MP algorithms, shown as two black dots in Figure 2.2, are Pareto optimal for the A2A problem.

- **Flexibility in Parallelism**: Although the PS and MP algorithms are Pareto optimal, they may not be very useful for users solving the A2A problem because both algorithms execute only at one parallelism level. A better algorithm would be flexible in its parallelism levels, i.e., can be parameterized to run at different level of $M$ to give more options to the users when optimizing their budgets. In particular, the best one-round MR algorithm would be flexible enough to execute at every parallelism level and also be Pareto optimal (or almost Pareto optimal)
TABLE 2.2: Summary of one-round MR results.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input Size</th>
<th>Tradeoff Curve</th>
<th>Parallelism Levels for Pareto Optimal Algorithms</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense Matrix Multiplication (two $n \times n$ matrices)</td>
<td>$2n^2$</td>
<td>$C \geq \frac{4n^4}{M}$</td>
<td>$\forall M = 2sn$, where $s$ divides $n$ (Square Tiling)</td>
<td>2.3</td>
</tr>
<tr>
<td>Hamming-Distance-1 (n-bit strings)</td>
<td>$2^n$</td>
<td>$C \geq \frac{2^n n}{\log_2 M}$</td>
<td>$\forall M = 2^{n/s}$ where $s$ divides $n$ (Generalized Splitting)</td>
<td>2.4</td>
</tr>
<tr>
<td>Multiway Equijoins (m tables of size $\frac{IN}{m}$)</td>
<td>$IN$</td>
<td>$C \geq \frac{IN\rho^<em>}{M\rho^</em>-1}$</td>
<td>approximately for all $M$ either over a specific distribution of inputs on any query (Shares [1]) or for any instance on specific queries (Grouping)</td>
<td>2.5</td>
</tr>
</tbody>
</table>

for each level. Such an algorithm for the A2A problem would be plotted as the squares in Figure 2.2.

2.2.6 Summary of Our Results

Table 2.2 summarizes our results. The table enumerates for each problem the total number of inputs $|I|$, the lower bound on the communication cost we derived, and the parallelism levels for which Pareto optimal algorithms exist for the problem. For all problems listed, we have algorithms that are Pareto optimal for a wide range of values. For the multiway equijoin problem, the algorithms we discuss are Pareto optimal either over a specific distribution of inputs but for any query, or for an arbitrary instance on a set of specific queries. For this problem, the term IN specifies the number of tuples in the input for the problem and the parameter $\rho^*$ will be described in Section 2.5.
2.3 Dense Matrix Multiplication

We start with the problem of dense matrix multiplication. We are given as input two \( n \times n \) matrices \( R = [r_{ij}] \) and \( S = [s_{jk}] \) and we wish to form their product \( T = [t_{ik}] \), where \( t_{ik} = \sum_{j=1}^{n} r_{ij} s_{jk} \). As shown in Figure 2.3, each output \( t_{ik} \) depends on an entire row of \( R \) and an entire column of \( S \), that is, \( 2n \) inputs, as suggested by Fig. 2.3.

![Figure 2.3: Input/output relationship for the matrix-multiplication problem.](image)

2.3.1 The Tradeoff Curve

We use our 4-step proof template to derive the tradeoff curve.

- **Deriving \( g(M) \):** Suppose a reducer covers the outputs \( t_{14} \) and \( t_{23} \). Then all of rows 1 and 2 of \( R \) are input to that reducer, and all of columns 4 and 3 of \( S \) are also inputs. Thus, this reducer also covers outputs \( t_{13} \) and \( t_{24} \). As a result, the set of outputs covered by any reducer form a “rectangle,” in the sense that there is some set of rows \( i_1, i_2, \ldots, i_w \) of \( R \) and some set of columns \( k_1, k_2, \ldots, k_h \) of \( S \) that are input to the reducer, and the reducer covers all outputs \( t_{i_u k_v} \), where \( 1 \leq u \leq w \) and \( 1 \leq v \leq h \).

We can assume this reducer has no other inputs, since if an input to a reducer is not part of a whole row of \( R \) or column of \( S \), it cannot be used in any output made by the reducer. Thus, the number of inputs to this reducer is \( n(w + h) \), which must be less than or equal to \( M \), the upper bound on the number of inputs to a reducer. As the total number of outputs covered is \( wh \), it is easy to show...
that for a given $M$, the number of outputs is maximized when the rectangle is a square; that is, $w = h = M/(2n)$. In this case, the number of outputs covered by the reducer is $g(M) = \frac{M^2}{4n^2}$.

• **Number of Inputs and Outputs:** There are two matrices each of size $n^2$. Therefore $|I| = 2n^2$ and $|O| = n^2$.

• $\sum_{i=1}^{p} g(M_i) \geq |O|$ Inequality: Substituting for $g(M_i)$ and $|O|$:

$$\sum_{i=1}^{p} \frac{M_i^2}{4n^2} \geq n^2$$

• **Lower Bound On C:** We first leave one factor of $M_i$ on the left as is, and replace the other factor $M_i$ by $M$. Then, we manipulate the inequality so the expression on the left is the communication and obtain:

$$C = \sum_{i=1}^{p} M_i \geq \frac{4n^4}{M}$$

The tradeoff curve is shown by the curve in Figure 2.4.
2.3.2 Algorithms

Figure 2.4 shows the algorithms that match the lower bound $C \geq \frac{4n^4}{M}$ for a wide range of parallelism levels. The purely serial (PS) algorithm that performs the entire multiplication at one reducer would trivially cover every output and have a reducer size and communication cost of $2n^2$, which would be Pareto optimal. A maximum parallel (MP) algorithm that has one reducer $r_{ik}$ for each output cell $t_{ik}$ would send one row of $R$ and one row of $S$ to each reducer. Note that this algorithm would by construction cover every output. It would also be Pareto optimal with a reducer size of $2n$ and communication cost of $2n^3$.

As we show next, we can actually generalize the PS and MP algorithms and match the lower bound by giving each reducer a set of rows of $R$ and an equal number of columns of $S$. We call this the Square Tiling algorithm. The technique of computing the result of a matrix multiplication by tiling the output by squares is very old indeed [88]. Let $s$ be an integer that divides $n$, and let $M = 2sn$. Partition the rows of $R$ into $n/s$ groups of $s$ rows, and do the same for the columns of $S$. There is one reducer for each pair $(G, H)$ consisting of a group $G$ of $R$’s rows and a group $H$ of $S$’s columns. So there are $n/s \times n/s = n^2/s^2$ reducers in total. This reducer has $M = 2sn$ inputs, and can produce all the outputs $t_{ik}$ such that $i$ is one of the rows in group $G$ and $k$ is one of the columns in the group $H$. Since every pair $(i, k)$ has $i$ in some group for $R$ and has $k$ in some group for $S$, every $t_{ik}$ will be produced by exactly one reducer.

Since there are $n^2/s^2$ reducers and each one gets $2sn$ inputs, the communication cost of the Square Tiling algorithm is $2n^3/s$. If we choose $s = 2n/M$, we see that the communication cost is exactly $4n^4/M$, which matches the lower bound exactly. So for each $s$ that divides $n$, the Square Tiling algorithm is Pareto optimal with an $M$ value of $2sn$. Essentially, if there are a large number of $s$ values that divide $n$, Square Tiling is the best one-round MR algorithm we can hope for.
Chapter 2. A Theoretical Framework for MapReduce

2.4 Hamming Distance 1

We next study a type of fuzzy or similarity join problem of finding pairs of similar strings from a given corpus. Our setup is similar to the A2A problem. We are given as input a corpus of all bit strings of length $n$, and with each bit string some unknown associated data. Our goal will be to execute some UDF on the associated data for each pair of strings $(s, t)$ that are at Hamming Distance 1 (HD-1), i.e., $s$ and $t$ change in exactly one bit. Figure 2.5 shows in the input and output representation of the HD-1 problem. Such fuzzy join computations appear in several applications, such as recommendation systems [109], entity recognition among labeled records in the web [127], and clustering large-scale genetics data [92]. For example, consider a friendship recommendation system in a social network. Bit strings could represent a set of features of customers. For example, a 1/0 in the first bit could represent a customer’s gender being male or female, and the second bit could represent a customer’s age being less than or equal to 30 or greater than 30. In addition, each bit string $s$ could have an associated list of customers who have exactly the features that $s$ represents. A recommendation algorithm can then recommend friendships between customers that have similar features, say those whose feature strings $s$ and $t$ are at some Hamming distance $d$.

We start our analysis of HD-1 with a lemma bounding $g(M)$—the maximum

![Figure 2.5: Input/output relationship for HD-1.](image)
number of outputs that a reducer can cover with $M$ inputs; we will use this lemma in our proof of the tradeoff curve for HD-1.

**Lemma 2.4.1.** For HD-1, a reducer that is assigned $M$ inputs can cover no more than $(M/2) \log_2 M$ outputs.

**Proof.** The proof is an induction on $n$, the length of the bit strings in the input. The basis is $n = 1$. Here, there are only two strings, so $M$ is either 1 or 2. If $M = 1$, the reducer can cover no outputs. $(M/2) \log_2 M$ is 0 when $M = 1$, so the lemma holds in this case. If $M = 2$, the reducer can cover at most one output. $(M/2) \log_2 M$ is 1 when $M = 2$, so again the lemma holds.

Now let us assume the bound for $n$ and consider the case where the inputs consist of strings of length $n + 1$. Let $X$ be a set of $M$ bit strings of length $n + 1$. Let $Y$ be the subset of $X$ consisting of those strings that begin with 0, and let $Z$ be the remaining strings of $X$—those that begin with 1. Suppose $Y$ and $Z$ have $y$ and $z$ members, respectively, so $M = y + z$.

An important observation is that for any string in $Y$, there is at most one string in $Z$ at Hamming distance 1. That is, if $0w$ is in $Y$, it could be Hamming distance 1 from $1w$ in $Z$, if that string is indeed in $Z$, but there is no other string in $Z$ that could be at Hamming distance 1 from $0w$, since all strings in $Z$ start with 1. Likewise, each string in $Z$ can be distance 1 from at most one string in $Y$. Thus, the number of outputs with one input in each of $Y$ and $Z$ is at most $\min(y, z)$.

Let us count the maximum number of outputs that can have their inputs within $X$. By the inductive hypothesis, there are at most $(y/2) \log_2 y$ outputs both of whose inputs are in $Y$, at most $(z/2) \log_2 z$ outputs both of whose inputs are in $Z$, and, by the observation in the paragraph above, at most $\min(y, z)$ outputs with one input in each of $Y$ and $Z$.

Assume without loss of generality that $y \leq z$. Then the maximum number of strings of length $n + 1$ that can be covered by a reducer with $M$ inputs is

$$\frac{y}{2} \log_2 y + \frac{z}{2} \log_2 z + y$$

We must show that this function is at most $(M/2) \log_2 M$, or, since $M = y + z$, we
need to show:

\[
\frac{y}{2} \log_2 y + \frac{z}{2} \log_2 z + y \leq \frac{y + z}{2} \log_2(y + z) \tag{2.6}
\]

under the condition that \(y \leq z\).

First, observe that when \(y = z\), Equation 2.6 holds with equality. That is, both sides become \(y(\log_2 y + 1)\). Next, consider the derivatives, with respect to \(z\), of the two sides of Equation 2.6. \(d/dz\) of the left side is

\[
\frac{1}{2} \log_2 z + \frac{\log_2 e}{2}
\]

while the derivative of the right side is

\[
\frac{1}{2} \log_2(y + z) + \frac{\log_2 e}{2}
\]

Since \(0 \leq y \leq z\), the derivative of the left side is always less than or equal to the derivative of the right side. Thus, as \(z\) grows larger than \(y\), the left side remains no greater than the right. That proves the induction step, and we may conclude the lemma.

\[\square\]

### 2.4.1 The Tradeoff Curve

We can use Lemma 2.4.1 to derive the tradeoff curve for HD-1. We use our proof template from Section 2.2.4.

- **Deriving \(g(M)\):** By Lemma 2.4.1, \(g(M) = (M/2) \log_2 M\)
- **Number of Inputs and Outputs:** There are \(2^n\) bitstrings of length \(n\). The total number of outputs is \((n/2)2^n\). Therefore \(|I| = 2^n\) and \(|O| = (n/2)2^n\).
- **\(\sum_{i=1}^{p} g(M_i) \geq |O|\) Inequality:** Substituting for \(g(M_i)\) and \(|O|\) from above:

\[
\sum_{i=1}^{p} \frac{M_i}{2} \log_2 M_i \geq \frac{n}{2} 2^n \tag{2.7}
\]
Figure 2.6: Tradeoff curve and algorithms for HD-1.

- **Lower Bound on $C$:** Finally we employ the manipulation trick from Section 2.2.4 where we arrange the terms of this inequality so that the left side is the communication cost. Recall we must separate a factor $M_i$ from other factors involving $M_i$ by replacing all other occurrences of $M_i$ on the left by the upper bound $M$. That is, we replace $\log_2 M_i$ by $\log_2 M$ on the left of Equation 2.7. Since doing so can only increase the left side, the inequality continues to hold:

$$\sum_{i=1}^{p} \frac{M_i}{2} \log_2 M \geq \frac{n}{2} 2^n$$  \hspace{1cm} (2.8)

The communication cost is $C = \sum_{i=1}^{p} M_i$. We can move factors in Equation (2.8) to derive the tradeoff curve for HD-1: $C = \sum_{i=1}^{p} M_i \geq n 2^n / \log_2 M$.

The tradeoff curve for HD-1 is shown as the line in Figure 2.6. Note that compared to both the matrix multiplication and the A2A problem, HD-1 is easier to parallelize. More specifically, changes to parallelism levels ($M$) have very little effect on the total communication costs of algorithms, because $M$ is inside the logarithm.

### 2.4.2 Algorithms

Figure 2.6 shows the algorithms that match the lower bound $C \geq n 2^n / \log_2 M$ for a wide range of parallelism levels. Similar to the matrix multiplication and A2A problems, we have a purely serial (PS) and a maximum parallel (MP) algorithms.
on two extremes. Similarly, we can generalize PS and MP by an algorithm we call Generalized Splitting, which is Pareto optimal for a variety of parallelism levels.

**PS:** We have one reducer that gets all inputs. Therefore, $M = C = 2^n$, which is Pareto optimal.

**MP:** We have one reducer $r_{s,t}$ for each pair of strings $(s,t)$ that are Hamming distance 1. We send each string $s$ to all reducers that contain $s$ in its index. Each reducer $r_{s,t}$ gets exactly two strings, therefore $M = 2$. There are $(n/2)2^n$ outputs, so there are $(n/2)2^n$ reducers, each getting 2 strings. Therefore the communication cost is $n2^n$, which is also Pareto optimal.

**Generalized Splitting:** In reference [3], there is an algorithm called Splitting for solving fuzzy joins with general Hamming distance $d$. For HD-1 Splitting uses $2^{n/2+1}$ reducers, assuming $n$ is even. Half of these reducers correspond to the $2^{n/2}$ possible bit strings that may be the first half of an input string. Call these *Group I reducers*. The second half of the reducers correspond to the $2^{n/2}$ bit strings that may be the second half of an input. Call these *Group II reducers*. Thus, each bit string of length $n/2$ corresponds to two different reducers.

An input $s$ is sent to 2 reducers: the Group-I reducer that corresponds to its first $n/2$ bits, and the Group-II reducer that corresponds to its last $n/2$ bits. Splitting covers every output pair, because every pair of inputs $(s,t)$ must either agree in the first half of their bits, in which case they are sent to the same Group-I reducer, or they agree on the last half of their bits, in which case they are sent to the same Group-II reducer. Splitting uses reducers of size $M = 2^{n/2}$ and incurs a communication cost of $2^{n+1}$ because each input is assigned to exactly two reducers. Therefore according to our tradeoff curve, Splitting is Pareto optimal.

We can generalize the Splitting Algorithm to divide strings into $c > 2$ pieces for any $c$ that divides $n$. As we show, with this generalization, we also recover the performances of the PS, MP, and Splitting algorithms. We call this algorithm *Generalized Splitting*. We have $c$ groups of reducers, numbered 1 through $c$. For $i = 1, \ldots, c$, we map $s$ to the Group-$i$ reducer that corresponds to bit string $s_1 \cdots s_{i-1}s_{i+1} \cdots s_c$, that is, $s$ with the $i$th substring $s_i$ removed. We need to argue that this mapping schema solves the problem. Any two strings $s$ and $t$ at Hamming distance 1 will disagree
in only one of the $c$ segments of length $n/c$, and will agree in every other segment. If they disagree in their $i$th segments, then they will be sent to the same Group-$i$ reducer, because we map them to the Group-$i$ reducers ignoring the values in their $i$th segments. Thus, this Group-$i$ reducer will cover the output pair $(s, t)$.

Finally, let us compute the reducer size and communication cost of Generalized Splitting for different values of $c$. In each group, there are $2^{n-n/c}$ reducers, corresponding to each of the $2^{n-n/c}$ bit strings of length $n-n/c$. Since there are exactly $2^{n/c}$ strings with a particular fixed $n-n/c$ bits, each reducer gets exactly $2^{n/c}$ strings. Since each string is being sent to exactly $c$ reducers, the communication cost is equal to $c2^n$, which is the optimal communication cost for $M = 2^{n/c}$ according to the tradeoff curve.

### 2.5 Multiway Equijoins

In this section, we study the problem of computing the multiway equijoin or natural join of multiple database relations in MR. We are given as input $n$ relations $R_1, \ldots, R_n$, each with an equal number of $IN/n$ tuples, where $IN$ is some positive number. Let $A_1, \ldots, A_m$ be the $m$ attributes that appear in the schemas of the input relations, each taking values from a large domain $\mathcal{D}$. Let $\text{attrs}(R_j)$ be the set of attributes that appear in the schema of $R_j$. Our goal is to compute the equijoin query $Q(A_1, \ldots, A_m) = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. As an example, recall the Chain-3 join from Section 2.2.1 whose possible input and outputs with $|\mathcal{D}| = 100$ were given in Figure 2.1b.

Compared to the A2A, matrix multiplication, and the HD-1 fuzzy joins problem we studied, the version of the multiway equijoins we study here has two differences:

1. The inputs and outputs are not actual but possible inputs and outputs. As a result, when deriving the tradeoff curve, we will constrain ourselves to a distribution of instances called the max-out distribution (Section 2.5.2).

2. There is not a single tradeoff curve applicable to all queries. Each query has a separate tradeoff curve that depends on the attribute structures of the input relations. Specifically, each query has a property called an edge covering number.\[15\]
CHAPTER 2. A THEORETICAL FRAMEWORK FOR MAPREDUCE

### Fractional Edge Covering

<table>
<thead>
<tr>
<th>Fractional Edge Covering</th>
<th>Fractional Vertex Packing</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall j \in [1, n], u_j \geq 0$</td>
<td>$\forall i \in [1, m], v_i \geq 0$</td>
</tr>
<tr>
<td>$\forall i \in [1, m], \sum_{j \text{ s.t. } A_i \in \text{attrs}(R_j)} u_j \geq 1$ (2.9)</td>
<td>$\forall j \in [1, n], \sum_{i \text{ s.t. } A_i \in \text{attrs}(R_j)} v_i \leq 1$ (2.10)</td>
</tr>
<tr>
<td>minimize $\sum_{j \in [1, n]} u_j$</td>
<td>maximize $\sum_{i \in [1, m]} v_i$</td>
</tr>
</tbody>
</table>

**Figure 2.7:** Fractional edge coverings and vertex packings of queries.

which defines the tradeoff curve for the query.

Before we derive the tradeoff curve, we first review two concepts, *fractional edge coverings* and *fractional vertex packings* of a query, and then define the max-out distribution.

##### 2.5.1 Fractional Edge Coverings

A fractional edge cover of $Q$ is any feasible solution of the linear program (LP) shown on the left of Figure 2.7. A fractional edge cover assigns a non-negative real number $u_j$ to each relation $R_j$ such that every attribute $A_i$ is “covered”, i.e., $\sum_{j \text{ s.t. } A_i \in \text{attrs}(R_j)} u_j \geq 1$. The *value of a fractional edge cover* is the sum of $u_j$ over all $j$’s. The edge covering *number* of a query $Q$, $\rho^*$, is the solution to this LP, i.e., value of the fractional edge cover with the minimum value. The name “edge” covering, as opposed to, say, “relation” covering, will become clear in Chapter 4 when we define the hypergraph representations of queries.

In a remarkable result [15], Atserias, Grohe, and Marx have shown a tight bound on the maximum number of tuples in the output of $Q$, $|Q|$, that is a function of only the sizes of the input tables and $\rho^*$. We call this the *AGM bound* after the names of authors of reference [15]. In our setting, when joining $n$ tables of size $\text{IN}/n$, their results imply that $|Q| \leq (\text{IN}/n)^{\rho^*}$. Table 2.8 gives several example queries and their edge covering numbers. The values under each relation represent the $u_j$ value assigned to the relation in an optimal fractional edge covering of the query. For example, consider the Chain-n query: $R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4) \bowtie \ldots \bowtie R_n(A_n, A_{n+1})$. One
can show that minimum value of any edge covering number is obtained by assigning 1 to $R_1$ and $R_n$ and alternating between 0 and 1 when assigning values to $R_2$ through $R_{n-1}$. Therefore $\rho^*$ for Chain-n is equal to $\lceil n/2 \rceil$ and the maximum size of the output of the Chain-n query when each table has $\IN/n$ tuples is equal to $(\IN/n)^{\lceil n/2 \rceil}$. We note that the optimal fractional edge covering need not be unique.

### 2.5.2 Vertex Packing and Max-Out Distribution

A *fractional vertex packing* of a query, shown on the right side of Figure 2.7, is a solution to the dual of the edge covering LP. A vertex packing associates a non-negative real number $v_i$ with each attribute $A_i$ such that no relation $R_j$ is “over packed”, i.e. $\sum_{i.t. A_i \in \text{attrs}(R_j)} v_i \leq 1$. The *value of a fractional vertex packing* is the sum of $v_i$. Due to strong duality of LP, the edge covering number of a query is equal to the optimal (or maximum) value of its vertex packing.

Atserias et al. has used an optimal vertex packing to construct an input for $Q$ with an output that is exactly equal to $(\IN/n)^{\rho^*}$, thereby showing that the $(\IN/n)^{\rho^*}$ is tight. We review a modified version of their construction to define a distribution of instances, which we call the *max-out distribution*, where each instance of the query has exactly $(\IN/n)^{\rho^*}$ outputs.

Pick an optimal vertex packing $\vec{v}^* = (v_1^*, \ldots, v_m^*)$ of the query $Q$. Pick any subset of $(\IN/n)^{\nu^*}$ values from domain $D$ for each attribute $A_i$ and let $R_j$ consist of the Cartesian product of the values in $\text{attrs}(R_j)$. Then:

$$
\begin{array}{|c|c|c|}
\hline
\text{Query} & \text{Edge Covering} & \rho^* \\
\hline
\text{Chain-n} & R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4) \bowtie \cdots \bowtie R_n(A_n, A_{n+1}) & 1 \\
& 1 & 0 & 1 & \ldots & 1 & \lceil n/2 \rceil \\
\hline
\text{Cycle-n} & R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4) \bowtie \cdots \bowtie R_n(A_n, A_1) & 1/2 \\
& 1/2 & 1/2 & 1/2 & \ldots & 1/2 & n/2 \\
\hline
\text{Snowflake-n} & S(B_1, \ldots, B_n) \bowtie R_1(A_1, B_1) \bowtie R_2(A_2, B_2) \bowtie \cdots \bowtie R_n(A_n, B_n) & 0 \\
& 0 & 1 & 1 & \ldots & 1 & n \\
\hline
\text{Star-n} & R_1(A_1, B) \bowtie R_2(A_2, B) \bowtie \cdots \bowtie R_n(A_n, B) & 1 \\
& 1 & 1 & \ldots & 1 & n \\
\hline
\end{array}
$$

Figure 2.8: Example queries and their edge covering numbers.
CHAPTER 2. A THEORETICAL FRAMEWORK FOR MAPREDUCE 41

$$|R_j| = \pi_{i \in \text{attrs}(R_j)} (\text{IN}/n)^{v_i^*} = (\text{IN}/n)^{\Sigma_{i \in \text{attrs}(R_j)} v_i^*} \leq \text{IN}/n$$

The last inequality above follows from the fact that $\Sigma_{i \in \text{attrs}(R_j)} v_i^* \leq 1$ by definition of a vertex packing. If $|R_j|$ is strictly less than $\text{IN}/n$, we add dummy null tuples to it to make it of size $\text{IN}/n$. Let us next calculate the size of the output of $Q$. Note that by construction, two relations $R_j$ and $R_k$ that have the same attribute $A_i$ have exactly the same values. Let $\gamma^*$ be the value of $\vec{v}^*$. Then, $|Q| = \pi_{i \in [1,m]} (\text{IN}/n)^{v_i^*} = (\text{IN}/n)^{\gamma^*}$. Note that since we picked an optimal vertex packing, by strong duality of LP, $\gamma^* = \Sigma_i v_i = \rho^*$. Therefore the total size of $Q = R_1 \bowtie \ldots \bowtie R_n$ is exactly equal to $(\text{IN}/n)^{\rho^*}$.

The max-out distribution of $Q$ with input size $\text{IN}$ and vertex packing $\vec{v}^*$ consists of all of the instances where the inputs correspond to all combinations of picking the attributes $A_i$ according to the construction we described above. When obvious from the context, we omit the vertex packing $\vec{v}^*$ that the max-distribution comes from. We next give an example.

**Example 2.5.1.** Consider the Chain-2 query $R_1(A_1, A_2) \bowtie R_2(A_2, A_3)$. The optimal vertex packing for Chain-2 is $\vec{v}^* = (1, 0, 1)$. That is $\vec{v}^*$ associates 1 with $A_1$, 0 with $A_2$, and 1 with $A_3$, which is also the unique vertex packing with value $\gamma^* = 2$. Let $|D| = 4$. Let us describe the max-out distribution for Chain-2 with input size 6. There are $\binom{4}{(6/2)} = 4$ possible subsets of $A_1$, $\binom{4}{(6/2)} = 4$ possible subsets of $A_2$, and $\binom{4}{(6/2)} = 4$ possible subsets of $A_3$. Therefore there are $4 \times 4 \times 4 = 64$ possible instances of the max-distribution of Chain-2 with input size 6. Here is one example that corresponds to the choices of $A_1 = \{a_{1,1}, a_{1,2}, a_{1,4}\}$, $A_2 = \{a_{2,1}\}$, and $A_3 = \{a_{3,1}, a_{3,2}, a_{3,3}\}$. Then, $R_1$ contains three tuples $(a_{1,1}, a_{2,1}), (a_{1,2}, a_{2,1}), (a_{1,4}, a_{2,1})$, and $R_3$ contains three tuples $(a_{2,1}, a_{3,1}), (a_{2,1}, a_{3,2}), (a_{2,1}, a_{3,3})$. The output of this instance contains 9 tuples formed from the Cartesian product of $\{a_{1,1}, a_{1,2}, a_{1,4}\} \times \{a_{2,1}\} \times \{a_{3,1}, a_{3,2}, a_{3,3}\}$. Finally, we note that according to the AGM bound, the maximum output for the Chain-2 join with 3 tuples for $R_1$ and 3 tuples for $R_2$ is equal to $3^{\rho^*} = 3^{\gamma^*} = 3^2 = 9$. 
2.5.3 The Tradeoff Curve

We derive a tradeoff curve for the multiway equijoin problem for solving the max-out distribution instances of size \( IN \). We use our proof template from Section 2.2.4. To simplify our analysis, we assume that \( n \) and \( m \) are constants and omit them in big-oh notation unless they appear in the exponent of some term.

- **Deriving \( g(M) \):** If the reducer gets \( M/n \) tuples from each relation, then by the AGM bound the maximum outputs it can cover is \( (M/n)^{\rho^*} = O(M^{\rho^*}) \). Now assume that the reducer gets \( r_j \) tuples from \( R_j \) instead, where each \( r_j \leq M \). Notice that even if each \( r_j \) is equal to \( M \), by the AGM bound the reducer can cover at most \( M^{\rho^*} \) outputs, which is also \( O(M^{\rho^*}) \). Therefore a reducer that gets \( M \) tuples, no matter how the \( M \) inputs are distributed across the relations, can cover \( \Theta(M^{\rho^*}) \) outputs.

- **Number of Inputs and Outputs:** By construction of the max-out distributions there are \( IN \) inputs and \( (IN/n)^{\rho^*} = O(IN^{\rho^*}) \) outputs.

- **Lower Bound on \( C \):** Finally, we employ the manipulation trick to derive a lower bound on \( C \). For clarity of presentation, we omit the big-oh notation.

\[
\sum_{i=1}^{p} M_i^{\rho^*} \geq IN^{\rho^*} \Rightarrow \sum_{i=1}^{p} M_i M^{\rho^*-1} \geq IN^{\rho^*}
\]

\[
C = \sum_{i=1}^{p} M_i \geq \frac{IN^{\rho^*}}{M^{\rho^*-1}} \tag{2.11}
\]

We have derived a lower bound on the communication cost of algorithms that solve the instances in the max-out distribution of a query \( Q \). Note that this lower bound is trivially a valid lower bound for algorithms solving all instances as well. We also note that, to the best of our knowledge, there is no known better tradeoff curve that is defined over all instances of size \( IN \).
2.5.4 Algorithms

Trivially there is the purely serial algorithm that sends all of the input to one reducer, which is Pareto optimal on an arbitrary instance. However, there is no known algorithm that is close to being Pareto optimal for any other parallelism level on an arbitrary input. Instead we describe two algorithms, the Shares algorithm from reference [4] and the Grouping algorithm, which can be Pareto optimal or close in two different settings: (1) If we limit ourselves to solving instances of the max-out distribution, then Shares can be parameterized to be Pareto optimal for a wide range of values; and (2) On an arbitrary input, if the edge covering number of \( Q \) is close to \( n \), i.e., \( \rho^* \approx n \), then Grouping can be close to Pareto optimal for a wide range of parallelism levels. We describe each algorithm next.

Shares: We give a brief description of the algorithm here and refer the reader to reference [4] for details. The Shares algorithm takes as input a number of reducers \( p \) and a set of shares \( s_i \) for each attribute \( A_i \) such that \( p = s_1s_2\ldots s_m \). Each reducer is indexed from \( r_{s_1,1}, r_{s_2,1}, \ldots, r_{s_m,1} \) to \( r_{s_1,s_2}, \ldots, r_{s_m} \). The algorithm uses \( m \) hash functions \( h_1, h_2, \ldots, h_m \), where \( h_i \) hashes values from \( |D| \) to \( [1, s_i] \). Suppose relation \( R_j \) has \( k \) attributes \( A_{1j}, A_{2j}, \ldots, A_{kj} \). The algorithm maps each tuple \( t = (a_{j1}, a_{j2}, \ldots, a_{jk}) \) to all reducers with indices \( j_1 = h_1(a_{j1}), j_2 = h_2(a_{j2}), \ldots, j_k = h_k(a_{jk}) \). This guarantees that the input tuples that make up each output tuple \( o = (a_1^*, a_2^*, \ldots, a_m^*) \) are covered by reducer \( r_{h_1(a_1^*), h_2(a_2^*), \ldots, h_m(a_m^*)} \). For example, consider the Chain-2 join and suppose \( p \) is 30 and \( s_1, s_2, \) and \( s_3 \) are 5, 3, and 2, respectively. Consider the \( R_1 \) tuple \((7, 9)\) and \( R_2 \) tuple \((9, 6)\). Further suppose \( h_1(7) = 4, h_2(9) = 1, \) and \( h_3(6) = 2 \). Tuple \((7, 9)\) will be sent to reducers \( r_{4,1,1} \) and \( r_{4,1,2} \). Tuple \((9, 6)\) will be sent to reducers \( r_{1,1,2}, r_{2,1,2}, r_{3,1,2}, \) and \( r_{4,1,2} \). Therefore, both tuples will meet at reducer \( r_{4,1,2} \), which can produce the output \((7, 9, 6)\).

Notice that the reducer size and communication cost of Shares depends on the actual inputs and how the Shares are picked. Consider the max-out distribution of a query \( Q \) with input size \( IN \) and vertex packing \( \bar{v}^* = (v_1^*, \ldots, v_m^*) \). Let \( \gamma^* \) be the value of \( \bar{v}^* \) as usual. Recall that \( \gamma^* \) is equal to the edge covering number \( \rho^* \) of \( Q \). Our next lemma shows that Shares can be parameterized to be Pareto optimal for a wide range of parallelism levels on this max-out distribution.
Lemma 2.5.1. On the max-out distribution with $\bar{v}^* = (v^*_1, \ldots, v^*_m)$, Shares with each $s_i = \frac{v^*_i}{\rho^*}$, with high probability, assigns $M = O\left(\frac{\IN}{p\rho^*}\right)$ inputs per reducer and incurs a communication cost of $O\left(\frac{\IN\rho^*}{M^{\rho^* - 1}}\right)$.

Proof. Note that with $s_i = \frac{v^*_i}{\rho^*}$, $s_1s_2\ldots s_m$ is equal to $p$, since $\Sigma v^*_i = \rho^*$. Recall that by construction, on the max-out distribution, there are $\pi_{i \in \text{attrs}(R_j)}\IN v^*_i = \IN \Sigma_{i \in \text{attrs}(R_j)} v^*_i$ tuples in $R_j$. For simplicity, we assume that we have perfect hash functions. Otherwise, our results can be restated to be true with high probability by standard application of Chernoff bounds. Then any reducer $r$ on average will get $\frac{\IN \Sigma_{i \in \text{attrs}(R_j)} v^*_i}{\pi_{i \in \text{attrs}(R_i)} p^*}$ of the tuples in $R_j$. Replacing $s_i$ by $\frac{v^*_i}{\rho^*}$, we see that each reducer on average gets

$$\frac{\IN \Sigma_{i \in \text{attrs}(R_j)} v^*_i}{\pi_{i \in \text{attrs}(R_i)} p^*} = \frac{\IN \Sigma_{i \in \text{attrs}(R_j)} v^*_i}{\pi_{i \in \text{attrs}(R_i)} p^*} \cdot \frac{\Sigma_{i \in \text{attrs}(R_j)} v^*_i}{\rho^*} = \left(\frac{\IN}{p\rho^*}\right) \Sigma_{i \in \text{attrs}(R_j)} v^*_i$$

Then the total tuples that reducer $r$ gets is equal to:

$$\sum_{j \in [1,n]} \left(\frac{\IN}{p^*}\right) \Sigma_{i \in \text{attrs}(R_j)} v^*_i$$

(2.12)

Notice that for each $R_j$, $\Sigma_{i \in \text{attrs}(R_j)} v^*_i \leq 1$ by definition of vertex packing. However, because we assume that $\bar{v}^*$ is an optimal vertex packing, then at least for one of the relations, $\Sigma_{i \in \text{attrs}(R_j)} v^*_i$ has to be exactly 1, which then dominates the sum in Equation 2.12. Therefore, $M$ is equal to $\frac{\IN}{p^*\rho^*}$. Since there are $p$ reducers, then $C$, i.e., the communication cost, equals $pM = \frac{\IN\rho^*}{p^* \rho^*}$. Solving for $p$ from $M = \frac{\IN}{p^*\rho^*}$, we get that $p = \left(\frac{\IN}{M}\right)^{\rho^*}$, which gives us that $C = \frac{\IN\rho^*}{M^{\rho^* - 1}}$, which is exactly the tradeoff curve we derived in Equation 2.11, completing our proof.

Grouping: We next describe the general version of the Grouping algorithm we demonstrated on the Chain-3 join in Section 2.2.2. Let $p$ be the number of reducers. The algorithm divides each relation into $p^{\frac{1}{\rho^*}}$ groups of size $\frac{\IN}{np^{\frac{1}{\rho^*}}}$. The reducers are indexed from $r_{1,1,\ldots,1}$ to $r_{p^{\frac{1}{\rho^*}},p^{\frac{1}{\rho^*}},\ldots,p^{\frac{1}{\rho^*}}}$. Group $k$ of relation $R_j$ is sent to all reducers whose $j$th index is equal to $k$. Consider an output $o$ that depends on tuples $t_1, \ldots, t_n$.
from each relation $R_j$, which are assigned to groups $g_1, \ldots, g_n$. Then $o$ will be covered by reducer $r_{g_1,\ldots,g_n}$, so the Grouping algorithm is correct. The reducer size $M = \frac{IN}{p\pi}$ because each reducer gets $\frac{IN}{np\pi}$ tuples from each of the $n$ relations. Therefore $p = \frac{IN^n}{Mn}$, and therefore the communication cost is equal to $C = pM = \frac{IN^n}{Mn^2}$. Compared to the tradeoff curve $C = \frac{IN\rho^*}{M\rho^* - 1}$, we can see that the Grouping algorithm is Pareto optimal for essentially all parallelism levels when $\rho^* = n$. Examples of queries whose edge covering numbers are equal to $n$ include the Chain-2 join, Star-n join, or any Cartesian product of $n$ relations. In general, as long as $\rho^* \approx n$, the Grouping algorithm performs well on every parallelism level. We emphasize that the Grouping algorithm’s optimality is on arbitrary instances, and not just on a max-out distribution of the query when $\rho^* = n$. However, as we will discuss in Chapter 4, if we assume a different, potentially easier to solve distribution of instances, the Grouping algorithm’s performance can be suboptimal.

Finally, we note that in reference [19], Beame et al. has also shown that the shares of the Shares algorithm can be picked to yield a communication cost of $\frac{IN^m}{M^{m-1}}$ on arbitrary instances, which performs well on every parallelism level for queries with $\rho^*$ values equal or close to $m$. For example, consider a query $Q$ that consists of all of the $\binom{m}{2}$ binary relations one can form out of $m$ attributes. The edge covering number of this query is equal to $m/2$, while $n = O(m^2)$. Therefore Shares parameterized as described in reference [19] would perform much better than the Grouping algorithm for this query.

2.6 Related Work

We divide related work into models of MR, MR algorithms, and other work that relates to the background material for this chapter.

2.6.1 Models of MapReduce

An interesting model of computation that is inspired by MR is the Massively Unordered Distributed (MUD) model from reference [43]. In this model, algorithms are
only allowed to solve a computational problem by performing a binary merge operation between two arbitrary inputs in iterations. MUD is more constrained than general MR algorithms, as in MR, reducers are not constrained to binary merge operations and can perform any computation on all of the inputs they receive. MUD can be thought of a distributed streaming model, where instead of one machine that can process the entire stream, there are multiple machines that process different partitions of the stream. The focus of reference [43] is to compare the power of streaming model over the MUD model.

Karloff et al. [77] has proposed the first theoretical model directly for analyzing algorithms in MR. They define a computational complexity class called $\mathcal{MRC}^i$, which can be computed in $\log^i$ rounds of MR. In addition, the size and the number of reducers that algorithms are allowed to use is sublinear in the size of the input to problems. They focus on showing concrete algorithms to show that several problems are in $\mathcal{MRC}^0$ or $\mathcal{MRC}^1$, i.e., can be executed in constant or logarithmic number of rounds.

The work closest to our setting is the $\mathcal{MPC}$ model, for massively parallel communication, introduced by Beame, Koutris, and Suciu to study the complexity of multiway equijoins in a series of papers [18, 19, 80]. In the $\mathcal{MPC}$ model, there are $p$ machines. Each machine initially holds $\frac{IN}{p}$ of the tuples of the input relations. Each machine performs some local computation on its partition and sends messages to other machines in synchronous rounds. The constraint is that each machine is allowed to receive at most $\frac{IN}{p^{1-\epsilon}}$ number of tuples, where $\epsilon$ is called the space exponent of the model. Proving bounds on the space exponent $\epsilon$ for evaluating different queries imply lower bounds on the communication cost of algorithms for different reducer sizes, i.e., a tradeoff curve. However, their model is slightly stronger than our model because each machine is allowed to look at its entire partition of data before the first round starts. Algorithms can use this extension to estimate the frequency of each attribute value. We next review each of the papers written by the authors:

- In reference [80], the authors have characterized queries that can be evaluated in an “embarrassingly parallel” fashion, i.e., with $C = IN$ for any parallelism level, in one round.
• In reference [18], the authors study the tradeoff for evaluating queries in one round when the instances of the query come from a distribution called matching databases. Matching databases represent instances without any skew. They also show how to pick the shares of the Shares algorithm to execute in a Pareto optimal fashion over matching databases for different parallelism levels. Our results were derived for the max-out distribution, which essentially represents inputs with maximum skew. For example, for the Chain-2 join, each instance of the max-out distribution contains a single $A_2$ value. While our tradeoff curves relate to the optimal edge coverings of queries, the tradeoff curves derived in this work relate to the optimal edge packings of queries. A query’s edge packings are always smaller than their edge coverings. Therefore a good interpretation of these results is that systems require more communication to solve skewed instances of queries than instances without skew, i.e., skew makes problems “harder” to parallelize. In this paper, authors also derive tradeoff curves for multiround algorithms for a subclass of queries called tree-like queries under matching databases.

• In reference [19], the authors study the tradeoff of evaluating queries under any skew in the input distribution in one round. Essentially, between matching databases and max-out distribution we defined, there is a spectrum of skew assumptions one can make and the author rigorously study the effects of different amounts of skew.

We note that no prior work derives tradeoff curves for fuzzy joins or matrix multiplication for one-round MR computations.

2.6.2 MapReduce Algorithms

• Matrix Multiplication: Reference [102] studies the problem of matrix multiplication in MR. They derive lower bounds on the number of MR rounds required to multiply two dense or two sparse matrices as a function of different reducer sizes and cumulative memory available in the cluster. Although their results yield lower bounds for constant number of rounds, they are not at the granularity to yield lower bounds for one-round MR algorithms.
• **Fuzzy Joins:** A number of recent works have explored MR algorithms for fuzzy joins. Usually, the notion of similarity is that the two elements are within distance $d$ according to some distance measure. Reference [3] presents several one-round algorithms for solving fuzzy joins under Hamming, Edit, and Jaccard distances. The Generalized Splitting algorithm we described in Section 2.4.2 is a generalization of the Splitting algorithm from reference [3]. Reference [125] presents a one-round MR algorithm that identifies similar records based on the Jaccard similarity of sets, using the length/prefix-based methods of [30], combined with the positional and suffix filtering techniques of reference [129]. Reference [17] shows improvements over reference [125] by using two MR jobs rather than one. Finally, reference [113] gives multiround algorithms for fuzzy joins.

• **Multiway Equijoins:** The one-round Shares algorithm for evaluating multiway equijoins was introduced in reference [4]. In the original description of the algorithm, the shares of each attribute was picked to minimize the communication cost assuming that input tuples were drawn uniformly at random from domain $D$ and the size of input tables were not necessarily equal. There has also been several other studies that design multiple round query plans that break a multiway equijoin into multiple binary joins [49, 128]. In Chapter 4, where we study multiround algorithms for multiway equijoins, we review related work on multiround join algorithms in more detail.

### 2.6.3 Other Related Work

We have covered two structural properties of equijoin queries called their edge covers and vertex packings. References [51, 58] provide more details on these notions and other structural properties of queries.
Chapter 3

Fuzzy Joins With Hamming and Edit Distance $d > 1$

Recall from Chapter 2 that in the fuzzy-joins problem, we are given a corpus of strings, and with each string some unknown associated data. Our goal is to execute a user-defined function (UDF) for each pair of strings that are at distance at most $d$ for some distance measure. In Chapter 2, we looked at the tradeoff curve and one-round MapReduce (MR) algorithms for “HD-1”, where the distance measure was Hamming distance and $d$ was 1. In this chapter, we study fuzzy joins for larger Hamming distances, and also for edit distances. Unfortunately, for larger Hamming distances and for any edit distance, we cannot derive tight tradeoff curves. However, we will describe an algorithm, called the Generalized Anchor Points algorithm (GAP), to solve fuzzy joins under large Hamming and edit distances. GAP improves and generalizes the Anchor Points (AP) algorithm from reference 3, which relies on finding a set of strings, called anchor points, with the property that all strings in the input corpus have small distance to some anchor point. The problem of finding a set of anchor points for Hamming distance has been studied under the term “covering codes.” Although there are existing covering codes for Hamming distance, no covering codes for edit distance have been proposed. The main technical contribution of this chapter is to show the construction of the first edit-distance covering codes.

Computing fuzzy joins under larger Hamming and edit distances in one round of
MapReduce have been studied previously in reference \[3\]. Figure 3.1 shows the reducer size and communication costs of existing algorithms for Hamming distance, subject to some assumptions we will explain later. As usual, the x-axis (M) is the reducer size (or parallelism level) and the y-axis (C) is the communication cost. The different points drawn are the communication costs of different algorithms at different reducer sizes. The Naive, Splitting, Ball-Hashing-2, and the Anchor Points algorithms in the figure are from reference \[3\]. The Splitting algorithm extend the one we described in Chapter 2 for Hamming distance 1 to distances greater than 1. All of these algorithms, including the Generalized Anchor Points algorithm we will describe in this chapter, can also run under edit distance with slightly different performance. So, a figure with the same overall structure as Figure 3.1 can be drawn for edit distance.

The main shortcoming of previous algorithms is that, aside from the Naive algorithm, they are designed for a single parallelism level, thus the single point shown in Figure 3.1. Recall from Chapter 2 that an important feature of good MR algorithms is flexibility in parallelism levels: Algorithms that can be parameterized to execute at many parallelism levels allow users to pick different parallelism levels for different clusters. The Naive algorithm provides this flexibility, but is very inefficient.
The GAP algorithm provides flexibility in parallelism in an efficient way. Specifically, GAP supports \( n \) different parallelism levels, where \( n \) is the length of the input strings, and is more efficient than the Naive algorithm for each parallelism level at which it can execute.

We begin this section by defining the problem of fuzzy joins under larger Hamming and edit distances in Section 3.1. In Section 3.2, we give an overview of the Naive algorithm as a comparison point against the AP and GAP algorithms. In Section 3.3, we review the notion of covering codes and describe GAP, which generalizes and improves the AP algorithm. Section 3.4 discusses a method to explicitly construct covering codes for longer strings, if we already have codes for shorter strings. This method can be used to construct new codes for both Hamming and edit distance. The bulk of this chapter is devoted to Sections 3.5–3.7, where we describe our covering-code constructions for edit distance. Finally, in section 3.8, we review related work on covering codes.

### 3.1 Problem Definition

**Hamming Distance \( d \) (HD-d):** The problem setup is similar to the setup for the HD-1 problem from Chapter 2. We are given as input a corpus of all bit strings of length \( n \), and with each bit string some unknown associated data. Our goal is to execute a UDF on the associated data for each pair of strings \((s, t)\) that are at Hamming Distance at most \( d \), i.e., \( s \) and \( t \) differ in at most \( d \) bits.

**Edit Distance \( d \) (ED-d):** We are given as input a corpus of all strings of length \( n \) over some alphabet \( \Gamma \), and with each string some unknown associated data. Throughout our analyses we assume that \( |\Gamma| = a \) and without loss of generality the letters in \( \Gamma \) are the integers from 0 to \((a - 1)\). Our goal is to execute a UDF for all pairs of strings \( u \) and \( w \) that are at edit distance \( d \) for some even \( d \), i.e., \( u \) can be turned into \( w \) by a combination of \( d/2 \) insertions and \( d/2 \) deletions. Figure 3.2 shows the bipartite input-output representation of the ED-2 problem for strings of length 6 over an alphabet of size 4. For example, the output contains the pair \( w = 013011 \) and \( u = 030112 \), because we could delete the first 1 in \( w \) and get \( z = 03011 \) and then
add the letter 2 to the end of z to construct u. A natural application of edit-distance fuzzy joins is in genetics: The four letters 0-3 could correspond to the the four nucleobases A, C, G, T in the DNA, and each string w could correspond to the possible code segments in a particular location of the genome. Finally, the associated data for string w could be the list of species that contain w in their genomes.

3.2 The Naive Algorithm

The Naive algorithm is very similar to the Grouping algorithm we described for the multiway equijoins problem in Chapter 2. Since $|\Gamma| = a$, there are $a^n$ input strings of length n. We divide the input strings into $g \in [2, a^n]$ groups of size $\frac{a^n}{g}$. We have $\binom{g}{2}$ reducers $r_{i,j}$ s.t $1 \leq i < j \leq g$ and an additional $g$ reducers $s_1, \ldots, s_g$. We send each group i to all reducers that contain i in its index. This mapping guarantees that every pair of input strings meet in one reducer. Trivially, it also guarantees that every pair of strings that are at HD-d, or ED-d meet in one reducer. We next calculate the reducer size and communication cost of the Naive algorithm:

- **M:** Reducers $r_{1,1}, \ldots, r_{g-1,g}$ get two groups of inputs of size $\frac{a^n}{g}$ and reducers $s_1, \ldots, s_g$ get only a single group. Therefore, the reducer size $M$ is $\frac{2a^n}{g}$.
- **C:** The communication cost is $\frac{a(a-1)a^n}{2} + g\frac{a^n}{g} = \frac{(a+1)a^n}{2}$. By substitution $g$ for $\frac{2a^n}{M}$, we see that the communication cost of the Naive algorithm is $\approx \frac{a^{2n}}{M}$. 

Figure 3.2: Input/output relationship for ED-2.
3.3 Generalized Anchor Points Algorithm and Covering Codes

The Naive algorithm is very inefficient because it compares every pair of input strings regardless of their distance. GAP does much better by comparing significantly fewer strings. The main building block of GAP is a covering code. An \((n, k)\) covering code is a set \(A\) of strings (called codewords) over an alphabet of size \(a\) such that every string of length \(n\) is within distance \(k\) of some codeword in \(A\). We call \(k\) the radius of code \(A\). The length of the codewords can be different for different distance measures. Under Hamming distance, the codewords are also of length \(n\). However, in edit distance the codewords can be shorter or longer than \(n\), as we allow insertions and deletion to the strings.

**Example 3.3.1.** Consider Hamming distance when the strings are constructed from the binary alphabet \(\{0, 1\}\), i.e., \(a = 2\). Then, \(A = \{00000, 11111\}\) is a \((5, 2)\) covering code. Notice that any bit string of length 5 either has at most two 1’s, in which case it is distance at most 2 from 00000, or it has at most two 0’s, in which case it is at distance 2 or less from 11111.

The GAP algorithm generalizes the AP algorithm from reference 3. We first give a very brief overview of the AP algorithm. Let \(A\) be an \((n, d)\) covering code. AP

| \(n\) | \(k\) | \(|A|\) | Explicit/Existence |
|------|------|-------|-------------------|
| \(n \in \mathbb{N}\) | 0    | \(2^n\) | Explicit: contains all strings of length \(n\). |
| \(n \in \mathbb{N}\) | \(n\) | 1     | Explicit: contains an arbitrary string. |
| \(2^r - 1\) for \(r \in \mathbb{N}\) | 1    | \(\frac{2^n}{n+1}\) | Explicit: the Hamming code. |
| \(n \in \mathbb{N}\) | \(k \leq n\) | \(\frac{n2^n}{B(k)}\) | Existence |

Table 3.1: Hamming distance covering codes.
uses $A$ to find pairs of strings at distance $d$ as follows. There is one reducer for each codeword in $A$. The mappers send each string $w$ to the reducer for each codeword at distance at most $2d$ from $w$. Each reducer then searches for strings at distance up to $d$ from each other, among the strings it has received. Reference [3] has shown that sending each string to codeword that are at distance $2d$ guarantees that each pair of strings that are at distance $d$ is covered by at least one codeword. GAP improves on AP in four ways:

1. We generalize the algorithm to work with any $(n, k)$ code, as opposed to fixed $(n, d)$ codes. This generalization gives the algorithm the flexibility to execute at $n$ different parallelism levels.

2. We show that it is possible to guarantee that two strings meet at one reducer by sending each string to all codewords at distance $3d/2$ as opposed to $2d$. This optimization decreases the communication cost of the algorithm for each reducer size.

3. We describe a general method to explicitly construct small codes for radius $k > 1$ and length $n$ for any distance measure, rather than relying on the nonconstructive existence proof in reference [3]. Our method, called the cross-product method, assumes codes of smaller radius $k' < k$ and shorter length $n' < n$ are known. For Hamming distance, we can construct near-optimal sets of codes for radius $k > 1$ and any $n$ by using the Hamming codes for shorter length strings and the cross-product method.

4. We show the existence and explicit construction of the first non-trivial edit distance covering codes for radius values 1 and 2. Using our cross-product method, we can use these codes to construct edit-distance codes for larger radius values. Our edit distance covering codes make it possible to use GAP to solve the ED-d problem efficiently.

We next explain the GAP algorithm. Let $A$ be an $(n, k)$ covering code. As before, there is one reducer for each codeword in $A$. The mappers now send each input string
to all the codewords at distance at most \( k + d/2 \) from \( w \). The reducers find all pairs of received strings that are at distance up to \( d \). The proof that all pairs of distance \( d \) are covered by one reducer follows from the triangle inequality, as we next prove:

**Theorem 3.3.1.** If \( A \) is an \((n,k)\) covering code, then any two strings that are within distance \( d \) are within distance \( k + d/2 \) from some codeword in \( A \), assuming the triangle inequality holds for the given distance measure.

**Proof.** Let \( w \) and \( x \) be the two strings at distance \( d \). Then, we may find a \( y \) at distance \( d/2 \) from both \( w \) and \( x \). For example, under Hamming distance we can do the following: we start with \( w \), choose any \( d/2 \) bits where \( w \) and \( x \) disagree, and flip those bits in \( w \) to agree with \( x \), and get to a string \( y \) that is at distance \( d/2 \) from both \( w \) and \( x \). Since \( A \) is a covering code, there is a member of \( C \), say \( z \), at distance at most \( k \) from \( y \). By the triangle inequality, \( w \) and \( x \) are each within \( k + d/2 \) distance from \( z \). \qed

**Example 3.3.2.** Let \( n \) be 5, \( k \) and \( d \) be 2, and the distance measure be Hamming distance. Then, we can use the \((5,2)\) code \( A = \{00000, 11111\} \) from Example 3.3.1. Let \( w = 01010 \) and \( x = 11000 \). Notice that \( w \) and \( x \) differ in their first and fourth bits; so they are at distance 2. Notice that in this case, there are two possible middle strings \( y \) that are at distance 1 from both \( w \) and \( x \): (1) 11010; and (2) 01000. Let \( y \) be 11010, which is at distance 2 from codeword 11111. Then we get a codeword that is at distance at most \( 2 + 1 = 3 \) from both \( w \) and \( x \). If we pick \( y \) as 01000, which is at distance 1 from codeword 00000, then we get a codeword that is at distance at most 2 from \( w \) and \( x \).

We note that the triangle inequality holds for both Hamming and edit distance, the two distance measures we study in this chapter.

### 3.3.1 Cost Analysis

Notice that GAP has the flexibility to use an \((n,k)\) code for \( k = 0, \ldots, n \). Thus, GAP can run at \( n \) different levels of parallelism. For a fixed value of \( k \), the reducer size of GAP is \( M = B(k + d/2) \). The communication cost of GAP depends on the size of
the code. Since each codeword gets all the strings within its radius of \( k + d/2 \), the communication cost of GAP is equal to \( C = |A|B(k + d/2) \).

### 3.4 Constructing Explicit Codes By Cross Product

In order to use GAP in practice, we need to construct small \((n, k)\) codes for different values of \( k \), alphabet sizes \( a \), and distance measures. Recall from Table 3.1 that for Hamming distance, we have perfect \((n, 1)\) codes for certain values of \( n \). In Section 3.5, we will also show how to construct small \((n, 1)\) and \((n, 2)\) codes for edit distance. In this section, we show how to construct codes with larger radius values \( k \) by using codes for shorter strings and smaller radius values. We call our method the cross-product method. Using the cross-product method, we can build on these codes and construct small \((n, k)\) codes for both Hamming and edit distance. We start with the smallest possible \((n/x, k/x)\) covering code over the given alphabet and extend it as follows.

**Theorem 3.4.1.** Let \( A \) be any \((n/x, k/x)\) covering code over an alphabet of size \( a \). Let \( A' = A^x \) be the set of strings that are formed by taking the cross-product of \( A \) with itself \( x \) times. That is, we concatenate the strings in \( A \) \( x \) times in all combinations allowing repetition. Then \( A' \) is a \((n, k)\) covering code over the same alphabet.

**Proof.** Given a string \( w \) of length \( n \), write \( w = w_1w_2 \cdots w_x \), where each \( w_i \) is of length \( n/x \). Then, by definition of \( A \), there is a codeword \( c_i \) in \( A \) that is at distance at most \( k/x \) from \( w_i \). Notice that the concatenation of \( c_1c_2 \cdots c_x \) is a string in \( A' \), and is at distance at most \( k \) from \( w \). \( \square \)

We note that Theorem 3.4.1 holds for any distance measure in which given two strings \( w = w_1w_2 \cdots w_x \) and \( u = u_1u_2 \cdots u_x \) of length \( n \), where each \( w_i \) and \( u_j \) is of length \( n/x \), \( \text{dist}(w, u) \) is at most the sum of the distances of \( \text{dist}(w_i, u_i) \), i.e., \( \text{dist}(w, u) \leq \sum_i \text{dist}(w_i, u_i) \). This property holds for both Hamming and edit distance.

**Example 3.4.1.** Let \( n = 28 \), \( k = 4 \), and \( a = 2 \), and the distance measure be Hamming distance. Let \( x \) be equal to 4 as well. Then \( n/x = 7 = 2^3 - 1 \), so \( m = 3 \). There
is a Hamming code of length $2^3 - 1 = 7$, with $2^{(n/2)^m} = 16$ members. Thus, there is a covering code for $n = 28$ and $k = 4$ with $16^4 = 2^{16}$ members. That is, fraction $2^{-km} = 2^{-12}$, or $1/4096$ of the $2^{28}$ bit strings of length 28 is in the covering code constructed by Theorem 3.4.1. In comparison, a hypothetical $(28, 4)$ perfect code, which is not necessarily attainable, states that one in $\sum_{i=0}^{4} \binom{28}{i}$, or $1$ in $24,158$ of the binary strings of length 28 must be in any covering code for $n = 28$ and $k = 4$.

In general, if we start with a perfect $(n/k, 1)$ covering code, the size of the code constructed by the cross-product method would be of size $\left(\frac{a^{n/k}}{B_{n/k}(1)}\right)^k = \frac{a^n}{B_{n/k}(1)^k}$. In contrast, a perfect $(n, k)$ code would have a size of $\frac{a^n}{B_n(k)}$. Where $B_{n/k}(1)$ is the size of the ball of radius 1 for strings of length $n/k$. Similarly, $B_n(k)$ is the size of the ball of radius $k$ for strings of length $n$. Therefore the ratio of the code constructed by the cross-product method and an hypothetical perfect code is $\frac{B_n(k)}{B_{n/k}(1)^k}$.

Example 3.4.2. Assume we are looking at Hamming distance over binary strings and $k = 2$. A hypothetical $(n, 2)$ code would have a size of $\frac{2^n}{B_n(2)} = \frac{2^n}{(n^2 + n + 1)/2}$. In contrast, the code constructed by the cross-product method starting with an $(n/2, 1)$ perfect code would have a size of $\frac{2^n}{B_{n/2}(1)^2} = \frac{2^n}{(n/2 + 1)^2} = \frac{2^n}{(n^2 + 4n + 4)/4}$. This quantity is less than twice the size of the hypothetical $(n, 2)$ code.

3.5 Edit-Distance Covering Codes

We can use GAP to solve the ED-d problem if we can construct covering codes for edit distance. Unlike Hamming distance, with edit distance, we can cover strings by using insertions, deletions, or a combination of these. We shall focus on covering codes that cover strings of a fixed length, using only insertions or only deletions, so the covering code itself has strings of a fixed length.

Definition 3.5.1. (Insertion-k Covering Code): A set $A$ of strings of length $n + k$ is an insertion-$k$ covering code for length $n$, distance $k$, and alphabet $\Gamma$ if for every string $w$ of length $n$ over $\Gamma$ we can insert $k$ characters from $\Gamma$ into $w$ and produce some string in $A$. Equivalently, for every $w$ of length $n$ we can find some string $x$ in
A such that it is possible to delete $k$ positions from $x$ and produce $w$. We say that $x$ covers $w$ in this case.

**Definition 3.5.2. (Deletion-$k$ Covering Code):** Similarly, we say a set $A$ of strings of length $n - k$ is a deletion-$k$ covering code for length $n$, distance $k$, and alphabet $\Gamma$ if for every string $w$ of length $n$ over $\Gamma$ we can delete $k$ positions from $w$ and produce some string in $A$. Again, we say that $x$ covers $w$ if so.

Recall that we assume that $|\Gamma| = a$ and w.l.o.g. the letters in $\Gamma$ are the integers from 0 to $(a-1)$. Finding covering codes for edit distance is harder than for Hamming distance, since there is no convenient “perfect” code like the Hamming codes to build from. One tricky aspect of working with edit distance is that certain deletions and insertions have the same effect. For instance, deleting from any of the three middle positions of 01110 yields 0110. When we want to develop a covering code, this phenomenon actually works against us. For example, under Hamming distance, when $n = 5$ and $k = 1$ and we have the binary alphabet, each string can be covered by $n$ other strings. However, in edit distance, if we want a deletion-1 code for $n = 5$, then 00000 requires us to have 0000 in the code, since every deletion of one position from 00000 yields 0000. Likewise, the code must have 1111; there are no options.

### 3.5.1 Elementary Lower Bounds

There are simple arguments that say a covering code cannot be too small; these are obtained by giving an upper bound on the number of strings one codeword can cover. For example, [3] shows that a string of length $n - 1$ over an alphabet of size $a$ yields exactly $n(a - 1) + 1$ strings by a single insertion. That observation gives us a lower bound on the size of a deletion-1 code for strings of length $n$. Such a code must contain at least

$$\frac{a^n}{n(a - 1) + 1}$$

strings.

For insertion-1 codes, different strings of length $n + 1$ can cover different numbers of strings of length $n$. The number of strings covered is the number of runs in the
string, where a run is a maximal sequence of identical symbols. For example, the string 00000, which has only one run, can cover only one string, 0000. In contrast, the string 01010 can cover four different strings: 1010, 0010, 0110, 0101. Surely, a string of length $n + 1$ can have no more than $n + 1$ runs. Thus, an insertion-1 code for strings of length $n$ must have at least $a/(n + 1)$ strings.

### 3.5.2 Summary of Results

Our results are summarized in Table 3.2. In the table and the rest of this chapter, we specify code sizes as fractions of the number of strings of length $n$. For example, the $a/(n + 1)$-size insertion-1 code of the first row of Table 3.2 contains $a/(n + 1)$ fraction of all strings of length $n$, i.e., exactly $aa^n/(n + 1) = a^{n+1}/(n + 1)$ codewords.

Section 3.6 begins by summarizing our proof template for explicitly constructing covering codes. In Section 3.6.1 we describe our explicit construction of insertion-1 covering codes. In Section 3.6.2 and Section 3.6.3 we give explicit constructions of deletion codes for distances 1 and 2, that are of size $O(1/a^2)$ and $O(1/a^3)$, respectively. Finally, in Section 3.7 we prove the existence of $O(\log(n)/n)$-size deletion-1 codes—a major improvement over our result from Section 3.6.2 for long strings. However, note that the existential upper bound we offer is greater by a factor of $O(a \log n)$ than the lower bound from Section 3.5.1. We note that depending on the actual values of $a$ and $n$, any of the codes we present here can be the smallest and yield the best performance for GAP.

| Insertion/Deletion | $|A|$ | Explicit/Existence |
|--------------------|------|---------------------|
| insertion-1        | $a/(n + 1)$ | explicit            |
| deletion-1         | $O(\log(n)/n)$ for $\frac{n}{\log(n)} \geq 48a$ | existence            |
| deletion-1         | $O(1/a^2)$ for $n \geq 3a \log(a)$ | explicit            |
| deletion-2         | $O(1/a^3)$ for $n \geq \frac{a}{2} + \log(a)$ | explicit            |

Table 3.2: Summary of edit-distance covering codes.
3.6 Explicit Construction of Edit-Distance Covering Codes

Let \( w = w_n w_{n-1} \cdots w_1 \) be a string of length \( n \) over an alphabet \( \Gamma \) of size \( a \), and let \( A \) be the edit-distance covering code we are constructing. We first outline the proof template we use to construct \( A \):

1. **Sum value**: Assign each string \( w \) a *sum* value \( \text{sum}(w) \), the value of applying some function to \( w \), treating each of its positions \( w_i \) as an integer (recall we assume the symbols of the alphabet are integers 0, 1, \ldots, \( a - 1 \)).

2. **Modulus value**: Pick an appropriate integer \( c \) and let \( \text{score}(w) = \text{sum}(w) \mod c \).

3. **Residues**: Pick one or more residues modulo \( c \). Put into \( A \) all strings of appropriate length (e.g \( n + 1 \) for insertion-1 codes or \( n - 1 \) for deletion-1 codes), whose score values are equal to one of the residues.

We then count the strings in \( A \) and prove that \( A \) covers all strings of length \( n \). In some cases, we do not cover all strings with \( A \). Rather, we show that the number of strings not covered (called *outliers*) is small compared to the size of \( A \). We can then argue that by adding one codeword into \( A \) for each outlier string, we can construct an extended code \( A' \) that covers all strings and that has the same asymptotic size as \( A \).

### 3.6.1 Insertion-1 Covering Codes

We follow the proof template above to construct an insertion-1 covering code:

- **Sum value**: \( \text{sum}(w) = \sum_{i=1}^{n} w_i \times i \)
- **Modulus value**: \( c = (n + 1) \times (a - 1) \)
- **Residues**: Any \( a - 1 \) consecutive residues, \( \{(i \mod c), (i + 1 \mod c), \ldots, (i + (a - 2) \mod c)\} \). For example, if \( a = 4 \) and \( n = 5 \), then \( c = 18 \). As residues, we can pick 2, 3, 4 or 17, 0, 1.
Before we prove that the code we constructed covers every string of length $n$, we give an example:

**Example 3.6.1.** Let $a = 4$, $n = 5$, and assume we pick 8, 9, and 10 as our residues. Then our code consists of all strings of length 6, whose score values equal 8, 9, or 10. Consider the string 23010. Then we can insert 0 between the fourth and fifth digits (3 and 2), and produce 203010, which is a codeword since its sum value is $0 \times 1 + 1 \times 2 + 0 \times 3 + 3 \times 4 + 0 \times 5 + 2 \times 6 = 26$ and score value is 8. Similarly consider the string of all zeros: 00000. We can insert 3 between the second and third digits, and produce 000300, which also is a codeword as it has a score of 9.

It is not a coincidence that we were able to take a string $w$ of length 5 and generate a codeword by inserting a 0 or a 3 into $w$. As we prove momentarily, our code has the property that every string $w$ of length $n$ is covered by inserting one of the symbols 0 or $a - 1$ somewhere in $w$.

Consider a string $w$ of length $n$. Let $\text{sum}X_j$ and $\text{score}X_j$, for $j = n + 1, \ldots, 1$ be the sum and score values, respectively, of the string that is constructed by adding 0 to the left of $w_{j-1}$. If $j = 1$, we add 0 at the right end. Similarly, let $\text{sum}Y_j$ and $\text{score}Y_j$ be the sum and score values, respectively, of the string constructed by adding $(a - 1)$ to the left of $w_{j-1}$, or at the right end if $j = 1$. For example, for the string 23010, $\text{sum}X_3$ is the sum value of the string 230010 (the second 0 is the one inserted) and is equal to 29. $\text{score}X_3$ is then 29 $\text{mod}$ 18 = 11. Similarly, $\text{sum}Y_1$ is the sum value of the string 230103 and is equal to 33, and $\text{score}Y_1$ is 33 $\text{mod}$ 18 = 15.

**Lemma 3.6.1.** (i) $\text{sum}Y_{n+1} - \text{sum}X_{n+1} = (n + 1)(a - 1)$ (ii) $\text{sum}Y_1 - \text{sum}X_1 = (a - 1)$.

**Proof.** (i) Let $u = (a - 1)w_n \cdots w_1$ and $v = 0w_n \cdots w_1$. $u$ and $v$ differ only in the $(n + 1)$st digit. Therefore the difference between $\text{sum}(u)$ and $\text{sum}(v)$ is exactly $(n + 1) \times (a - 1)$.

(ii) Let $z = w_n \cdots w_1(a - 1)$ and $t = w_n \cdots w_10$. $z$ and $t$ differ only in the first digit. Therefore the difference between $\text{sum}(z)$ and $\text{sum}(t)$ is exactly $a - 1$. □

Consider the sequences $\text{sum}X_{n+1}, \text{sum}X_n, \ldots, \text{sum}X_1$ and $\text{sum}Y_{n+1}, \text{sum}Y_n, \ldots, \text{sum}Y_1$ of the sum values produced by inserting a 0 and $(a - 1)$ to the left of each digit in
Figure 3.3: Simulation of insertions of symbols 0 and \((a - 1)\) into strings.

We can visualize these sequences as two walkers, an \(X\) walker and a \(Y\) walker, taking an \(n\)-step walk on the number line. Figure 3.3 shows the walk for the string 23010. In the figure, the top labels of the lines are the sum values and bottom labels are the score values. Note that the \(X\) (\(Y\)) walker being on a position with a particular sum value \(s\) and score value \(r\) corresponds to constructing a string of length six from 23010 by a single insertion of 0 ((a-1)) with sum value \(s\) and score value \(r\). We know from Lemma 3.6.1 that \(\text{sum}_Y_{n+1} - \text{sum}_X_{n+1} = (n + 1)(a - 1)\) and \(\text{sum}_Y_{1} - \text{sum}_X_{1} = (a - 1)\): the walkers start \((n + 1)(a - 1)\) positions away from each other and finish exactly \((a - 1)\) positions away from each other. We will next prove that the walkers always walk in opposite directions in steps of size at most \(a - 1\).

**Lemma 3.6.2.** \(\text{sum}_X_{j} - \text{sum}_X_{j+1} = i\) and \(\text{sum}_Y_{j} - \text{sum}_Y_{j+1} = -(a - 1 - i)\), for some \(i \in 0, \ldots, a - 1\).

**Proof.** Let \(w_{j+1}\) be \(i\). Then

\[
\text{sum}_X_{j} = \text{sum}(w_n \ldots w_{j+2}0w_j \ldots w_1)
\]
\[ \text{sum} X_{j+1} = \text{sum}(w_n \ldots w_{j+20}i w_j \ldots w_1) \]

Notice that the inputs to the sum functions differ only in the \((j+1)\)st and \((j+2)\)nd digits. Subtracting one from another, \( \text{sum} X_j - \text{sum} X_{j+1} = i(j+2) - i(j+1) = i \).

Similarly

\[ \text{sum} Y_j = \text{score}(w_n \ldots w_{j+2}i(a-1)w_j \ldots w_1) \]

\[ \text{sum} Y_{j+1} = \text{score}(w_n \ldots w_{j+2}(a-1)iw_j \ldots w_1) \]

Therefore,

\[ \text{sum} Y_j - \text{sum} Y_{j+1} = \]

\[ i(j+2) + (a-1)(j+1) - [(a-1)(j+2) + i(j+1)] = -(a-1-i) \]

In other words, the sum values are always increasing for walker X and decreasing for walker Y. Moreover, the sum values differ by at most \((a-1)\) for each walker and cumulatively they travel a distance of \((a-1)\). In Figure 3.3, this can be visualized as two walkers at two ends of a line walking towards each other synchronously. At each step, if walker X moves \(i\) amount to the right, then walker Y moves \((a-1-i)\) amount to the left.

**Theorem 3.6.1.** Fix any \((a-1)\) consecutive residues

\[ R = \{i \mod c, i+1 \mod c, \ldots, (i+(a-2)) \mod c\} \]

where \(c = (n+1)(a-1)\). The code \(A\) constructed by taking all strings of length \(n+1\) whose score values are in \(R\) covers all strings of length \(n\) by a single insertion.

**Proof.** Consider any string \(w\) of length \(n\) and the corresponding X and Y walkers for it. We know from Lemma 3.6.1 that the walkers starts exactly \((n+1)(a-1)\) sum values away. Therefore the score values of the numbers between their initial positions cover exactly a full residue cycle of modulo \(c = (n+1)(a-1)\). We also know that they walk in opposite directions (Lemma 3.6.2) and finish the walk exactly
(a − 1) sum values away (Lemma \ref{lem:sum_value}). Since the step sizes of the walkers is \(\leq (a − 1)\) (Lemma \ref{lem:step_size}), neither of the walkers can skip over all the \((a − 1)\) consecutive residues in \(R\) in a single step, which implies that at least one of the walkers must step on one of the residues in \(R\). In other words we can insert 0 or \((a − 1)\) into some position \(j\) of \(w\) and generate a codeword.

**Corollary 3.6.1.** We can construct an \(a/(n+1)\) size insertion-1 covering code \(C\) for strings of length \(n\).

*Proof.* Let \(A_j\) be the code we construct by selecting the \((a − 1)\) residues between \(j(a − 1)\) and \((j + 1)(a − 1)\), for \(j \in 0, \ldots, n\). Note that \(A_j\)'s are disjoint, and every string of length \(n + 1\) belongs to one \(A_j\). We have \(n+1\) disjoint codes and their union has size \(a^{n+1}\) (all strings of length \(n+1\)). Therefore one of the codes must contain at most \(a^{n+1}/n + 1\) strings and is an \(a/n + 1\)-size code.

### 3.6.2 \(O(1/a^2)\)-size Deletion-1 Covering Codes

We next use our proof template to construct an \(O(1/a^2)\) size deletion-1 code, for large enough \(n\).

- **Sum value**: \(\text{sum}(w) = \Sigma_{i=1}^{n} w_i\). That is, the sum value of \(w\) is the sum of the integer values of its digits.
- **Modulus value**: \(c = a\)
- **Residues**: 0

This code covers nearly all strings of length \(n\). Consider a string \(w\) of length \(n\). Let \(\text{score}(w) = i\). If \(w\) has any occurrence of the symbol \(i\), delete it, and you get a codeword. Thus, our code covers all strings that contain their modulus. To make it a covering code, we take any string that is not covered, remove its first digit, and add it to the code. Then any string \(w\) of length \(n\) will either be covered by the original code, or it will be covered by the codeword that we added specifically for it.

To determine the size of our code, we first observe that induction on \(n\) shows that there are \(a^{n-1}\) strings of length \(n\) with score \(r\) for each residue \(r \in \{0, \ldots, a − 1\}\). Thus, in particular, there are \(a^{n-2}\) strings of length \(n − 1\) with score 0, making the
original code a $1/a^2$-size code. We show that the number of strings of length $n$ that are missing their modulus is $O(1/a^2)$. To do so, we exhibit a bound on the size of the set $S$ of strings that are missing at least one symbol, which certainly contains every string that is missing its modulus. Observe that $S = \bigcup_i S_i$, where $S_i$ is the set of strings of length $n$ that do not contain symbol $i$. By the union bound, we have that $|S| \leq \sum_i |S_i|$, and thus it suffices to show that each $|S_i|$ represents an $O(1/a^3)$ fraction of the strings of length $n$. The number of strings that do not contain the symbol $i$ is exactly $(a-1)^n$ which is exactly $(1-1/a)^n$ fraction of all strings. This quantity is at most $e^{-n/a}$ and is bounded above by $1/a^3$ for $n \geq 3a\log(a)$, proving the following result:

**Theorem 3.6.2.** For $n \geq 3a\log(a)$, there is an $O(a^{n-2})$-size deletion-1 code.

### 3.6.3 $O(1/a^3)$-size Deletion-2 Covering Code For Shorter Strings

For our deletion-2 code we use the following scheme.

- **Sum value:** $\text{sum}(w) = \Sigma_{i=1}^n w_i$, as in Section 3.6.2
- **Modulus value:** $c = a$
- **Residues:** 0

Suppose we have a string $x$ of length $n$ and $\text{score}(x) = i$. We need to find a pair of positions of $x$ that sum to $i$ modulo $a$ and delete them both. To start, we assume that $a$ is even; the situation for odd $a$ is very similar and we will discuss it in the end. We can group the integers from 0 to $a-1$ into pairs that sum to $i$ modulo $a$. There is a special case where for some integer $j$, we have $2j = i \mod a$. In that case there are actually two such integers $j_1$ and $j_2$. To see why, assume that $2j_1 = i \mod a$. Then let $j_2 = j_1 + a/2 \mod a$. Then $2j_2$ is also equal to $i$ modulo $a$. In this special case, we group those two integers into one group.

**Example 3.6.2.** Let $a = 6$. Figure 3.4 shows the pairs that sum to $i$ modulo 6. For example, if $i = 1$, then the three groups are $\{0, 1\}, \{2, 5\}$, and $\{3, 4\}$. If $i = 2$, then the three groups are $\{0, 2\}, \{1, 4\}$, and $\{3, 5\}$. Note that $1+1$ and $4+4$ are both equal to 2 mod 6, so we put them into one group.
In general, if \( a \) is even, then the pairs that sum to 0 modulo \( a \) are 0 + 0, \( \frac{a}{2} + \frac{a}{2} \), 1 + (\( a - 1 \)), 2 + (\( a - 2 \)), 3 + (\( a - 3 \)), and so on, until \( (\frac{a}{2} - 1) + (\frac{a}{2} + 1) \). If we want the pairs that sum to \( i \), where \( i \) is even, then we add \( i/2 \) to every integer in this list of pairs. The integers \( i/2 \) and \( (a + i)/2 \), when added to themselves, make \( i \) modulo \( a \), while the other \( \frac{a}{2} - 1 \) pairs of two different integers also sum to \( i \) modulo \( a \).

If we want the pairs of integers that sum to 1 modulo \( a \), we note that these are \( 0 + 1 \), \( 2 + (a - 1) \), \( 3 + (a - 2) \), and so on, until \( (\frac{a}{2} - 1) + (\frac{a}{2} + 1) \). That is, there are \( \frac{a}{2} \) pairs of distinct integers. If we want to find the pairs that sum to \( i \), for odd \( i \), then we add \( (i - 1)/2 \) to each of the integers, and again we get \( \frac{a}{2} \) pairs of distinct integers.

The important point is that regardless of the desired sum \( i \), we can divide the integers modulo \( a \) into \( \frac{a}{2} \) groups. Each group either consists of two distinct integers that sum to \( i \) modulo \( a \) or consist of the two integers that, when added to themselves, yield \( i \) modulo \( a \).

If there are \( k \) positions in the string holding members of the same group, then the probability is at least \( 1 - 2^{-(k-1)} \) that these positions hold two symbols that sum to \( i \) modulo \( a \). First, look at groups composed of two different values that sum to \( i \) modulo \( a \), such as \( \{3, 5\} \) for \( a = 6 \) and \( i = 2 \). All positions belonging to the group are independent (assuming we have chosen a string \( x \) randomly). So each position after the first has probability \( 1/2 \) of disagreeing with the first. That is, the probability that all \( k \) positions hold the same symbol is \( 2^{-(k-1)} \).

For a group that is composed of two symbols each of which, added to itself makes \( i \), such as the group \( \{1, 4\} \) for \( a = 6 \) and \( i = 2 \), then the situation is even better. If
$k = 2$, the probability is $1/2$ that the two positions for that group are the same, but if $k \geq 3$, then we are certain to find two positions that sum to $i$ modulo $a$. If the length of $x$ is $n$, then there are at least $n - (a/2)$ positions of $x$ that are not the first in their group. Thus, the probability that we are unable to find any pair of positions of $x$ that sum to $i$ modulo $a$ is at most $2^{n-(a/2)}$. If $n$ is bigger than $a/2 + \log(a)$, then the number of outliers is at most $1/a$ of the total number of strings of length $n - 2$. Thus, we can expand $C$ to include one codeword for each outlier, proving the following result:

**Theorem 3.6.3.** For $n \geq \frac{a}{2} + \log(a)$, there is an $O(a^{n-3})$-size deletion-2 code.

### 3.7 Existence of $O(\log(n)/n)$-size Deletion-1 Covering Codes

We next show that for sufficiently long strings there are deletion-1 covering codes that are much smaller than the $O(1/a^2)$-size code from Section 3.6.2. The proof of the existence of such codes is much more involved than our previous constructions. Instead of showing the existence of an edit-distance-1 covering code directly, we convert the strings of length $n$ and alphabet size $a$ into binary strings of lengths $\leq n$. We then show the existence of a Hamming-distance-1 covering code $H$ for the converted binary strings and convert $H$ into a deletion-1 covering code $C$ for the original strings.

We begin with a roadmap and proof outline. All the terminology we use in the outline, e.g. “run patterns,” “bits of runs,” or “safe bits” will be defined in the specified sections.

1. Convert each string $w$ of length $n$ to its run pattern, $\text{runs}(w)$ (Section 3.7.1).

2. Convert run patterns of $w$ to a binary string, which we refer to as bits of runs of $w$ (Section 3.7.2).

3. Partition the strings of length $n$ into two groups based on the number of safe bits their bits of runs have: LS (low-safe-bit) and HS (high-safe-bit). The strings
in LS will be the first set of outlier strings for the final code we construct and will be covered separately at the end of our construction (Section 3.7.3).

4. Show the existence of a deletion-1 code $A$ that covers all but $1/n$ fraction of the strings in HS. The remaining $1/n$ fraction of the strings in $HS$ will be the second set of outlier strings. We will construct $A$ from a Hamming-Distance-1 code $H$ for binary strings, which covers the bits of runs of the strings in HS on their safe bits (Section 3.7.4).

5. For each outlier string $s$, put into $A$ the string that we get by removing the first symbol of $s$, and construct a deletion-1 covering code (Section 3.7.6).

6. Count the number of outliers and the total number of strings in $A$. (Section 3.7.6).

### 3.7.1 Step 1: Run Patterns

We view strings as sequences of runs – consecutive positions that hold the same character. The length of a run is the number of consecutive positions that hold the same character. A run pattern (or just “pattern”) is a list of positive integers. Every string $w$ of length $n$ corresponds to exactly one pattern $P$, which is the list of positive integers, the $i$th of which is the length of the $i$th run of $w$. We denote this pattern $P$ by $\text{runs}(w)$. Note that the run pattern of a string has the same length as the number of runs in that string.

**Example 3.7.1.** String $w = 002111100$ consists of four runs, 00, 2, 1111, 00, in that order. The lengths of these runs are 2, 1, 4, and 2, respectively, so $\text{runs}(w) = [2, 1, 4, 2]$.

### 3.7.2 Step 2: Converting Run Patterns into Binary Strings

For the second step of our proof, we need to convert run patterns into bit strings of the same length. Define $\text{bits}(P)$ to be the bit string whose $i$th position holds 0 if the $i$th integer on the list $P$ is even, and holds 1 if the $i$th integer is odd.
Example 3.7.2. If $w = 002111100$, then $\text{runs}(w) = [2, 1, 4, 2]$ and $\text{bits}(\text{runs}(w)) = \text{bits}([2, 1, 4, 2]) = 0100$.

3.7.3 Step 3: Partitioning Strings Based on Safe Bit Counts

Deletion of a symbol from a string $w$ in general can generate a string $v$ with a shorter run pattern, and hence $\text{bits}(\text{runs}(v))$ can be shorter than $\text{bits}(\text{runs}(w))$. For example, deletion of the symbol 2 from 00211100, whose run pattern is $[2, 1, 3, 2]$, generates 00111100, whose run pattern is $[2, 4, 2]$. However, if we delete a symbol from $w$ that belongs to a run of length 2 or more, we will get a string $v$ with the following properties:

- $|\text{bits}(\text{runs}(v))| = |\text{bits}(\text{runs}(w))|$. $v$ has the same number of runs as $w$. Since the size of the bits of runs of a string is equal to the number of runs it has, $|\text{bits}(\text{runs}(v))|$ is equal to $|\text{bits}(\text{runs}(w))|$.
- $\text{bits}(\text{runs}(v))$ and $\text{bits}(\text{runs}(w))$ differ in exactly one bit and hence have Hamming distance 1. The bit in which they differ corresponds to the run from which we removed the symbol.

Example 3.7.3. If we remove one of the 1’s in

$$w = 002111100$$

we get $v = 00211100$. $\text{bits}(\text{runs}(v)) = 0110$, which is at Hamming distance one from $\text{bits}(\text{runs}(w)) = 0100$. Note that because we removed a symbol from the third run of $w$, the two bit strings differ in the third bit.

We call a bit in $\text{bits}(\text{runs}(w))$ a safe bit for $w$, if it corresponds to a run of length $\geq 2$. Consider again the string $w = 002111100$ as an example. Every 0 in $\text{bits}(\text{runs}(w)) = 0100$, is safe, and the bit 1, which corresponds to the second run of $w$ is unsafe because it corresponds to a run of length 1. Different strings have different numbers of safe bits. For example, a string composed of an alternating sequence of different symbols, such as 212121 has no safe bits, since it has no runs of length $\geq 2$. 

We partition the set of strings we want to cover into two groups based on the number of safe bits they have. Let LS (for low safe-bit strings) be the set of strings of length $n$ that have fewer than $n/6a$ safe bits. Similarly, let HS (for high safe bit strings) be the set of strings with at least $n/6a$ safe bits. Furthermore, we partition HS into $HS_1, \ldots, HS_n$, where $HS_i$ is the set of high safe bit strings with $i$ runs.

We finish this section with a key definition. Consider a Hamming covering code $H$ that covers all bit strings of length $i$ and a string $w$ with $i$ runs. We say that $H$ covers $w$ on a safe bit, if there is a codeword $h \in H$, such that:

1. $h$ and $\text{bits}(\text{runs}(w))$ are at Hamming distance 1, and
2. The bit on which $h$ and $\text{bits}(\text{runs}(w))$ differ corresponds to a safe bit of $w$.

We note that two strings $w_1$ and $w_2$ can have the same bits of runs, yet it is possible that a Hamming covering code covers only one of them on a safe bit. We give an example.

**Example 3.7.4.** Let $w_1 = 22111300$ and $w_2 = 33022211$. The bits of runs for both strings is 0110. Consider a Hamming covering code $H$ containing the string 0010, which is at Hamming distance 1 from 0110. Then $H$ covers both $w_1$ and $w_2$ on the second bit from left, which is a safe bit of $w_1$ but not $w_2$. If there is no other string in $H$ that covers 0110, then $H$ covers $w_1$ on a safe bit but not $w_2$.

In the next section, we will construct an edit covering code $A$ that covers all but $1/n$ fraction of all strings in $HS_i$ using Hamming covering codes that cover the bits of runs of strings in $HS$ on safe bits.

### 3.7.4 Step 4: Existence of a Deletion-1 Code Covering $(1-1/n)$ Fraction of HS

We start this section by explaining how we can take a Hamming covering code and turn it into a deletion-1 code (not necessarily a covering code). Let $HCC_i$ be any covering code for Hamming distance 1 and bit strings of length $i$. We construct a
particular deletion-1 code $EC_{r=i}$ from $HCC_i$ as follows.

$$EC_{r=i} = \{ x | \text{bits}(\text{runs}(x)) \in HCC_i \}$$

That is, we put into $EC_{r=i}$ all strings of length $n - 1$, whose bits of runs is in $HCC_i$.

In the rest of this section, we first state three key lemmas. Lemmas 3.7.1 and 3.7.2 are proved in Section 3.7.5. Then we prove, using the lemmas, that we can build an $O(\log(n)/n)$-size deletion-1 code $C$ that covers all but $1/n$ fraction of strings in $HS$. In Section 3.7.6 we will expand $C$ by codewords that cover the all the strings in $HS$ not covered and the strings in $LS$ and construct a deletion-1 covering code.

**Lemma 3.7.1.** Let $X$ be a subset of the strings in $HS_i$. Suppose there exists a Hamming covering code $HCC_i$ for bit strings of length $i$, such that $|HCC_i| = m$. Then there exists a set $EC_{r=i}$ of strings of length $n - 1$, such that the following is true.

1. $|EC_{r=i}| \leq m/2^{i-1}$ fraction of the strings of length $n - 1$ with $i$ runs.
2. $EC_{r=i}$ covers at least $nm/12a2^i$ fraction of all strings in $X$ on their safe bits.

We defer the proof until Section 3.7.5 At a high level, this lemma says that if we have a small Hamming covering code $HCC_i$ for bit strings of length $i$ and a subset $X$ of strings in $HS_i$, we can construct a small size deletion-1 code $EC_{r=i}$ that covers an important fraction of the strings in $X$. Our next lemma says that such small size Hamming covering codes indeed exist.

**Lemma 3.7.2.** There is an $HCC_i$ code with at most $2^{i+1}/i$ codewords. Put another way, there is a code $HCC_i$ with at most fraction $2/i$ of the binary strings of length $i$.

Again, the proof is deferred to Section 3.7.5 We next state an immediate corollary to Lemmas 3.7.1 and 3.7.2

**Corollary 3.7.1.** For any $i$, with $1 \leq i < n$, there is a deletion-1 code $EC_{r=i}$ of strings of length $n - 1$, such that
1. \(|\text{EC}_{r=i}| \leq 4/i\) fraction of the strings of length \(n - 1\) with \(i\) runs, and

2. \(\text{EC}_{r=i}\) covers at least
\[
\frac{n2^{i+1}}{12ai2^{i+1}} = \frac{n}{12ai} \geq 1/12a
\]
fraction of all strings in \(HS_i\).

Proof. The corollary follows from substituting \(2^{i+1}/i\) from Lemma 3.7.2 for \(m\) in Lemma 3.7.1.

Finally, we need the following lemma to count the number of strings in the deletion-1 code we construct in Theorem 3.7.1.

Lemma 3.7.3. The number of strings of length \(n - 1\) over an alphabet of size \(a\), with \(i\) runs, is \(a(a - 1)^{i-1} \binom{n-2}{i-1}\).

Proof. Imagine a string of length \(n - 1\) with \(i - 1\) “fenceposts” separating the runs. A string of length \(n - 1\) may thus be viewed as \(n - 1\) “regular” symbols and \(i - 1\) fenceposts. However, there are some constraints on where the fenceposts appear. A fencepost cannot occupy the last position, and each fencepost must be preceded by a regular symbol. Thus, we can think of the string and fenceposts as \(i - 1\) pairs consisting of a regular symbol followed by a fencepost, \(n - i - 1\) regular symbols that are not at the end and not followed by a fencepost, and finally, a regular symbol at the end. The number of arrangements of the \(i - 1\) pairs and \(n - i - 1\) regular symbols is \(\binom{n-2}{i-1}\). The factor \(a(a - 1)^{i-1}\) is justified by the fact that the first run can be any of the \(a\) symbols of the alphabet, and each of the \(i - 1\) succeeding runs may be any of the \(a\) symbols except for the symbol that is used for the previous run.

We can now prove that we can construct a \(O(\log(n)/n)\)-size deletion-1 code that covers all but at most \(1/n\) fraction of the strings in \(HS\).

Theorem 3.7.1. There is an \(O(\log(n)/n)\)-size deletion-1 code \(A\) that covers \(1 - 1/n\) fraction of the strings in \(HS\).

Proof. For each \(i, 1 \leq i < n\), we construct a deletion-1 code \(\text{EC}_{r=i}\) as follows: We let \(X = HS_i\) and using Corollary 3.7.1, find a deletion-1 code \(\text{EC}_{(r=i),1}\) that covers at
least fraction $1/12a$ of $X_i$, and contains at most fraction $4/i$ of the strings of length $n-1$ with $i$ runs. Then, we remove the covered strings from $X$ and find an $EC_{(r=i),2}$ that covers at least fraction $1/12a$ of the remaining $X$, and is of size at most fraction $4/i$ of the strings of length $n-1$ with $i$ runs. We repeat this construction $\log_{12a} (n)$ times, to construct $EC_{(r=i),j}$ for $j = 3, 4, \ldots$. We then take the union of all $EC_{(r=i),j}$'s and construct $EC_{r=i}$ which

1. contains $4 \log_{12a} (n)/i$ fraction of all strings of length $n-1$ with $i$ runs, and

2. covers $1 - \frac{1}{n}$ fraction of the strings in $HS_i$.

Let $A = \cup_i EC_{r=i}$. By construction, $A$ covers $1 - \frac{1}{n}$ of all strings in $HS$. That is, each $EC_{r=i}$ covers $1 - \frac{1}{n}$ fraction of all strings in $HS_i$, and $HS = \cup HS_i$. We only have to prove that $A$ is a $O\left(\frac{\log(n)}{n}\right)$-size code: i.e., it contains $O\left(\frac{\log(n)}{n}\right)$ fraction of all strings of length $n$.

By Lemma 3.7.3, the number of strings of length $n-1$ with $i$ runs is $a(a-1)^{i-1}\frac{(n-2)}{i!}$. We know that each $EC_{r=i}$ contains $4 \log_{12a} (n)/i$ fraction of those strings. When we sum over $i$, we get an upper bound on the size of $A$:

$$4 \log_{12a} (n) \sum_{i=1}^{n-1} a(a-1)^{i-1} \frac{(n-2)}{i!} \frac{1}{i}$$

Now, expand the combinatorial function in factorials:

$$4 \log_{12a} (n) \sum_{i=1}^{n-1} a(a-1)^{i-1} \frac{(n-2)!}{(i-1)!(n-1-i)!} \frac{1}{i}$$

Multiply by $(n-1)/(n-1)$, and group the factor $i$ with $(i-1)!$ to get:

$$4 \log_{12a} (n) \sum_{i=1}^{n-1} a(a-1)^{i-1} \frac{(n-2)!}{n-1} \frac{1}{i!(n-1-i)!}$$

Next, observe that the factorials give exactly $\binom{n-1}{i}$. Move all the factors that do not involve $i$ outside the summation to get
The summation is all the terms in the expansion of

\[ [1 + (a - 1)]^{n-1} \]

with the exception of the first and last terms — those for \( i = 0 \). Thus, a good upper bound on Equation 3.1 is

\[
a^{n-1} \frac{4a \log_{\frac{12a-1}{12a-1}}(n)}{(n-1)(a-1)} \sum_{i=1}^{n-1} (a - 1)^i \binom{n-1}{i} \tag{3.2}
\]

The factor \( \log_{\frac{12a-1}{12a-1}}(n) \) is approximately \((12a - 1) \log(n)\):

\[
\log_{\frac{12a-1}{12a-1}}(n) = \log_1 + \frac{1}{12a-1}(n) = \log_1 + \frac{1}{12a-1}(e) \log(n)
\]

As long as \( e \) is the base of the log, \( \log(1 + e) \) is approximately \( e \) for small values of \( e \). Therefore,

\[
\log_1 + \frac{1}{12a-1}(e) = \frac{1}{\log(1 + \frac{1}{12a-1})} \approx \frac{1}{\frac{1}{12a-1}} = 12a - 1
\]

Substituting \((12a - 1) \log(n)\) for \( \log_{\frac{12a-1}{12a-1}}(n) \) in Equation 3.2 we get:

\[
|A| = O\left(\frac{a^n \log(n)}{n}\right)
\]

We will next prove Lemmas 3.7.1 and 3.7.2. Finally, in Section 3.7.6 we will show that for sufficiently large \( n \), the number of outliers that we have to add to \( A \) is less than fraction \( 1/n \) fraction of all strings of length \( n \), which will prove the existence of \( O(\frac{\log(n)}{n}) \)-size deletion-1 covering codes.
3.7.5 Proof of Lemmas

Recall Lemma 3.7.1 states that given a Hamming covering code $HCC_i$ for bit strings of length $i$ of size $m$ and given a set $X$ of strings with enough safe bits ($\geq \frac{n}{6a}$) that we want to cover, we can find a deletion-1 code $EC_{r=i}$ that contains $\frac{m}{2^{i-1}}$ fraction of the strings of length $n-1$ with $i$ runs and that covers a large fraction $\left(\frac{mn}{6a^2}\right)$ of the strings in $X$. Our strategy is to generate a large number of covering codes from $HCC_i$ and calculate the average number of strings they cover from $X$. We can then argue that at least one choice is average or above. We first introduce affine Hamming codes.

Affine Codes

Suppose we start with some fixed Hamming covering code $H = HCC_i$. For any bit string $x$ of length $i$, the affine code $H_x = H \oplus x$ is the set of strings that are formed by taking the bitwise modulo-2 sum of $x$ and any string $w$ in $H$.

**Example 3.7.5.** Suppose $i = 4$ and

$$H = \{0000, 0111, 1011, 1101, 1110\}$$

We leave it to the reader to verify that every string of length four is covered by $H$. There are sixteen ways we can construct an affine code from $H$; some of these codes will be the same, however. We can construct $H$ itself by choosing $x = 0000$. That is, $H_{0000} = H$. If we choose $x = 0011$, we get $H_{0011} = \{0011, 0100, 1000, 1110, 1101\}$, and so on.

Some useful facts about the collection of affine codes is the following.

**Lemma 3.7.4.** If $H$ is a Hamming covering code for strings of length $i$, then so is $H_x$ for any string $x$ of length $i$.

**Proof.** Let $w$ and $x$ be strings of length $i$. We need to show that $w$ is covered by some string in $H_x$. We know that $w \oplus x$ is covered by some string $y$ in $H$. That is, $y$ and $w \oplus x$ differ in at most one bit. Then $y \oplus x$ is in $H_x$. We claim that $w$ and $y \oplus x$ differ in at most one bit, and therefore $w$ is covered by $H_x$. 
Consider any bit \( j \) in which \( y \) and \( w \oplus x \) agree; there will be at least \( i - 1 \) such bits. Let \( w_j, x_j, \) and \( y_j \) be the \( j \)th bits of \( w, x, \) and \( y, \) respectively. Then we are given that \( y_j = w_j \oplus x_j. \) If we add \( y_j \oplus w_j \) to both sides modulo 2 we get 
\[
y_j \oplus y_j \oplus w_j = w_j \oplus x_j \oplus y_i \oplus w_j.\]
Since \( \oplus \) is associative and commutative, and \( z \oplus z = 0 \) for any \( z \), it follows that \( w_j = y_j \oplus x_j. \) Therefore, \( w \) and \( y \oplus x \) differ in at most one bit, and \( w \) is covered by \( H_x. \)

\[\text{Lemma 3.7.5.} \quad \text{Suppose } H \text{ is a Hamming covering code with } m \text{ members, for strings of length } i. \text{ Then among all the affine codes } H_x, \text{ each string of length } i \text{ appears exactly } m \text{ times.}\]

\[\text{Proof.} \quad \text{The string } w \text{ appears in } H_x \text{ if and only if } w = y \oplus x \text{ for some } y \in H. \text{ But } w = y \oplus x \text{ if and only if } x = w \oplus y \text{ (the argument is the same as that given for bits in Lemma 3.7.4). Thus, } w \text{ is in one affine code } H_x \text{ for every member } y \text{ of } H. \text{ Therefore, } w \text{ is in exactly } m \text{ affine codes.} \]

We are now ready to prove Lemma 3.7.1.

\[\text{Proof of Lemma 3.7.1}\]

Let \( HCC_i \) be a Hamming covering code for bit strings of length \( n, \) and let \( B_i \) be the set of strings of length \( n - 1 \) with \( i \) runs. Consider a randomly picked affine code \( HCC_x \) of \( HCC_i \) out of the \( 2^i \) possible affine codes. By Lemma 3.7.5, for each string \( w \) in \( B_i, \) there are exactly \( m \) affine codes of \( HCC_i \) for which \( w \) is in the code. By linearity of expectations, the expected number of strings of \( B_i \) in \( H_x \) is \( m|B_i|/2^i. \) By Markov’s inequality, the probability that the number of strings from \( B_i \) in \( HCC_x \) is greater than twice the expectation is \( \leq 1/2, \) which implies:

\[
Pr(\# \text{ of strings from } B_i \in HCC_x \leq 2m|B_i|/2^i) > \frac{1}{2} \quad (3.3)
\]

Now consider the set \( X \) and a string \( x \in X. \) Recall that strings in \( X \) have more than \( \frac{n}{64} \) safe bits. Let

\[b = \text{bits(runs}(x)\) = b_i, \ldots, b_1\]
Let $b_j$ be a safe bit for $x$. Then there are exactly $m$ affine codes $HCC_y$ of $HCC_i$, such that $HCC_y$ covers $b$ by flipping $b_j$. That is because by Lemma 3.7.5, there are exactly $m$ affine codes of $HCC_i$ that contain the string $b' = b_i, \ldots, b_{j+1}, -b_j, \ldots, b_1$—the string $b$ with $b_j$ flipped. Since there are at least $\frac{n}{6a}$ safe bits in $b$ for $x$, there are at least $\frac{mn}{6a}$ affine codes, whose generated deletion-1 covering code will cover $x$. Therefore, expected number of strings that a randomly picked affine code $H_x$ will cover from $X$ is $\frac{|X|mn}{12a}$. Again by Markov’s inequality, the probability that a random $H_x$ covers fewer than $\frac{|X|mn}{12a}$ strings from $X$ is $\leq \frac{1}{2}$, which implies:

$$Pr(HCC_x \text{ covers } > \frac{|X|mn}{12a} \text{ strings from } X) > \frac{1}{2} \quad (3.4)$$

By Equation 3.3 the probability that a randomly picked $HCC_x$ contains less that $2m|B_i|/2^i$ fraction of the strings of length $n - 1$ with $i$ runs is $> \frac{1}{2}$. By Equation 3.4 the probability that a random $HCC_x$ covers more than $|X|mn/12a$ strings from $X$ is $> \frac{1}{2}$. Then, there must exist one $HCC_x$ for which both conditions hold, completing the proof of the lemma.

We next prove Lemma 3.7.2.

**Proof of Lemma 3.7.2**

Let $i$ be in the range $2^{r-1} - 1 < i \leq 2^r - 1$ for some integer $r$. There is a Hamming code $A$ for strings of length $2^{r-1} - 1$, and it is a perfect code, so it is also a Hamming covering code for that length. Take the cross-product of $A \times \{0,1\}^{i-2^{r-1}+1}$; call the resulting code $A'$. That is, expand the code $A$ by appending all possible bit strings to each of the codeword, to make strings of length $i$ rather than length $2^{r-1} - 1$.

We claim that $A'$ is a covering code for strings of length $i$. In proof, let $w$ be a string of length $i$. Since $A$ is a covering code, we know we can find a member $x$ of $A$ such that the first $2^{r-1} - 1$ bits of $w$ differ from $x$ in at most one bit. Extend $x$ with the last $i - 2^{r-1} + 1$ bits of $w$. We now have a codeword of $A'$, and that codeword differs from $w$ in at most one bit.
3.7.6 Steps 5 and 6: Existence of $O(\log(n)/n)$-size Deletion-1 Codes

We are now ready to complete our proof that $O(\log(n)/n)$-size deletion-1 codes exist. So far, we partitioned strings of length $n$ into $LS$, those with $< \frac{n}{6a}$ safe bits, and $HS$, those with $> \frac{n}{6a}$. We then showed in Theorem 3.7.1 that we can cover all but $\leq \frac{1}{n}$ fraction of the strings in $HS$ with a $O(\log(n)/n)$-size code $C$. The two groups of outliers to $C$ are: (1) the $\leq \frac{1}{n}$ fraction of strings in $HS$ that $C$ does not cover; and (2) the strings in $LS$. Notice that the size of the strings in (1) is $\leq \frac{1}{n}$ fraction of all strings of length $n$, since $HS$ is a subset of all strings of length $n$. Our next lemma states that for large enough $n$, the size of $LS$ is also $\leq \frac{1}{n}$ of all strings of length $n$.

**Lemma 3.7.6.** For $n$ such that $n/\log(n) \geq 24a$, $|LS| \leq \frac{1}{n}$ fraction of all strings of length $n$.

**Proof.** Instead of counting $|LS|$ directly, we will count another set $LSP$ for low "special" letter strings which contains $LS$. Divide a string $w$ of length $n$ into chunks of length three: $w = w_1w_2w_3|w_4w_5w_6| \ldots |w_{n-2}w_{n-1}w_n$. For simplicity, we assume $n$ is divisible by 3. Call $w_{3j}$, last letter of a chunk, for $j \in 1, \ldots, \frac{n}{3}$ a special letter if the following two conditions hold:

1. $w_{3j}$ equals $w_{3j-1}$ (the symbol to its left)
2. $w_{3j}$ is different from $w_{3j-2}$ (the symbol two positions to its left)

In other words, the letter has to be in a position congruent to 0 mod (3) and be the second letter in a run of length $\geq 2$. For example, if $w = 231|100|034$, the 0 in position six (bolded) is the only special letter. Notice that the 1 at position four is the second letter in a run of length 2. However, it is not a special letter because it is not in position congruent to 0 mod 3. Let $LSP$ be the set of strings with fewer than $\frac{n}{6a}$ special letters.

We first show that $LSP$ contains $LS$. Consider a string $w \in LS$. Then $w$ has $< \frac{n}{6a}$ runs of length $\geq 2$. Then it has $< \frac{n}{6a}$ letters that satisfy conditions (1) and (2) above. Therefore it must have $< \frac{n}{6a}$ letters that satisfy conditions (1) and (2) and are also in a position congruent to 0 mod (3). Therefore $w$ must also be in $LSP$. 
We complete the proof by showing that for large $n$, the size of $LSP$ is very small. Consider a procedure that generates strings of length $n$ by generating $n$ independent letters. We look at the generated string in chunks of three. Let $Y_1, \ldots, Y_{n/3}$ be random variables, such that $Y_i = 1$ if the last letter of the $i$th chunk is special and 0 otherwise. 

$$\Pr(Y_i = 1) = \frac{a - 1}{a} \times \frac{1}{a} = \frac{a - 1}{a^2}.$$ 

This is because for $Y_i$ to assume a value of 1: (1) the first letter can be anything; (2) the second letter has to be different from the first letter: a probability of $\frac{a - 1}{a}$; and (3) the third letter has to equal to the second letter: a probability of $\frac{1}{a}$. And notice that $Y_i$ are independent of each other because the value that $Y_i$ takes only depends on the three bits produced for chunk $i$. By linearity of expectation, number of special bits in a random string is 

$$\frac{n(a - 1)}{3a^2} \geq \frac{n}{6a}.$$ 

Let $Z = \sum Y_i$. By Chernoff bounds, $\Pr(Z < n/12a) < e^{-n/48a}$ which is less than $1/n$ for $n/\log(n) > 48a$. Since $|LS| < |LSP|$, for $n/\log(n) > 48a$, $|LS| < 1/n$ of all strings of length $n$ completing our proof.

We can now formally prove that $O(\log(n)/n)$-size deletion-1 codes exist for long enough strings.

**Theorem 3.7.2.** There exists a $O(\log(n)/n)$-size deletion-1 code for strings of length $n$ when $n/\log(n) > 48a$.

**Proof.** We start with the $O(\log(n)/n)$-size code $A$ from Theorem 3.7.1. For each uncovered string $w$ in $HS$ and $LS$, we put one codeword into $A$ covering $w$, for example by deleting the first symbol of $w$, and produce a deletion-1 covering code. The number of uncovered strings in $HS$ is $\leq \frac{1}{n}$ fraction of all strings of length $n$. Similarly, by Lemma 3.7.6, the size of $LS$ is $\leq \frac{1}{n}$ fraction of all strings of length $n$ for $n/\log(n) > 48a$. Therefore expanding $A$ does not affect its asymptotic size of $O(\log(n)/n)$, for $n/\log(n) > 48a$.

### 3.8 Related Work

We reviewed related work on the general problem of fuzzy joins in MapReduce in Chapter 2. Here, we review prior work on covering codes. There is a significant literature regarding covering codes for Hamming distance. Reference 34 provides
a good overview of this material. A significant portion of this prior literature finds Hamming distance covering codes for specific $n$, $k$, and $a$ values. For some $n$, $k$, and $a$ values these codes can be smaller than the ones we generate in this chapter, and therefore yield better performance when computing fuzzy joins with GAP. A modification of the problem called “asymmetric” covering codes has been considered for the binary alphabet \[ 13, 35 \]. An asymmetric binary covering code covers every bit string $w$ of length $n$ by changing at most $k$ 1’s of $w$ to 0’s. As for covering codes in the original formulation, lower and upper bounds on the sizes of asymmetric covering codes are known only for small values of $n$ and $k$. The edit-distance covering codes we proposed can also be considered asymmetric codes, as they cover strings either only by a deletion or only by an insertion. No prior work we are aware of proposes covering codes for edit distance. We constructed the first edit-distance covering codes in this thesis.
Chapter 4

Multiround MapReduce

In Chapters 2 and 3 we focused on one-round MapReduce (MR) computations. In this chapter, we show that we can solve problems much more efficiently than any possible one-round algorithm if we can execute a sequence of MR jobs. We focus on two problems in this chapter: (1) dense matrix multiplication; and (2) multiway equijoins. Recall from Chapter 2 that, for some problems, such as “all-to-all”, dense matrix multiplication, fuzzy joins under Hamming distance 1, and multiway equijoins, we derived lower bound curves for the communication cost of one-round MR algorithms for different reducer sizes (parallelism levels). Using these lower-bound curves, we were able to show that some algorithms were Pareto optimal, i.e., incurred the best possible communication cost for their reducer sizes. Unfortunately we do not have lower-bound curves for multiround algorithms solving the problems we study in this chapter. Although we can show that our multiround algorithms are significantly more efficient than any possible one-round algorithm, whether or not there are more efficient multiround algorithms is an interesting open problem not solved in this thesis.

In Section 4.1 we begin by describing a two-round algorithm for dense matrix multiplication called the Partial-sums Algorithm. The bulk of this chapter is devoted to Sections 4.2-4.8 where we describe a multiround algorithm for the multiway-eqijoin problem; the algorithm is called GYM, for Generalized Yannakakis in MapReduce. Recall from Chapter 2 that in the multiway-eqijoin problem, we want to evaluate
a query $Q$ that involves $n$ relations. The performance of GYM depends on two important structural properties of $Q$: the depth and width of its generalized hypertree decompositions (GHDs), which we will define later in the chapter. GYM takes as input a GHD of a query $Q$ of width $w$ and depth $d$, and executes $O(d)$ rounds of computation with a communication cost that increases as a function of $w$ for different reducer sizes $M$. The main technical contributions of this chapter are in Sections 4.6 and 4.7, where we present methods for constructing GHDs of different depths and widths for a given query, exposing a spectrum of tradeoffs one can make between the number of rounds and communication. Figure 4.1 shows how the reducer-size and communication cost curves of GYM change as we construct lower width but higher depth GHDs for the Chain-n query: $R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \cdots \bowtie R_n(A_n, A_{n+1})$. As shown in the figure, as we increase the number of rounds of computation, the curves get lower; that is, we can compute queries more efficiently.

#### 4.1 Dense Matrix Multiplication

Recall from Section 2.3 of Chapter 2 that in the dense matrix multiplication problem, we are given two $n \times n$ matrices $R = [r_{ij}]$ and $S = [s_{jk}]$ and we wish to form their
product $T = [t_{ik}]$, where $t_{ik} = \sum_{j=1}^{n} r_{ij} s_{jk}$. Recall further the one-round Square Tiling algorithm, which was Pareto optimal at up to $n$ parallelism levels. Square Tiling sends each reducer a set of rows of $R$ and an equal number of columns of $S$. In this section, we show another algorithm, called the Partial-sums algorithm, for performing matrix multiplication using two MR jobs. As we show, the Partial-sums algorithm always beats Square Tiling. An interesting aspect of our analysis is that, while tiling by squares works best for a one-round computation, the Partial-sums algorithm tiles the matrices into rectangles that have aspect ratio 2:1. The Partial-sums algorithm performs as follows:

1. In the first round, we compute $x_{ijk} = r_{ij} s_{jk}$ for each $i$, $j$, and $k$ between 1 and $n$. We sum the $x_{ijk}$’s at a given reducer if they share common values of $i$ and $k$, thus producing a partial sum for the pair $(i, k)$.

2. In the second round, we have one reducer for each $t_{ik}$ output cell. The partial sums produced for $(i, k)$ are sent to the reducer for $t_{ik}$ and summed there to produce $t_{ik}$.

Figure 4.2 shows the computations performed by the mappers and reducers of the Partial-sums algorithm. We next analyze the reducer size and communication cost of Partial-sums.
• **First Round:** As for the Square Tiling algorithm, we can simplify the options regarding what inputs go to what reducers in the first round by observing that the set of outputs covered by a reducer again forms a “rectangle.” That is, if a reducer covers both $x_{ijk}$ and $x_{yjz}$, then it also covers $x_{ijz}$ and $x_{yjk}$. The proof is that to cover $x_{ijk}$ the reducer must have inputs $r_{ij}$ and $s_{jk}$, while to cover $x_{yjz}$ the same reducer gets inputs $r_{yj}$ and $s_{jz}$. From these four inputs, the reducer can also cover $x_{ijz}$ and $x_{yjk}$.

We now know that the set of outputs covered by a reducer can be described for each $j$ by a set $G_j$ of row numbers of $R$ and a set of column numbers $H_j$ of $S$, such that the outputs covered are all $x_{ijk}$ for which $i$ is in $G_j$ and $k$ is in $H_j$. As before, the greatest number of covered outputs occurs when the rectangle is a square. That is, each reducer has an equal number of rows and columns for each $j$. We do not know that these rows and columns must be the same for each $j$, but it is easy to argue that if not, we could reduce the communication in the first and second rounds, or both, by using the same sets of rows and columns for each $j$.

Thus, we shall assume that each reducer in the first round is given a set of $s$ rows of $R$, $s$ columns of $S$, and $t$ values of $j$ for some $s$ and $t$. Therefore the reducer size of the first round is equal to $2st$. There is a reducer covering each $x_{ijk}$, which means that the number of reducers is $(n/s)^2(n/t)$, for $s$ and $t$ that divide $n$. Then each element of matrices $R$ and $S$ is sent to $n/s$ reducers, so the total communication in the first round is $2n^3/s$. To see why, consider an element $r_{ij}$ of matrix $R$. We know $i$ and $j$, so only $k$ is unknown. The number of reducers that need inputs with the particular $i$ and $j$ and some $k$ is $n/s$. The analogous argument applies to elements of matrix $S$. Therefore, $M = 2st$, and $C = 2n^3/s$ in the first round.

• **Second Round:** The second round is embarrassingly parallel, since each partial sum contributes to only one output. Since we have $n/t$ partial sums for each cell of the output matrix, each reducer gets $n/t$ inputs. Since there are $n^2$ output cells to produce, we have $n^2$ reducers, so the communication cost is
$n^2(n/t) = n^3/t$. Therefore $M = n/t$ and $C = n^3/t$ in the second round.

Therefore, the reducer size and the communication cost of the entire computation is:

$$M = \max\{2st, \frac{n}{t}\}, \quad C = \frac{2n^3}{s} + \frac{n^3}{t}$$

First notice that we would never pick an $M$ value in which $n/t$, the second round reducer size, is the dominant factor. To see why, notice that if $2st < n/t$, we can always increase $t$ and therefore decrease both the reducer size and the communication. As a result, we can assume that we pick a reducer size in which $M = 2st \geq n/t$. Let us minimize the communication subject to the constraint that $2st = M$. The method of Lagrangean multipliers lets us show that the minimum is obtained when $s = 2t$. Therefore $M = 4t^2 \geq n/t$, and $C = 2n^3/t = 4n^3/\sqrt{M}$.

Figure 4.3 shows the one-round tradeoff curve, the one-round Square Tiling algorithm, whose cost was $\frac{4n^3}{M}$, and the Partial-sums algorithm. Similar to our previous figures, the x-axis (M) is the reducer size (or parallelism level) and the y-axis (C) is the communication cost. Since we pick integer values for $t \leq n$, and $s = 2t \leq n$, and $4t^2 \geq n/t$, $t$ can take any integer value between $[(n/4)^{1/3}, n/2]$. For each $t$ value in this range, Partial-sums algorithm runs at a parallelism level of $M = 4t^2$ and incurs a communication cost of $C = 4n^3/\sqrt{M}$. As can be seen, Partial-sums beats
the one-round tradeoff curve on every level it can execute at. When \( t = n \), then our
Lagrangean solution of \( s = 2t \) does not apply, since \( s \) can be at most \( n \). However in
this case, we can take \( s \) to be equal to \( n \) as well and meet the performance of the
one-round algorithm. This degenerate case corresponds to pure serial computation
where we multiply the matrices inside a single reducer in the first round, and the
second round can be omitted.

Finally, we note that another advantage of the the Partial-sums algorithm is that
it can run at parallelism levels that no one-round algorithm can run at. For example,
using Partial-sums, we can pick \( t = (n/4)^{1/3} \) and run at a parallelism level where
\( M = O(n^{2/3}) \). However, no one-round algorithm can run at this parallelism level,
since any one-round algorithm needs to get at least one entire row of \( R \) and one
entire column of \( S \) to output any cell. So even at maximum parallelism, \( M \) is at least
\( 2n \) for any one-round algorithm.

4.2 Multiway Equijoins

The remainder of this chapter studies a multiround algorithm called GYM for the
mulitway equijoin problem. Before we start describing GYM, we begin this section
with a discussion of why multiround algorithms are necessary for evaluating some
joins efficiently. Recall that in Section 2.5 of Chapter 2, we showed that it can be
very difficult to parallelize some queries in a single round of MR. When deriving a
one-round tradeoff curve to evaluate a query \( Q \), we focused on instances of input
relations that came from the max-out distribution. The formula for the one-round
tradeoff curve for evaluating \( Q \) over the max-out distribution was
\[ C \geq \frac{\text{IN} \rho^*}{M \rho^* - 1}, \]
where \( \rho^* \) is the value of an optimal fractional edge covering of \( Q \) (or equivalently a vertex
packing). The max-out distribution intuitively corresponded to skewed instances,
whose outputs were the maximum possible, given that there are \( \text{IN}/n \) tuples in each
input relation.

However, the difficulty of parallelizing some of these queries in a single round is
actually not due to their potential for having large outputs. Interestingly, Beame
et al. [18] has shown that even when the inputs come from a distribution called
CHAPTER 4. MULTIROUND MAPREDUCE

Fractional Edge Packing

\( \forall j \in [1,n], u_j \geq 0 \)

\( \forall i \in [1,m], \sum_{j \text{ s.t. } A_i \in \text{attrs}(R_j)} u_j \leq 1 \) (4.1)

maximize \( \sum_{j \in [1,n]} u_j \)

Figure 4.4: Fractional edge packing.

<table>
<thead>
<tr>
<th>Query</th>
<th>Edge Covering</th>
<th>( \rho^* )</th>
<th>( \tau^* )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain-n</td>
<td>( R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4) \bowtie \ldots \bowtie R_n(A_n, A_{n+1}) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chain-n</td>
<td>1 0 1 1 ... 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle-n</td>
<td>( R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4) \bowtie \ldots \bowtie R_n(A_n, A_1) )</td>
<td>1/2 1/2 1/2 ... 1/2</td>
<td>( n/2 )</td>
</tr>
<tr>
<td>Snowflake-n</td>
<td>( S(B_1, \ldots, B_n) \bowtie R_1(A_1, B_1) \bowtie R_2(A_2, B_2) \bowtie \ldots \bowtie R_n(A_n, B_n) )</td>
<td>0 1 1 1 ... 1</td>
<td>( n )</td>
</tr>
<tr>
<td>Snowflake-n</td>
<td>0 1 1 1 ... 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Star-n</td>
<td>( R_1(A_1, B) \bowtie R_2(A_2, B) \bowtie \ldots \bowtie R_n(A_n, B) )</td>
<td>1/n 1/n ... 1/n</td>
<td>( n )</td>
</tr>
<tr>
<td>Star-n</td>
<td>1 1 1 ... 1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1: Example queries and their edge covering and edge packing numbers.

matching databases, which have very small outputs, i.e., \( \text{OUT} = O(\text{IN}) \), evaluating some queries in a single round remain very difficult. In comparison to the max-out distribution, matching databases intuitively correspond to instances with no skew.

The exact formula for the lower bounds over matching databases is \( \frac{\text{IN}^\tau}{M} \), where \( \tau^* \) is the value of an optimal fractional edge packing of \( Q \). A fractional edge packing of \( Q \) is any feasible solution to the LP shown in Figure 4.4.

Similar to an edge covering (recall Figure 2.7 from Chapter 2), an edge packing assigns non-negative values to each relation \( R_j \). However, unlike an edge covering, which guarantees that every attribute \( A_i \) is “covered”, an edge packing guarantees that \( A_i \) is not “overpacked”, i.e., \( \sum_{j \text{ s.t. } A_i \in \text{attrs}(R_j)} u_j \leq 1 \).
CHAPTER 4. MULTIROUND MAPREDUCE

The value of $\tau^*$ of any query $Q$ is at most the value of $\rho^*$ for $Q$, implying that evaluating queries in a single round is easier over matching databases than over max-out distribution. Table 4.1 extends Table 2.8, showing both the optimal edge covering and edge packing values of several queries. Similar to Table 2.8, the first and second set of values under the relations represent edge coverings and edge packings, respectively.

For some queries, the difference between $\rho^*$ and $\tau^*$ can be very large. For example, consider the Star-n query:

$$R_1(A_1, B) \bowtie R_2(A_2, B) \bowtie \ldots \bowtie R_n(A_n, B)$$

Evaluating Star-n over the max-out distribution can generate $\text{IN}^O(n)$ communication even when the reducer size $M = \text{IN}^c$ for some constant $c$. In contrast, over matching databases, evaluating Star-n query in a single round is embarrassingly parallel. Notice that this is intuitive, because if there is no skew and we use $p$ reducers, we can evaluate Star-n by simply hashing each tuple on attribute $B$ and sending to a single reducer, and expect that each reducer gets $\text{IN}/p$ of the tuples on average. However, for other queries, there may be very little or no difference between $\rho^*$ and $\tau^*$. For example, for the Cycle-n and Snowflake-n queries, the difference is 0; for the Chain-n query, the difference is at most 1. As a result, the one-round tradeoff curve for these queries is very robust and remains almost the same even when the outputs of these queries are very small and there is no skew in the inputs. In other words, even when the inputs come from an easy distribution, these queries cannot be evaluated efficiently in a single round. Therefore, designing multiround algorithms is the only way to evaluate these queries more efficiently.

In the rest of this section first summarize our main results on evaluating joins with GYM (Section 4.2.1). Then, we give an outline for the rest of the sections of this chapter (Section 4.2.2).

4.2.1 Summary of GYM and Main Results

GYM is a distributed and generalized version of Yannakakis’s algorithm for acyclic queries [132], explained momentarily. GYM takes as input the $n$ relations of a query $Q$ and a generalized hypertree decomposition (GHD) of $Q$, defined in Section 4.3. The number of rounds and communication cost of GYM depends on the depth and
the width of the input GHD, respectively. Specifically, GYM can be highly efficient on GHDs with low widths, and execute for a small number of rounds on GHDs with short depths. We present two algorithms, Log-GTA (Section 4.6.4) and C-GTA (Section 4.7), for constructing GHDs of different depths and widths of \( Q \), exposing a spectrum of tradeoffs one can make between the number of rounds and communication. The green, blue, and red curves in Figure 4.1 show the performance of GYM on three different GHDs for the Chain-n query, subject to some assumptions we will explain in Section 4.8.

The original algorithm of Yannakakis runs on a single machine and evaluates only acyclic queries. The width of a query, i.e., the minimum width of any of its GHDs, characterizes its degree of cyclicity, where acyclic queries are equivalent to width-1 queries. Yannakakis’s algorithm takes as input a width-1 GHD of an acyclic query over \( n \) relations and always executes a sequence of \( \Theta(n) \) semijoins and joins in \( \Theta(IN + OUT) \) time. As we show in Section 4.4.3 Yannakakis’ algorithm can easily be mapped to the MR setting. The resulting algorithm, which we call DYM (for Distributed Yannakakis in MapReduce), has a communication cost of \( O(n \frac{(IN+OUT)^2}{M}) \).

Throughout our analysis, we let \( B(X, M) \) denote the communication cost of any binary join, i.e., join of two relations, where the the total size of the relations is \( X \) and machine size is \( M \). Recall from Chapter 2 Section 2.5.4 that if we use the Grouping algorithm, we can join any two relations of size \( X \) with a communication cost of \( \frac{X^2}{M} \). The communication cost of DYM can thus be written as \( O(nB(IN + OUT, M)) \). We note that some of our results have a \( \log_M(IN) \) term (see Section 4.4.2). To simplify these formulas, we also assume throughout this chapter that \( M = IN^{\frac{1}{\epsilon}} \) for some constant \( \epsilon > 1 \); so \( \log_M(IN) \) is a constant that can be omitted inside big-oh notation. For practical values of input and machine sizes, \( \epsilon \) is a small constant greater than 1. For instance, if \( IN \) is in terabytes and the machine size is in megabytes, then \( \epsilon \approx 2 \). Even if the machine size is in kilobytes, \( \epsilon \approx 4 \).

GYM generalizes DYM by taking as input any GHD of any query with any degree of cyclicity. Let \( D \) be a width-\( w \), depth-\( d \) GHD of a query \( Q \). Our first main result is the following:
CHAPTER 4. MULTIROUND MAPREDUCE

Main Result 1: GYM computes $Q$ in $O(d + \log(n))$ rounds of MR with

$$O(nB(IN^w + OUT, M))$$

communication cost.

GYM is based on three observations:

1. Each join between two input relations can be executed in one round with a communication cost in $O(B(IN + OUT, M)) = O\left(\frac{(IN+OUT)^2}{M}\right)$. Each semijoin can be computed in a constant number of rounds with the same cost.

2. We can further parallelize the algorithm from step 1 by executing some of the semijoins or joins in parallel, reducing the number of rounds to $O(d + \log(n))$ without affecting the communication cost of the algorithm from step 1.

3. We can generalize the algorithm from step 2 to take as input any GHD $D$ of any (possibly cyclic) query by first running the Grouping or the Shares algorithm to join some of the input relations of $Q$. This preprocessing step takes a constant number of rounds and at most $O\left(\frac{INw}{M}\right)$ cost and generates a set of acyclic intermediate relations over which the algorithm from step 2 can be run.

On acyclic queries with constant-depth GHDs, GYM executes for only $O(\log(n))$ rounds and incurs a communication cost of $O(nB(IN + OUT, M))$. Snowflake-n or Star-n are examples of acyclic queries with constant depths. We will define the notions of width and cyclicity in Section 4.3. However, there are acyclic queries, such as the Chain-n query, whose width-1 GHDs have a depth of $\Theta(n)$. On width-1 GHDs of such queries, GYM executes for $\Theta(n)$ rounds.

Our second main result shows how to execute such queries by GYM in fewer rounds but with more communication cost by proving a combinatorial lemma about GHDs, which may be of independent interest to readers:

Main Result 2: Let $Q$ be an equijoin query of $n$ relations. Given a width-$w$, depth-$d$ GHD $D$ of $Q$, we can construct a GHD $D'$ of $Q$ of depth $O(\log(n))$ and width at most $3w$.

We describe a GHD transformation algorithm called Log-GTA to achieve our second main result. This result implies that by increasing the communication cost from
$O(nB(IN^w + OUT, M))$ to $O(nB(IN^{3w} + OUT, M))$, we can decrease the number of rounds from $O(n)$ to $O(\log(n))$ for width-$w$ queries with long depth GHDs. Interestingly, our second main result implies the result proven by ACQ [46] that constant-width queries are in the complexity class NC. We discuss this implication more in Section 4.9 when we discuss related work. We also describe another GHD transformation algorithm called $C$-GTA, using which one can further reduce the depths of the GHDs of queries to $O(\log((\frac{10}{16})^t n))$ at the cost of increasing their widths to $2^t3w$, exposing a layer of other tradeoffs that are possible between number of rounds and communication.

We note that all of our results hold under any amount of skew in the input data. Specifically, we state our results assuming that there may be heavy hitter values on the attributes of the input tables. For example, if during any round GYM joins two tables, $R_1(A_1, A_2)$ and $R_2(A_2, A_3)$, column $A_2$ might contain a single heavy hitter value $b_i$.

Our approach to modifying Yannakakis’s algorithm to run in logarithmic rounds might surprise database researchers, who have often thought of Yannakakis’s algorithm as having a sequential nature, executing for $\Theta(n)$ steps in the PRAM model. In the PRAM literature [38 52 53], acyclic queries have been described as being polynomial-time sequentially solvable by Yannakakis’s algorithm, but highly “parallelizable” by the $ACQ$ algorithm [46], where parallelizability refers to executing for a small number of PRAM steps. By simulating MR in the PRAM model, our results show that unlike previously thought, we can easily parallelize Yannakakis’s algorithm with simple optimizations.

4.2.2 Outline for the Rest of the Chapter

Here is the outline for the rest of the sections in this chapter:

- Section 4.3 covers the necessary background on GHDs.
- In Section 4.4 we describe two distributed versions of Yannakakis’s algorithm, DYM-n and DYM-d, as stepping stones to GYM. DYM-n and DYM-d take as input width-1 GHDs of acyclic queries.
• In Section 4.5, we describe GYM, which generalizes DYM-d to take as input any width-\(w\), depth-\(d\) GHD of any query and run in \(O(d + \log(n))\) rounds and \(O(nB(\text{IN}^w + \text{OUT}, M))\) communication.

• In Section 4.6, we describe our Log-GTA algorithm for transforming any width-\(w\), depth-\(d\) GHD \(D\) of a query \(Q\), with \(d \in \Omega(\log(n))\) into another GHD \(D'\) of \(Q\), whose depth is \(O(\log(n))\) and width is at most \(3w\). By giving \(D'\) as input to GYM, we can compute width-\(w\) queries in \(O(\log(n))\) rounds and \(O(nB(\text{IN}^{3w} + \text{OUT}, M))\) cost.

• In Section 4.7, we describe our C-GTA algorithm, which transforms a width-\(w\) GHD \(D\) of a query with \(n\) vertices into width-2\(w\) GHD \(D'\) with at most \(\frac{15n}{16}\) vertices. We can use C-GTA along with Log-GTA to construct GHDs with shorter depths but higher widths.

• In Section 4.8, we demonstrate how to construct different GHDs for the Chain-n query and the effects of these GHDs on the performance of GYM.

• Section 4.9 covers related work on multiround join algorithms and GHDs.

4.3 Generalized Hypertree Decompositions

In this section, we review the notions related to generalized hypertree decompositions of queries [51]. A hypergraph is a pair \(H = (V(H), E(H))\), consisting of a nonempty set \(V(H)\) of vertices, and a set \(E(H)\) of subsets of \(V(H)\), the hyperedges of \(H\). Multiway equijoin queries can be expressed as hypergraphs, where we have a vertex for each attribute of the query, and a hyperedge for each relation.

**Example 4.3.1.** Consider the query \(Q:\)

\[
R_1(A, B, C) \bowtie R_2(B, F) \bowtie R_3(B, C, D) \bowtie \\
R_4(C, D, E) \bowtie R_5(D, E, G)
\]

The hypergraph corresponding to \(Q\) is shown in Figure 4.5a.

Let \(H\) be a hypergraph. A generalized hypertree decomposition (GHD) of \(H\) is a triple \(D = (T, \chi, \lambda)\), where:
• \( T(V(T), E(T)) \) is a tree;
• \( \chi : V(T) \to 2^{V(H)} \) is a function associating a set of vertices \( \chi(t) \subseteq V(H) \) to each vertex \( t \) of \( T \);
• \( \lambda : V(T) \to 2^{E(H)} \) is a function associating a set of hyperedges to each vertex \( t \) of \( T \);

such that the following properties hold:

1. For each \( e \in E(H) \), there is a vertex \( t \in V(T) \) such that \( e \subseteq \chi(t) \).
2. For each \( v \in V(H) \), the set \( \{ t \in V(T) | v \in \chi(t) \} \) is connected in \( T \).
3. For every \( t \in V(T) \), \( \chi(t) \subseteq \bigcup \lambda(t) \).

Consider a query \( Q \) that joins a set of \( n \) relations \( R_0, ..., R_{n-1} \), where the schemas of the relations contain \( m \) attributes \( A_0, ..., A_{m-1} \). We could rephrase these definitions and properties as follows. A GHD of \( Q \) is a triple \( D = (T, \chi, \lambda) \), where:

• \( T(V(T), E(T)) \) is a tree;
• \( \chi : V(T) \to 2^{V(H)} \) is a function assigning a set of attributes to each vertex \( t \) of \( T \); we refer to \( \chi(t) \) as the attributes of \( t \).
• \( \lambda : V(T) \to 2^{E(H)} \) is a function assigning a set of relations to each vertex \( t \) of \( T \); we refer to \( \lambda(t) \) as the relations of \( t \).

such that the following properties hold:

1. For each relation \( R_i \), the attributes of \( R_i \) are contained within at least one vertex
2. For any attribute $A_i$, let $T_{A_i}$ be the subgraph in $T$ containing only the vertices containing $A_i$. Then $T_{A_i}$ must be connected.

3. For every $t \in V(T)$, each attribute of $t$ is contained in at least one of the relations of $t$.

**Example 4.3.2.** Figure 4.5b shows a GHD of the query from Example 4.3.1. In the figure, the attribute values on top of each vertex $t$ is the $\chi$ assignments for $t$ and the $\lambda$ assignments are explicitly shown.

The **depth** of a GHD $D = (T, \chi, \lambda)$ is the depth of the tree $T$. The **width** of a GHD $D$ is the $\max_{t \in V(T)}\{|\lambda(t)|\}$, i.e., the maximum number of relations assigned to any vertex $t$. The **generalized hypertree width (ghw)** of a hypergraph $H$ is the minimum width of all hypertree decompositions of $H$. With some abuse of notation, when we say the “width” of a query $Q$ is $w$, we will mean that the ghw of the hypergraph corresponding to $Q$ is $w$. The width of a query captures its degree of cyclicity. In general, the larger the width of a query, the more “cyclic” it is. By definition, a query is acyclic if and only if its hypergraph is acyclic. Equivalently, acyclic queries are exactly the queries with width 1 [31].

In the rest of this chapter we restrict ourselves, for simplicity of presentation, to queries whose hypergraphs are connected. However, all of our results generalize to queries with disconnected hypergraphs. We end this section by stating a lemma about connected hypergraphs and GHDs of queries that will be used in later sections:

**Lemma 4.3.1.** If a query $Q$ involving $n$ relations has a width-$w$ GHD $D = (T, \chi, \lambda)$ of depth $d$, then $Q$ has GHD $D' = (T', \chi', \lambda')$ with width $w$ and $|V(T')| \leq n$.

**Proof.** Call a GHD $D = (T, \chi, \lambda)$ minimal if for any vertices $u, v \in V(T)$, neither of the sets $\chi(u)$ and $\chi(v)$ is a subset of the other. If $\chi(u) \subset \chi(v)$, then we could simply merge vertices $u$ and $v$ into a single vertex $s$ and assign $s$ v’s $\chi$ and $\lambda$ values. Collapsing an edge in a tree cannot introduce cycles; so we get another GHD for the same query with fewer vertices and at most the same width and depth. Thus if $D$ is not minimal, we can make it minimal by merging some of its vertices iteratively, without increasing its depth or width.
We now prove that for any minimal GHD $D = (T, \chi, \lambda)$, if $D$ is a GHD for a query $Q$ having $n$ relations, then $|V(T)| \leq n$. We use induction on $|V(T)|$. Base Case: $|V(T)| = 1 \leq n$, since the query being covered by a non-empty GHD must have at least one relation. Inductive Step: Assume that for all minimal GHDs with $|V(T)| \leq k - 1$, any query that they cover must have at least $|V(T)|$ relations. Now consider a GHD $D = (T, \chi, \lambda)$ with $|V(T)| = k$. Let $l$ be a leaf vertex of tree $T$. Because $D$ is minimal, $\chi(l)$ contains at least one attribute $a$ that is not contained in $\chi(u)$ for any other $u \in V(T)$. Query $Q$ must have at least one relation $R$ that contains $a$, and this relation $R$ can only lie in $\lambda(l)$. Now consider the GHD $D_2 = (T_2, \chi_2, \lambda_2)$ obtained by deleting $l$ from $T$, and query $Q_2$ obtained by deleting all relations from $\lambda(l)$. Since we deleted at least one relation from $Q$, $Q_2$ has $\leq n - 1$ relations. Then $D_2$ is a minimal GHD for $Q_2$, and by the inductive hypothesis $|V(T_2)| \leq n - 1$. And since $|V(T)| = |V(T_2)| + 1$, we have $|V(T)| \leq n$ as required. Therefore we can take any $D$ and make it minimal without affecting its depth or width and get a $D'$ that has at most $n$ vertices.

\section*{4.4 Distributed Yannakakis}

We first review the serial version of Yannakakis’s algorithm for acyclic queries (Section 4.4.1). We then show that the algorithm can be parallelized in a straightforward fashion to yield an $O(n)$-round MR algorithm with $O(nB(IN + OUT, M))$ communication cost (Section 4.4.2). Recall that $B(X, M)$ is the communication cost of joining two relations of size $X$ using machines of size $M$ and is equal to $\frac{X^2}{M}$ (Section 4.4.2). Finally, we show that we can reduce the number of rounds of the algorithm to $O(d + \log(n))$, where $d$ is the depth of a width-1 GHD of the input acyclic query (Section 4.4.3).

\subsection*{4.4.1 Serial Yannakakis Algorithm}

The serial version of the Yannakakis algorithm takes as input an acyclic query $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$, and constructs a width-1 GHD $D = (T, \chi, \lambda)$ of $Q$. Since $D$ is
a GHD with width 1, each vertex of $D$ is assigned exactly one relation $R_i$ and each $R_i$ is assigned to some vertex of $D$. We will refer to relations that are assigned to leaf (non-leaf) vertices in $T$ as leaf (non-leaf) relations. Therefore $D$ is effectively a join tree (also called a parse tree) for $Q$ that can be joined in any bottom-up fashion. However, instead of directly joining the relations of $Q$, Yannakakis’s algorithm first eliminates all dangling tuples from the input, i.e., those that will not contribute to the final output, by a series of semijoin operations. The overall algorithm consists of two consecutive phases: (1) a semijoin phase; and (2) a join phase. As shown in reference [132], the dangling tuple elimination in the semijoin phase guarantees that the sizes of all intermediate tables during the join phase are smaller than or equal to the final output. We next discuss the details of each phase.

Semijoin Phase: Consider a GHD $D = (T, \chi, \lambda)$ of an acyclic query $Q$. The semijoin phase operates recursively as follows.

BASIS: If $T$ is a single vertex, do nothing.

INDUCTION: If $T$ has more than one vertex, pick a leaf $t$ that is assigned relation $R$, and let $S$ be the relation assigned to $t$’s parent.

1. Replace $S$ by the semijoin of $S$ with $R$, $S \bowtie R = S \bowtie \pi_{R \cap S}(R)$.
2. Recursively process $T \setminus R$.
3. Compute the final value of $R$ by computing its semijoin with the value of $S$ that results from step (3); that is, $R := R \bowtie S$.

The executions of step (1) in this recursive algorithm form the upward phase, and the executions of step (4) form the downward phase. In total, this version of the algorithm performs $2(n-1)$ semijoin operations. For example, for the GHD in Figure 4.5b, the 8 semijoins could be: (1) $BCD \bowtie ABC$; (2) $BCD \bowtie BF$, (3) $CDE \bowtie BCD$; (4) $CDE \bowtie DEG$; (5) $DEG \bowtie CDE$; (6) $BCD \bowtie CDE$; (7) $BF \bowtie BCD$; and (8) $ABC \bowtie BCD$. As argued in [132], the semijoin phase guarantees that all dangling tuples are eliminated.

Join Phase: Next, Yannakakis’s algorithm performs a series of $(n - 1)$ joins,
in any bottom-up order on $T$.

**Example 4.4.1.** One possible choice of bottom-up join executions for the GHD of Figure 4.5b could be:

1. $\text{Int}_1 = R_1 \bowtie R_3$
2. $\text{Int}_2 = R_2 \bowtie \text{Int}_1$
3. $\text{Int}_3 = R_5 \bowtie R_4$
4. $O = \text{Int}_2 \bowtie \text{Int}_3$

where $O$ is the final output of the join.

### 4.4.2 DYM-n

We now show that Yannakakis’ algorithm can be distributed in MR in a straightforward manner. We refer to this algorithm as **DYM-n**. Yannakakis’ algorithm involves multiple semijoins or joins of two relations at a time. To perform each pairwise join and semijoin, we can either use the Grouping algorithm or the Shares algorithm with appropriate share parameters. We explain our analysis using the Grouping algorithm.

Recall from Section 2.5.4 of Chapter 2 that the Grouping algorithm joins any two relations $R$ and $S$ in a single round as follows. We divide $R$ and $S$ into $g_r = \frac{2|R|}{M}$ and $g_s = \frac{2|S|}{M}$ disjoint groups of size $\frac{M}{2}$ each. Then, we use a total of $g_rg_s$ machines and send a distinct pair of groups to each machine, which joins its groups locally. This computation incurs a communication cost of $O\left(\frac{|R||S|}{M}\right) = O\left(\frac{(|R|+|S|)^2}{M}\right) = B(\text{IN}, M)$, where $\text{IN} = |R| + |S|$.

A semijoin is simply a join followed by a projection. So for computing $S \bowtie R$, we first join $S$ and $R$ in a single round. Then each reducer locally projects its tuples onto attributes of $S$. But each tuple of $S$ may join with multiple $R$ tuples, creating duplicates on up to $g_r = \frac{2|R|}{M}$ machines (up to one per group of $R$). We can eliminate the duplicates in $O(1)$ rounds as follows: each machine hashes its $\leq M$ output tuples into $\frac{|S|}{\sqrt{M}}$ buckets on the attributes of $S$. Then it sends each bucket to a distinct machine. Now each hash bucket contains about $\sqrt{M}$ unique tuples. For each such bucket, there are up to $g_r$ machines containing outputs for that bucket. In each subsequent round, $\sqrt{M}$ machines corresponding to the same bucket send their
Chapter 4. Multiround MapReduce

\( (\approx \sqrt{M}) \) outputs to one machine (so as to not exceed machine size \( M \)), which locally de-duplicates them. Since we combine outputs from \( \sqrt{M} \) machines in each round, the deduplication requires up to \( 1 + \log{\sqrt{M}(g_r)} \) rounds. Since \( |R| \leq \text{IN} \) and \( \epsilon = \log{M(\text{IN})} \), the number of rounds required for the semijoin is \( O(\epsilon) = O(1) \). The communication cost of the first round is \( (|R| + |S|)^2 \). The communication cost of the following rounds is simply equal to \( \text{IN} = |R| + |S| \). We now state the following theorem:

**Theorem 4.4.1.** DYM-n can compute every acyclic query \( Q = R_1 \bowtie ... \bowtie R_n \) in \( O(n) \) rounds of MR with \( O(nB(\text{IN} + \text{OUT}, M)) \) communication cost.

**Proof.** For each edge in \( T \), there are exactly two semijoin operations, once in the upward phase and once in the downward phase, and one join operation. The algorithm therefore executes a total of \( 3(n-1) \) pairwise joins and semijoins, in a total of \( O(n) \) MR rounds. The communication cost of each MR semijoin job \( R \bowtie S \) is \( O(B(|R| + |S|, M)) \). Since there are \( 2(n-1) \) semijoins and the largest input to any semijoin operation is the largest relation size, i.e., \( \max_i{|R_i|} \leq \text{IN} \), the total cost of the semijoin phase is \( O(nB(\text{IN}, M)) \). During the join phase, the sizes of the semijoined input relations and the intermediate relations generated are less than or equal to the size of the final output because there are no dangling tuples. Since there are \( n-1 \) join operations, the total cost of the join phase is \( O(nB(\text{OUT}, M)) \). Therefore the sum of the costs of both phases is \( O(nB(\text{IN} + \text{OUT}, M)) \), completing the proof.

### 4.4.3 DYM-d

In addition to parallelizing each semijoin and join operation, we can parallelize Yan- nakakis’s algorithm further by executing multiple semijoins and joins in parallel. With this extra parallelism, we can reduce the number of rounds to \( O(d + \log(n)) \), where \( d \) is the depth of the GHD \( D(T, \chi, \lambda) \), without asymptotically affecting the communication cost of the algorithm. We refer to this modified version of DYM-n as DYM-d.

**Upward Semijoin Phase in \( O(d + \log(n)) \) Rounds:** Consider any leaf vertex \( R \) of \( T \), with parent \( S \). During the upward semijoin phase, we replace \( S \) with \( S \bowtie R \) in \( O(1) \) rounds. But instead of using the rounds to only process \( R \), we can
process all leaves in parallel in each step, reducing the total number of rounds. We
now give a recursive procedure for performing the semijoin phase. Our input is a
GHD $D = (T, \chi, \lambda)$:

**BASIS:** If $T$ is a single vertex, do nothing.

**INDUCTION:** If $T$ has more than one vertex, consider the set $L$ of leaves of $T$. Let
$L_1$ be the set of leaves that have no siblings, and let $L_2$ be the remaining leaves.

1. For each $R$ in $L_1$ with parent $S$, replace $S$ with $S \prec R$, and remove $R$ from the
tree for the duration of the upward semijoin phase.

2. Divide the leaves in $L_2$ into disjoint pairs of siblings, and up to one triple of
siblings, if there is an odd number of siblings with the same parent. Suppose $R_1$
and $R_2$ form such a pair with parent $S$. Then replace $R_1$ with $(S \prec R_1) \cap (S \prec R_2)$
and remove $R_2$, for the duration of the upward semijoin phase. If there is a triple
$R_1, R_2, R_3$, replace $R_1$ with $(S \prec R_1) \cap (S \prec R_2) \cap (S \prec R_3)$ (using two pairwise
intersections) and remove $R_2$ and $R_3$.

3. Recursively process the resulting $T$.

Steps (1) and (2) above can be performed in $O(1)$ rounds, in parallel, for all leaves.
Moreover, the number of recursive calls made by the above procedure is $O(d + \log(n))$, as we next prove.

Let $X = \sum_{l \in L(T)} 2^{d(l)}$, where $L(T)$ stands for the leaves in the (remaining) $T$
during the above procedure, and $d(l)$ is the depth of leaf $l$. Then, $X$ is at most $n2^d$
in the beginning, as there are initially at most $n$ leaves with depth at most $d$. Now
consider what happens to $X$ in each recursive call. Each leaf $l$ is in either $L_1$ or $L_2$. If
it is in $L_1$, it gets deleted. If $l$’s parent has no other children, then the parent becomes
a new leaf, of depth $d(l) - 1$. Thus the $2^{d(l)}$ term in $X$ is at least halved for all leaves
in $L_1$. On the other hand, if $l_1, l_2$ form a pair in $L_2$, then one of them gets deleted,
while the other stays at the same depth. Thus the $2^{d(l_1)} + 2^{d(l_2)}$ term also gets halved.
For a triple in $L_2$, the term becomes one-third. Thus $X$ reduces by at least half in
each recursive call. Since its starting value is at most $n2^d$, the number of recursive
calls (and hence the number of rounds), is $O(\log(n2^d)) = O(d + \log(n))$. 
Additionally, since we perform $O(n)$ intersection or semijoin operations in total, the total communication cost of the upward semijoin phase is $O(nB(IN, M))$, as all initial and intermediate relations involved have size at most IN.

**Downward Semijoin Phase in $O(d)$ Rounds:** In the downward phase, the algorithm semijoins each child relation with its parent. Note however that the semijoins of the children relations with the same parent are independent and can be done in parallel in $O(1)$ rounds. Thus we can perform the downward phase in $O(d)$ rounds and in $O(nB(IN, M))$ communication. We note that the downward join phase can be performed in $O(d)$ rounds whereas the upward semijoin phase requires $O(d + \log(n))$ rounds. The reason for this is that in the downward semijoin phase, each relation is semijoined with a single parent. In contrast, in the upward semijoin phase each relation may be joined with multiple children relations, which requires more rounds of computation.

**Join Phase in $O(d + \log(n))$ Rounds:** The join phase is similar to the upward semijoin phase. The only difference is, we compute $S \bowtie R$ instead of $S \bowtie R$ for $R \in L_1$, and $(R_1 \bowtie S) \bowtie (R_2 \bowtie S)$ for pair $R_1, R_2 \in L_2$. The total number of rounds required is again $O(d + \log(n))$. The total communication cost of each pairwise join is $O(nB(OUT, M))$, since the intermediate relations being joined may be as large as OUT but no larger. Therefore, both the semijoin and join phases can be performed in $O(d + \log(n))$ rounds with a total communication cost of $O(nB(IN + OUT, M))$, justifying the following theorem:

**Theorem 4.4.2.** DYM-d can compute every acyclic query $Q = R_1 \bowtie \ldots \bowtie R_n$ in $O(d + \log(n))$ rounds of MR and $O(nB(IN + OUT, M))$ communication cost, where $d$ is the depth of a width-1 GHD $D(T, \chi, \lambda)$ of $Q$.

We note that in general, the depth of any minimum-width GHD for some acyclic queries can be $\Theta(n)$. As an example, recall the Chain-n query: $R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie \cdots \bowtie R_n(A_n, A_{n+1})$. The lowest depth width-1 GHD of Chain-n has a depth of $\frac{n}{2}$ as shown in Figure 4.6a. It can be shown that there is no shorter depth width-1 GHD for $C_n$. However, other acyclic queries, such as the Snowflake-n query,
Consider a GHD $D(T, \chi, \lambda)$ of a query $Q$ where the width and depth of $D$ are $w$ and $d$, respectively, and $|V(T)| = O(n)$. Note that the width of $Q$ may be strictly less than $w$ as it may have other GHDs with smaller width. In this section, we show that $Q$ can be computed in $O(nB(\text{IN}^w + \text{OUT}, M))$ communication and $O(d + \log(n))$ rounds of MR using an extension of DYM-d, which we call GYM.
CHAPTER 4. MULTIROUND MAPREDUCE

1 Input: GHD $D(T, \chi, \lambda)$ of a query $Q$

2 **Materialization Stage:**

3 foreach vertex $v$ in $D$ (in parallel):
4   Compute $\text{IDB}_v = \Join_{R_i \in \lambda(v)} R_i$ by the Shares algorithm.

5 foreach $R_j$ that is not in the $\lambda(v)$ for any $v$ (in parallel):
6   Let $\text{tempAss}(R_j)$ be a vertex $u$ s.t. $\chi(u) \supseteq$ attributes of $R_j$.
7   Compute $\text{IDB}_{\text{tempAss}(R_j)} = \Join u \Join R_j$

8 foreach $v$ s.t. $v = \text{tempAss}(R_j)$ for any $R_j$ (in parallel):
9   Compute $\text{IDB}_v = \Join \text{IDB}_{\text{tempAss}(R_j)}$ in $O(1)$ rounds of MR

10 **Yannakakis Stage:**
11   Let $Q' = \Join u \Join \text{IDB}_u$.
12   Execute DYM$-d$ on $Q'$.

---

**Figure 4.7:** GYM.

### 4.5.1 Overview of GYM

Assume for simplicity that each relation $R_i$ is assigned to some vertex $v \in V(T)$ of $D$ and consider “materializing” each $v \in V(T)$ by computing $\text{IDB}_v = \Join_{R_i \in \lambda(v)} R_i$. Now, consider the query $Q' = \Join_{v \in V(T)} \text{IDB}_v$. Note that $Q'$ has the exact same output as $Q$. This is because $Q'$ is also the join of all $R_i$, where some $R_i$ might (unnecessarily) be joined multiple times if they are assigned to multiple vertices.

However, observe that $Q'$ is now an acyclic query. In particular, after materializing each $\text{IDB}_v$, $D$ is now a width-1 GHD for $Q'$. Therefore we can directly run DYM$-d$ to compute $Q'$ with communication in $O(B(\sum_v |\text{IDB}_v| + \text{OUT, } M))$. Based on this observation, GYM, shown in Figure 4.7, consists of two stages:

**Materialization Stage:** Materializes each vertex $v \in V(T)$ using the Grouping algorithm.

**Yannakakis Stage:** Executes DYM$-d$ from Section 4.4 on the materialized GHD $D$.

The algorithm does not assume that each $R_i$ is assigned to some $v$ and ensures that each $R_i$ appears in the transformed $Q'$ during the materialization stage. We will discuss this technicality in our analysis of GYM in the next section.
4.5.2 Analysis of GYM

We start this section by stating our first main result:

**Theorem 4.5.1.** Given a GHD \( D(T, \chi, \lambda) \) of a query \( Q \) where the width and depth of \( D \) are \( w \) and \( d \), respectively, GYM executes \( Q \) in \( O(d + \log(n)) \) rounds and \( O(|V(T)|B(IN^w + OUT, M)) \) communication cost.

**Proof.** We start with the materialization stage. First, for each \( v \), the algorithm computes an initial \( IDB_v \) by joining the relations assigned to \( v \) (line 4). Now, there may be some relations that have not been assigned to any \( v \). So the algorithm next ensures that each such \( R_i \) appears in the final join. Let \( R_j \) be such a relation. We know by the definition of a GHD, that there is a vertex \( v \) whose attributes contain the attributes of \( R_j \). The algorithm then joins each such \( R_j \), in parallel, with its \( IDB_v \) to get \( IDB_{vj} \) (lines 5–7). Finally, if there are multiple \( IDB_{vj} \) relations for a particular \( v \), the algorithm joins them together to compute the final version of \( IDB_v \) (lines 8–9).

We next calculate the cost of each step of the materialization stage.

1. **Computing Initial \( IDB_v \)'s:** Since the width of \( D \) is \( w \), we join at most \( w \) relations for each \( v \). We assume the worst case scenario when the relations do not have any common attributes and the computation is a Cartesian product of the \( w \) relations. In this case, no matter what the parallelism level is, Grouping will have a cost of \( \text{IN}^w \). Therefore, we can perform this step in one round with a total cost of \( O(|V(T)|\text{IN}^w) \).

2. **Computing \( IDB_{vj} \)'s:** Note that this is essentially a semijoin operation filtering some tuples from \( IDB_v \), since each attribute of \( R_j \) is contained in the attributes of \( IDB_v \). Therefore each \( IDB_{vj} \) can be computed in \( O(1) \) rounds in \( O(B(|IDB_v| + |R_j|, M)) \) communication cost. The size of each \( IDB \) is \( \text{IN}^w \), so a loose bound on the cumulative cost of computing all \( IDB_{vj} \)'s is \( O(|V(T)|B(\text{IN}^w, M)) \).

3. **Computing Final \( IDB_v \)'s:** Since each \( IDB_{vj} \) has the same attributes, this is essentially an intersection operation, which can be performed in one round with a communication cost of again \( O(B(|IDB_{vj}|, M)) \)'s. Cumulatively, we can also bound this cost as \( O(|V(T)|B(\text{IN}^w, M)) \).
Therefore, the materialization stage takes $O(1)$ rounds and $O(|V(T)|B(IN^w, M))$ communication cost. Executing DYM-d on the $IDB_v$’s takes $O(d + \log(n))$ rounds and $O(|V(T)|B(IN^w + OUT, M))$ cost by Theorem 4.4.2 and the fact that the size of each $IDB_v$ is at most $IN^w$. Therefore GYM takes $O(d + \log(n))$ total rounds of MR, and $O(|V(T)|B(IN^w + OUT, M))$ cost.

An immediate corollary to Theorem 4.5.1 is the following:

**Corollary 4.5.1.** Any width-$w$ query can be computed in $O(n)$ rounds of MR and $O(nB(IN^w + OUT, M))$ communication cost.

**Proof.** The proof is immediate from Theorem 4.5.1 and Lemma 4.3.1 that states that any width-$w$ query has a GHD $D$ with at most $n$ vertices, which implies that $D$ has $O(n)$-depth.

We finish this subsection with two notes. First, one can show that there are queries with width $w$ whose GHDs have depth $\Theta(w^2)$, therefore causing GYM to execute for a large number of rounds. Second, in practice, when we compute the $IDB_v$’s, we might in the last step of the materialization stage, do a projection onto the attributes assigned to vertex $v$, i.e., $\chi(v)$, to save communication.

### 4.5.3 Example Execution of GYM

We finish this section by describing how to compute an example query with GYM. Consider the following chain query Chain-16: $R_0(A_0, A_1) \bowtie R_1(A_1, A_2) \bowtie ... \bowtie R_{15}(A_{15}, A_{16})$. Figure 4.8 shows a width-3 GHD for this query. GYM on this GHD would first compute the IDBs in each vertex of Figure 4.8. The materialized GHD, shown in Figure 4.9, is now a width-1 GHD over the IDBs and therefore the join over the IDBs is acyclic. Then, the algorithm simply executes DYM-d on the GHD of Figure 4.9 to compute the final output. Let $c$ be the (constant) number rounds to process the semijoin of two relations. Overall the algorithm would take $12c + 6$ rounds and $O(B(IN^3 + OUT, M))$ communication cost. For comparison, Figure 4.10 shows a width-1 GHD of the original chain query. Executing GYM directly on this GHD would take $32c + 16$ rounds and $O(B(IN + OUT, M))$ communication cost.
CHAPTER 4. MULTIROUND MAPREDUCE 105

Figure 4.8: Width-3 GHD of Chain-16.

Figure 4.9: After Computing IDBs of the GHD of Figure 4.8.

Figure 4.10: Width-1 GHD of Chain-16.
4.6 Constructing $O(\log(n))$-depth GHDs

We now describe our \textbf{Log-GTA} algorithm (for \textbf{Log}-depth GHD Transformation Algorithm) which can take any hypergraph $H$ and its GHD $D(T, \chi, \lambda)$ with width $w$ and construct another GHD $D^*$ of $H$ that has $O(\log(|V(T)|))$ depth and whose width is at most $3w$. This result implies that we can construct a $\log(n)$-depth GHD for any query, which has at most three times the width of the query (recall the width of the query is defined to be the minimum width of any GHD of the query). For an acyclic query $Q$, this result implies that we can construct a $O(\log(n))$-depth GHD of $Q$, whose width is at most 3. Therefore using GYM, we can execute $Q$ in $O(\log(n))$ rounds with $O(\log(n)B(IN^3 + OUT, M))$ communication.

Given a GHD $D(T, \chi, \lambda)$ of a hypergraph $H$, Log-GTA iteratively transforms it into a GHD $D^*(T^*, \chi^*, \lambda^*)$. For simplicity, we refer to all GHDs during the transformation as $D'(T', \chi', \lambda')$. In other words, $D' = D$ in the beginning and $D' = D^*$ at the end of the transformation.

Here is the outline of this section. In Section \ref{sec:extend_D_prime}, we describe some additional metadata that are assigned to the vertices and edges of $T'$ by Log-GTA. In Section \ref{sec:unique_child_grandchild}, we define \textit{unique-child-and-grandchild} vertices. These are one of the two types of vertices that will be modified in each iteration of Log-GTA. In Section \ref{sec:leaf_unique_child_grandchild_inactivations}, we describe the two transformation operations of Log-GTA: \textit{leaf and unique-child-grandchild inactivations}, which will be iteratively performed to modify $D'$. Finally, in Section \ref{sec:log_gta_algorithm}, we describe the entire Log-GTA algorithm.

4.6.1 Extending D’

Log-GTA associates two new labels with the vertices of $T'$:

1. \textbf{Active/Inactive}: Indicates whether $v$ will be modified in later iterations of Log-GTA. Every vertex is active in the beginning and inactive in the end. Once a vertex becomes inactive, it remains inactive until the end of the transformation. We will refer to the subtree of $T'$ that consists only of active vertices and the edges between them as \textbf{active($T'$)}. We will prove that $\text{active}(T')$ is indeed a tree (Lemma \ref{lem:active_subtree_tree}).
2. **Height**: Assigned to each inactive vertex \( v \) when \( v \) becomes inactive. The value of \( v \)'s height will be \( v \)'s height in \( T' \) in the iteration that \( v \) becomes inactive and, as we prove, all future iterations (Corollary 4.6.1). In particular, it will be \( v \)'s height in \( T^* \), i.e., the final tree at the end of the transformation. That is, if \( v \) is a leaf in \( T^* \), then \( v \)'s height will be 0. Similarly, if \( v \)'s longest path to a leaf in \( T^* \) has length \( l \), then \( v \)'s height will be \( l \).

In addition, Log-GTA associates a label with each “active” edge \( (u, v) \in E(\text{active}(T')) \):

- **Common-cover** \( (u, v) \): Is a set \( S \subseteq E(H) \) such that \( (\chi(u) \cap \chi(v)) \subseteq \bigcup s \). In words, the \( \text{cc}(u, v) \) is a subset of the hyperedges of \( H \) whose vertices cover the vertices that are shared by both \( \chi(u) \) and \( \chi(v) \). In query terms, the \( \text{cc}(u, v) \) is a set of relations whose attributes cover the common attributes between \( u \) and \( v \). Initially, in the original \( D(T, \chi, \lambda) \), for each \( (u, v) \), we set \( \text{cc}(u, v) \) simply to \( \lambda(v) \). We can also set \( \text{cc}(u, v) \) to \( \lambda(u) \) without affecting any of our results. Therefore, the size of each \( \text{cc}(u, v) \) is equal to \( w \). We will show that throughout Log-GTA, the size of common covers will be \( w \) for each edge between active vertices, including those that Log-GTA introduces (Lemma 4.6.3).

### 4.6.2 Unique-Child-And-Grandchild Vertices

Consider a tree \( T \) of \( n \) vertices with a very long depth, say, \( \Theta(n) \). Intuitively, such long depths are caused by long chains of vertices, where vertices in the chain have only a single child. Log-GTA will try to shorten long-depth GHDs by identifying and “branching out” such chains. At a high-level, Log-GTA will find a vertex \( v \) with a single child \( c \) (for child), which also has single child \( gc \) (for grandchild), and put \( v, c, \) and \( gc \) under a new vertex \( s \). We call vertices like \( v \) unique-child-grand-child (unique-c-gc) vertices. Figure 4.11 shows an example tree and two unique-c-gc vertices of the tree, which are drawn inside boxes.

In each iteration Log-GTA will identify a set of nonadjacent unique-c-gc vertices and some leaves of active(\( T' \)), and inactivate them (along with shortening the chains of unique-c-gc vertices). We next state an important theorem that lower bounds the
number of leaves and nonadjacent unique-c-gc vertices in any tree. This theorem will
be used to bound the number of iterations that Log-GTA executes in Section 4.6.4.

**Theorem 4.6.1.** If a tree has \( N \) vertices, then at least \( \frac{N}{4} \) vertices are either leaves or non adjacent unique-c-gc vertices.

We first discuss two lemmas that the proof of Theorem 4.6.1 depends on and then
prove Theorem 4.6.1.

**Lemma 4.6.1.** If a tree has \( U \) unique-c-gc vertices, we can select at least \( \lceil \frac{U}{2} \rceil \) of
them that are not parent and child of each other.

**Proof.** Partition the \( U \) unique-c-gc vertices into disjoint chains; some or all of the
chains may be of length two. We can select the first, third, fifth, and so on, of any
chain, and thus we select at least \( \lceil \frac{U}{2} \rceil \) vertices. \( \Box \)

**Lemma 4.6.2.** Suppose a tree has \( N \) vertices, \( L \) of which are leaves, and \( U \) of which
are unique-child vertices. Then \( 4L + U \geq N + 2 \).

**Proof.** The proof is an induction on the height \( h \) of the tree.
BASIS: If \( h=0 \), then the root is the only vertex in the tree and is a leaf. Therefore,
\( 4L + U = 4 \geq 1 + 2 = 3 \). If \( h = 1 \), then the tree is a root plus \( N - 1 \) children of
the root, all of which are leaves. Thus, \( L = N - 1 \), and \( U = 0 \). We must verify
\( 4(N - 1) + 0 \geq N + 2 \), or \( 3N \geq 6 \). Since \( N \) is necessarily larger than 2 for any tree
of height at least 1, we may conclude the bases.
INDUCTION: Now, assume \( h \geq 1 \). There are three cases to consider:

**Case 1:** The root has a single child \( c \) and \( c \) has a single child \( gc \). Then the root is a unique-c-gc vertex and the tree rooted at \( c \) has \( L \) leaves, \( U - 1 \) unique-c-gc vertices, and a total of \( N - 1 \) vertices. By the induction hypothesis, \( 4L + U - 1 \geq (N - 1) + 2 \), or \( 4L + U \geq N + 2 \), which completes the induction in this case.

**Case 2:** The root has a single child, which has \( k \geq 2 \) children \( c_1, ..., c_k \). Let the subtree rooted at \( c_i \) have \( L_i \) leaves, \( U_i \) unique-child vertices, and \( N_i \) vertices. By the inductive hypothesis, \( 4L_i + U_i \geq N_i + 2 \). Summing over all \( i \) we get \( 4L + U \geq N + 2 \), which completes the induction in this case.

**Case 3:** The root has \( k \geq 2 \) children \( c_1, ..., c_k \). Similarly, if the subtree rooted at \( c_i \) has \( L_i \) leaves, \( U_i \) unique-child vertices, and \( N_i \) vertices and we sum over all \( i \), we get \( 4L + U \geq (N - 1) + 2k \). Since \( k \geq 2 \), we again conclude that \( 4L + U \geq N + 2 \), which completes the proof.

Using Lemmas 4.6.1 and 4.6.2, we can now prove Theorem 4.6.1.

**Proof of Theorem 4.6.1.** Let the tree have \( N \) vertices, \( L \) leaves, and \( U \) unique-c-gc vertices. Lemma 4.6.2 says that \( 4L + U \geq N + 2 \). Suppose first that \( U = 0 \). Then since we can select all leaves, and \( 4L \geq N + 2 \), we can surely select at least \( N/4 \) vertices. Now, suppose \( U \geq 1 \), then by Lemma 4.6.1 we can select at least \( U/2 \) of the unique-child vertices. Since we may also select all leaves, and \( L + U/2 \geq L + U/4 \geq N/4 + 2/4 \), the theorem holds in both cases.

### 4.6.3 Two Transformation Operations

We next describe the two operations that Log-GTA performs on the vertices of \( \text{active}(T') \) during the transformation.

**Leaf Inactivation:** Takes a leaf \( l \) of the active \( (T') \) and (1) changes its label to inactive; and (2) sets its height \( (l) \) to 0 if \( l \) has no inactive children or to \( \max \{ \text{height}(c) \} + 1 \), where \( c \) is over the inactive children of \( l \). \( \chi(l) \) and \( \lambda(l) \) remain the same. The common-cover of the edge between \( l \) and \( l \)'s parent is removed. Figure 4.12 shows the effect of this operation on vertex \( v_4 \) of an extended GHD. In the figure, green
and red indicate that the vertex is active or inactive, respectively. In the figure, the attributes of each \( R_i \) are the \( \chi \) values on the vertices that \( R_i \) is assigned to.

**Unique-c-gc (And Child) Inactivation**: Takes a unique-c-gc vertex \( u \), \( u \)'s parent \( p \) (if one exists), \( u \)'s child \( c \), and \( u \)'s grandchild \( gc \), in active(\( T' \)) and performs the following actions:

1. Creates a new active vertex \( s \), where \( \chi(s) = (\chi(p) \cap \chi(u)) \cup (\chi(u) \cap \chi(c)) \cup (\chi(c) \cap \chi(gc)) \); \( \lambda(s) = cc(u, p) \cup cc(u, c) \cup cc(c, gc) \).
2. Inactivates \( u \) and \( c \). Similar to leaf inactivation, sets their heights to 0 if they have no inactive children, and one plus the maximum height of their inactive children otherwise.
3. Removes the edges \( (p, u) \) and \( (u, c) \) and adds an edge from \( s \) to both \( u \) and \( c \).
4. Adds an edge from \( p \) to \( s \) with \( cc(p, s) = cc(p, u) \) and \( s \) to \( gc \) with \( cc(s, gc) = cc(c, gc) \).

Figure 4.13 visually shows the effect of this operation when inactivating the unique-c-gc vertex \( v_1 \) from Figure 4.12. We next prove a key lemma about these two operations:

**Lemma 4.6.3.** Assume an extended GHD \( D'(T', \chi', \lambda') \) of width \( w \), active/inactive labels on \( V(T') \), and common cover labels on \( E(T') \) initially satisfies the following five properties:
1. The active($T'$) is a tree.

2. The subtree rooted at each inactive vertex $v$ contains only inactive vertices.

3. The height of each inactive vertex $v$ is $v$'s correct height in $T'$.

4. $|cc(u, v)| \leq w$ between any two active vertices $u$ and $v$ and does indeed cover the shared attributes of $u$ and $v$.

5. $D'$ is a GHD with width at most $3w$.

Then performing any sequence of leaf and unique-c-gc inactivations maintains these five properties.

Proof. Let $D'(T', \chi', \lambda')$ be a GHD that satisfies these five properties. First, consider inactivating an active leaf $l$ of $D'$.

1. For property (1), we observe that inactivating $l$ essentially removes a leaf of active($T'$), so active($T'$) remains a tree after the operation.

2. For property (2), we only need to consider the subtree $S_l$ rooted at $l$. Observe that none of $l$'s children can be active, since this would contradict the claim that $l$ is a leaf of active($T'$). In addition, none of $l$'s other descendants can be active because then the subtree rooted at one of $l$'s inactive children would contain an active vertex. This would contradict the assumption that initially all subtrees rooted at inactive vertices contained only inactive vertices.

3. For property (3), notice that the height that is assigned to $l$ is 0 if it has no children, which is its correct height in $T'$. Otherwise, $l$'s height is one plus the maximum of the heights of $l$'s children, which is also its correct height in $T'$ since all of $l$'s children are inactive and have correct heights by assumption.

4. Properties (4) and (5) hold trivially as leaf inactivation does not affect the common-covers, $\chi$, and $\lambda$ values and by assumption their properties hold in $D'$.

Now let’s consider the unique-c-gc inactivation operation.

1. Property (1) holds because by definition $u$ has one active child $c$ and $c$ also has one active child $gc$. So $u$ and $c$ are part of a chain of active($T'$). We effectively merge $u$ and $c$ together into another active vertex $s$ on this chain without affecting the acyclicity or connectedness of active($T'$). Notice that we also add two edges from $s$ to $u$ and $v$ but $u$ and $v$ are inactive.
2. For property (2) observe that the only two subtrees we need to consider are the subtrees rooted at \( u \) and \( c \), which we call \( S_u \) and \( S_c \), respectively. Notice that all of the edges that go down the tree from \( u \) and \( c \) after removing \((u, c)\) and \((c, gc)\) were to inactive vertices. Therefore, by the same argument we did for leaf elimination, both \( S_u \) and \( S_c \) have to consist of only inactive vertices.

3. We assign heights to \( u \) and \( c \) in the same way as we assigned the height of an inactivated leaf. The exact same argument we made for leaf elimination proves that \( u \) and \( c \) get assigned their correct heights in \( T' \).

4. We need to consider two common covers: \( cc(p, s) \), which is assigned \( cc(p, u) \) and \( cc(s, gc) \), which is assigned \( cc(c, gc) \). The sizes of \( cc(p, s) \) and \( cc(s, gc) \) are at most \( w \) because the sizes of \( cc(p, u) \) and \( cc(c, gc) \) are at most \( w \) initially by assumption. We next prove that \( cc(p, s) \) indeed covers the common attributes between \( p \) and \( s \). The proof for \( cc(s, gc) \) is similar and omitted. Notice that since \( \chi(s) = (\chi(p) \cap \chi(u)) \cup (\chi(u) \cap \chi(c)) \cup (\chi(c) \cap \chi(gc)) \), \( \chi(s) \cap \chi(p) \) is exactly equal to \( \chi(p) \cap \chi(u) \). This follows from the observation that \( p \) cannot share an attribute with \( c \) (or \( gc \)), say \( A_i \), that it does not share with \( u \), as this would contradict that the subtree containing \( A_i \) in \( D' \) is connected (and therefore contradicting that \( D' \) is a GHD). Therefore \( \chi(p) \cap \chi(s) \) is exactly \( \chi(p) \cap \chi(u) \), which is covered by \( cc(p, u) \) by assumption. Therefore \( cc(p, s) \), which includes \( cc(p, u) \), covers \( \chi(p) \cap \chi(s) \).

Figure 4.13: Unique-c-gc inactivation.
5. For property (5), we need to prove that the three properties of GHDs hold and also verify that the width of the modified $D'$ is at most $3w$.

- **1st property of GHDs:** The addition of $s$ with two edges to $u$ and $c$ cannot create a cycle or disconnect $T'$, and therefore $T'$ is still a tree.

- **2nd property of GHDs:** We need to verify that for each vertex $v$, $\chi(v) \subseteq \cup \lambda(v)$. The unique-c-gc inactivation only inserts the vertex $s$, and by assumption $\chi(s)$ is the union of three intersections, each of which is covered (respectively) by the three common-covers that comprise $\lambda(s)$.

- **Width of the modified GHD:** Again by assumption, the sizes of each common cover in $\lambda(s)$ is at most $w$, therefore $|\lambda(s)|$ is at most $3w$, showing that the width of GHD is still at most $3w$.

- **3rd property of GHDs:** We need to verify that for each attribute $X$, the vertices that contain $X$ must be connected. It is enough to verify that all attributes among $p$, $s$, $u$, $c$, and $gc$ are locally connected, since other parts of $T'$ remain unchanged. We need to consider all possible breaks in connectedness between $p$, $u$, $c$, and $gc$ introduced by the insertion of $s$. The proof of each combination is the same. We only show the proof for attributes between $p$ and $gc$. Consider any attribute $X \in \chi(p) \cap \chi(gc)$. Then, since the initial $D'$ was a valid GHD, $X$ must have been in $\chi(u)$ (and also $\chi(c)$). Then $\chi(s)$ also includes $X$ because $\chi(s)$ includes $\chi(p) \cap \chi(u)$, proving that the vertices of $X$ are locally connected among $p$, $s$, $u$, $c$, and $gc$.

\[\square\]

**Corollary 4.6.1.** Consider any GHD $D(T, \chi, \lambda)$ of a hypergraph $H$ with width $w$, extending it to GHD $D'(T', \chi', \lambda')$ with active/inactive labels, common-covers, and heights as described in Section 4.6.1, and then applying any sequence of leaf and unique-c-gc inactivations on $D'$. Then the resulting $D'$ is a GHD of $H$ with width at most $3w$, where the height of each inactive vertex $v$ is $v$'s actual height in $T'$.

**Proof.** Notice that extending $D$ as described in Section 4.6.1 trivially satisfies the five initial properties. Therefore by Lemma 4.6.3, the resulting $D'$ is a valid GHD with
1 Input: GHD \( D(T, \chi, \lambda) \) for hypergraph \( H \)
2 Extend \( D \) into \( D'(T', \chi', \lambda') \):
3 Mark each vertex active
4 Assign each vertex null heights
5 For each edge \( e = (u, v) \) set \( cc(u, v) = \lambda(v) \)
6 while (there are inactive vertices in \( T' \))
   7 Select at least \( \frac{1}{4} \) of the active vertices that are either
      leaves \( L' \) or non-adjacent unique-c-gc vertices \( U' \)
   8 Inactivate each \( l \in L' \)
   9 Inactivate each \( u \in U' \) and the child of \( u \)
10 return \( D' \)

Figure 4.14: Log-GTA.

width at most \( 3w \) and correct height assignments.

4.6.4 Log-GTA

Finally, we present our Log-GTA algorithm. Figure 4.14 shows the algorithm. Log-GTA takes a GHD \( D \) and extends it into a \( D' \) as described in Section 4.6.1. Then, Log-GTA iteratively inactivates a set of active leaves \( L' \) and nonadjacent unique-c-gc vertices \( U' \) (along with the children of \( U' \)), which constitute at least \( \frac{1}{4} \) of the remaining active vertices in \( T' \), until all vertices are inactive. By Theorem 4.6.1, we know that in each tree we can always select \( \frac{1}{4} \) of the vertices that are either leaves or a set of nonadjacent unique-c-gc vertices. Notice that because the unique-c-gc vertices are nonadjacent, activating any one of them, say \( u \), does not increase the number of active children of other unique-c-gc vertices in \( U' \), so other vertices in \( U' \) remain as unique-c-gc vertices. As a result, the inactivation of all vertices in \( U' \) is a well-defined procedure. Therefore the algorithm essentially performs a sequence of leaf and unique-c-gc inactivations on \( D' \), and by Corallary 4.6.1 we know that the final \( D' \) is a valid GHD of the input hypergraph \( H \) and has width \( 3w \). Figure 4.15 shows a simulation of Log-GTA on the width-1 GHD of the chain query \( R_0(A_0, A_1) \bowtie ... \bowtie R_6(A_6, A_7) \) which has depth 6. Log-GTA produces a width-3 GHD with depth 2. In the figure, we label the selected leaves and unique-c-gc vertices with \( L \) and \( U \), respectively and
omitted the common-cover labels.

We next prove that our algorithm takes \( O(\log(|V(T)|)) \) iterations to finish. We then prove the height of each inactive vertex \( v \) is at most equal to the iteration number at which \( v \) was inactivated.

**Lemma 4.6.4.** Log-GTA takes \( O(\log(|V(T)|)) \) iterations.

*Proof.* Observe that both leaf inactivation and unique-c-gc inactivation decrease the number of active vertices in \( T' \) by 1. Therefore in each iteration the number of active vertices decreases by at least \( \frac{1}{4} \) of the vertices. Initially there are \( |V(T)| \) active vertices, so the algorithm terminates in \( O(\log(|V(T)|)) \) iterations. \( \Box \)

**Lemma 4.6.5.** The height of each inactive vertex \( v \) is at most the iteration number at which \( v \) was inactivated.

*Proof.* By Corollary 4.6.1, the heights assigned to vertices are their correct heights in the final GHD returned. Moreover the height numbers start at 0 in the first iteration.
and increase by at most one in each iteration, therefore the height numbers assigned in iteration $i$ are less than $i$, completing the proof.

Finally we can state our second main result:

**Theorem 4.6.2.** Given any GHD $D(T, \chi, \lambda)$ of a hypergraph $H$ with width $w$, we can construct a GHD $D'(T', \chi', \lambda')$ of $H$ where $w' \leq 3w$, depth($T'$) = min{depth($T$), $O(\log(|V(T)|))$}, and $|V(T')| \leq 2|V(T)|$.

**Proof.** By Corollary 4.6.1 the width of $D'$ is at most $3w$. By Lemmas 4.6.3, 4.6.4 and 4.6.5, the height of each vertex $v$ is $v$’s true height in the graph and is $O(\log(|V(T)|))$, implying that the depth of $T'$ is $O(\log(|V(T)|))$. Also, the leaf and unique-c-gc inactivation operations never increase the depth of the tree, justifying that the depth of the final tree is also at most depth($T$). Finally, Log-GTA increases the number of vertices by one for each unique-c-gc inactivation. Since Log-GTA can make at most $|V(T)|$ unique-c-g-c inactivations, $|V(T')| \leq 2|V(T)|$.

Theorems 4.6.2, 4.5.1, and Lemma 4.3.1 imply that we can execute any width-$w$ query in $O(\log(n))$ rounds and $O(nB(IN^{3w} + OUT, M))$ communication, which we state as a theorem.

**Theorem 4.6.3.** Any query $Q$ with width $w$ can be executed in $O(\log(n))$ rounds of MR and $O(nB(IN^{3w} + OUT, M))$ communication.

**Proof.** By Lemma 4.3.1 any width-$w$ $Q$ has a GHD $D$ with width $w$ and $O(n)$ vertices. By Theorem 4.6.2 we can transform $D$ into a $D'$ with depth $O(\log(n))$ and width at most $3w$. Finally, by Theorem 4.5.1 we can run GYM on $D'$ and compute $Q$ in $O(\log(n))$-rounds in $O(nB(IN^{3w} + OUT, M))$ communication.

A corollary of Theorem 4.6.3 for acyclic queries is the following:

**Corollary 4.6.2.** Any acyclic query can be executed in $O(\log(n))$ rounds of MR and $O(nB(IN^{3} + OUT, M))$ communication.
4.7 C-GTA

We next describe another GHD transformation algorithm called C-GTA, for Constant-depth GHD Transformation Algorithm. C-GTA takes a GHD $D$ of width-$w$ and $n$ vertices and transforms it into a GHD $D'$ of width-$2w$ and $\leq \frac{15n}{16}$ vertices. Therefore, it can potentially shorten the depths of $\Theta(n)$-depth GHDs by a constant fraction.

C-GTA transforms a width-$w$ GHD $D(T, \chi, \lambda)$ using a series of merges. For any two vertices $t_1, t_2 \in V(T)$, we can “merge” them by replacing them with a new vertex $t \in V(T)$ and setting $\chi(t) = \chi(t_1) \cup \chi(t_2)$, $\lambda(t) = \lambda(t_1) \cup \lambda(t_2)$ and setting the neighbors of $t$ in $T$ to be the union of neighbors of $t_1$ and $t_2$. As long as $t_1$ and $t_2$ were either neighbors, or both leaves, $T$ remains a valid GHD tree after the merge operation. And as long as $t_1$ and $t_2$ have not been obtained from the previous merge, the width of the tree stays $\leq 2w$.

C-GTA operates as follows:

1. For each vertex $u$ that has an even number of leaves as children, divide $u$’s leaves into pairs and merge each pair.

2. For each vertex $u$ that has an odd number of leaves as children, divide all but one of the leaves into pairs and merge them, and merge the remaining leaf with $u$.

3. For each vertex $u$ that has a unique child $c$, if $c$ does not have an odd number of leaf children, then merge $u$ and $c$.

If $T$ has $L$ leaves and $U$ unique-c-gc vertices, then the above procedure removes at least half of $\max(L, U)$ vertices $s$ from $T$. Since $L + U \geq \frac{n+2}{4}$ by Lemma 4.6.2, the procedure removes at least $\frac{n+2}{16}$ vertices, and so after all the merges, the resulting tree $T'$ has at most $\frac{15n}{16}$ vertices left, justifying the following lemma:

**Lemma 4.7.1.** If a query $Q$ has a width-$w$ GHD $D = (T, \chi, \lambda)$, then $Q$ also has a GHD $D' = (T', \chi', \lambda')$ with width $\leq 2w$ and $|V(T')| \leq \frac{15|V(T)|}{16}$.

**Proof.** We argued above that applying C-GTA to $D$ increases the width by at most a factor of 2. Moreover, each merge operation of C-GTA maintains the GHD properties. Therefore at termination, C-GTA returns a valid GHD $D'$ with width at most $2w$. \qed
Lemma 4.7.1 implies that we can reduce the number of rounds of some queries with \( \Theta(n) \)-depth GHDs by a constant fraction, at the expense of increasing their communication costs from \( O(n(I^w + OUT)) \) to \( O(n(2^w + OUT)) \). We can also repeatedly use C-GTA and then combine it with Log-GTA and obtain the following lemma:

**Lemma 4.7.2.** If a query \( Q \) has a width-\( w \) GHD \( D = (T, \chi, \lambda) \), then for any \( i \), there exists a GHD \( D' = (T', \chi', \lambda') \) with width \( \leq 3 \times 2^i \times w \) and depth \( \leq \log((\frac{15}{16})^i n) \) \(|V(T')| \leq \frac{15|V(T)|}{16} \).

The proof of Lemma 4.7.2 follows directly from Lemma 4.7.1 and Theorem 4.6.2 and is omitted. By Lemma 4.7.2, we can further tradeoff communication by constructing even lower depth trees than a single invocation of Log-GTA. However, the depth decrease due to C-GTA invocations incurs significantly more width increases compared to the decrease due to the single Log-GTA invocation.

### 4.8 Example of Evaluating a Query with Different GHDs

We next compare the costs of evaluating the Chain-\( n \) query with the Shares algorithm and with different GHDs using GYM. Figure 4.16 demonstrates the performances of these algorithms. The figure is a repetition of Figure 4.1 from the beginning of the chapter. We assume that \( OUT = O(IN) \) in the figure.

- **Shares:** Computing Chain-\( n \) directly with Shares would take only one round but incur a communication cost of \( \frac{IN^2}{M^{1/2}} \). The performance of Shares is drawn as the highest, i.e., most inefficient, curve in Figure 4.16.

- **GYM(D):** As we explained in Section 4.2.1, Chain-\( n \) is a width-1 query, whose width-1 GHDs have a depth of \( \Theta(n) \). An example of a width-1 GHD for the Chain-16 was shown in Figure 4.10. On a similar width-1 GHD \( D \) of Chain-\( n \), GYM would execute for \( 3(n - 1) \) rounds and have a communication cost of \( B(IN + OUT, M) = B(O(IN), M) \). The performance of GYM on \( D \), which we refer to as GYM(D), is depicted as the lowest red curve in Figure 4.16.
• **GYM(C-GTA):** Instead of evaluating Chain-n on $D$, we can first transform $D$ with C-GTA and generate a width-2 GHD $D'$ with depth at most $15n/16$. Executing GYM on $D'$ would take at most $3(15n/16 - 1)$ rounds and have a communication cost of $B(O(IN^2), M)$. The performance of GYM on $D$ is depicted as the second lowest blue curve in Figure 4.16.

• **GYM(Log-GTA):** If $n$ is very large and one prefers to run fewer rounds, we can instead transform $D$ with Log-GTA and generate a $O(\log(n))$-depth width-3 GHD $D''$. Executing GYM on $D''$ would take $O(\log(n))$ rounds and $B(O(IN^3), M)$ cost, which is depicted as the green curve in Figure 4.16. We refer to this combined algorithm as GYM(Log-GTA).

Considering the four options we listed, effectively, one can increase the number of rounds of execution and evaluate Chain-n more efficiently.

### 4.9 Related Work

We reviewed the related work on one-round join algorithms in Chapter 2 and 3. Here we review the ACQ algorithm designed for the PRAM model, multiround MR algorithms for processing joins, and generalized hypertree decompositions.
The ACQ Algorithm: The ACQ algorithm [46] is the most efficient known \( O(\log(n)) \)-step PRAM algorithm for computing constant-width queries. By inventing ACQ, Gottlob et al. have proved that constant-width queries are in the complexity class NC, i.e., computable in \( O(\log(n)) \) PRAM steps. Because MR can simulate the PRAM model, the ACQ algorithm can easily be mapped to MR. We call this algorithm ACQ-MR. Given a width-\( w \) query \( Q \), ACQ-MR executes for \( \Theta(\log(n)) \) rounds with \( O(nB(IN^{3w} + OUT, M)) \) communication cost. If \( Q \) has short-depth GHDs, such as the Snowflake-\( n \) and Star-\( n \) queries, GYM outperforms ACQ-MR in communication cost while also using \( \Theta(\log(n)) \) rounds. On the other hand, if \( Q \) has long-depth GHDs, say of \( \Theta(n) \)-depth, such as the Chain-\( n \) query, then ACQ-MR can execute for exponentially fewer rounds than GYM, though at a higher communication cost. For such queries, we can also match the performance of ACQ-MR exactly with the GYM(Log-GTA) combined algorithm we defined in Section 4.8. In addition, by using our C-GTA transformation algorithm on \( D \), we can also execute queries in fewer rounds than ACQ-MR but with higher communication cost.

We note that because a PRAM can also simulate MR, our GYM(Log-GTA) method also proves that constant-width queries are in NC. We believe this result is interesting within itself, since we recover this positive parallel complexity result by using only a simple variant of Yannakakis’s algorithm, which has been thought to be a sequential algorithm.

Multiround MR Multiway Equijoin Algorithms: The only work other than ours we are aware of that theoretically studies multiround join algorithms in MR-related models is reference [18]. This work proves lower bounds on the number of rounds required to compute queries when the amount of data that each reducer is allowed to receive in each round is at most \( \frac{IN}{p} \), where \( p \) is the number of processors, and \( \epsilon \) is a parameter called the space exponent. The authors provide an algorithm that matches these lower bounds on only matching databases. Recall from Section 4.2 that matching databases contain instances with no skew. Another property of matching databases is that the size of the final output and any intermediate output is at most the size of the input. On non-matching databases however, the algorithm can produce
intermediate results of size $IN^{\Theta(n)}$ for any width-$w$ query, where $IN$ is the input size, and $n$ is the number of relations in the query. On matching databases, our algorithm asymptotically matches these lower bounds in terms of rounds and efficiency. On arbitrary databases, our algorithm can violate their space exponent. However, with a modification, we can be within their space requirements and only a $\log(n)$ factor away from their lower bounds in term of the number of rounds, while keeping intermediate relation sizes bounded by $IN^{3w} + OUT$. The authors of reference [18] also cover the topic of handling skew in a single round of computation in a follow up work [19]. The same skew-handling methods based on broadcasting can be applied to each round of GYM to get an even workload balance across reducers in each round.

Reference [128] designs a query optimizer for an MR based query processing system. The authors consider breaking a multiway join into multiple MR rounds consisting of smaller multiway joins, which can also potentially generate $IN^{\Theta(n)}$ size intermediate relations irrespective of the actual output size. Reference [19] focuses on query planning and optimization for massively parallel RDF queries. RDF data can be thought of as a set of binary relations. The authors try to decompose a query over these binary relations into as flat join plans as possible where each join is a star join. They only experimentally analyze their plans but their join plans could generate $IN^{\Theta(n)}$ size intermediate relations, irrespective of the actual output size. Our GYM algorithm avoids generating such large intermediate relations.

**Generalized Hypertree Decompositions:** GHDs are very useful mathematical structures to characterize the difficulty of computational problems that can be represented as hypertrees, such as joins or constraint satisfaction problems. We reviewed the formal definition of GHDs in Section 4.3. In the context of joins, several prior works study the problem of computing GHDs of queries with small, sometimes minimum, width [31, 51]. Our join algorithms take as input different GHDs of a query; so we assume that a minimum width GHD of a query has already been computed by one of the existing methods. Reference [5] characterizes the complexity of finding a logarithmic depth GHD of a query. However, it does not contain an algorithm for finding such a GHD. In contrast, our work provides a transformation algorithm that
takes as input any GHD $D$ of a query $Q$ and constructs a log-depth GHD $D'$ of $Q$ but with larger width than $D$. In addition, we provide another algorithm for constructing even shorter depth GHDs of $Q$ but also with larger widths.
Chapter 5

GPS: A Graph Processing System

In this chapter, we start studying the problem of processing large-scale graph data in highly parallel shared-nothing systems. The introduction of MapReduce as described in previous chapters, and its open-source implementation Hadoop, has made it simpler to process large-scale record-oriented data. These systems offer automatic scalability to extreme volumes of data, automatic fault-tolerance, and a simple programming interface based around implementing a set of functions. However, it has been recognized [84, 87] that MapReduce is not always suitable when processing data in the form of a large graph. There are two main shortcomings of MapReduce that make it unsuitable for processing large-scale graph data:

- **High I/O Costs:** Many graph algorithms are iterative. As an example, consider the PageRank algorithm [25], in which each vertex is initially assigned a PageRank value and updates this value iteratively by aggregating the PageRank values of its incoming neighbors. However, MapReduce is not an inherently iterative system. Executing such graph algorithms necessitates running a multiround MapReduce algorithm that executes each iteration of the graph algorithm as a separate MapReduce job. Each MapReduce job typically reads the entire graph data and the latest values of vertices from disk and also writes the latest vertex values to disk as output, incurring high I/O costs.
• **Lack of a Graph-specific API:** Many graph algorithms are naturally expressed as vertices communicating with other vertices of the graph. For example, in the PageRank algorithm, each vertex can naturally be viewed as receiving the latest PageRank values of its incoming neighbors as messages. Expressing such algorithms in MapReduce requires simulating such vertex-to-vertex communication within `map()` and `reduce()` functions, which can be difficult to program.

Due to these shortcomings, it was recognized [84, 87] that a framework similar to MapReduce—scalable, fault-tolerant, easy to program—but geared specifically towards graph data, would be of immense use. Google’s proprietary Pregel system [87] was developed for this purpose. Pregel is a *distributed message-passing system*, in which the vertices of the graph are distributed across compute nodes and send each other messages to perform the computation. We refer the reader to Chapter [1] for an overview of Pregel’s computational model and API. The rest of this thesis covers our work on processing large-scale graph data in systems that are modeled after Google’s Pregel, which we refer to as *Pregel-like systems*.

We begin our discussion in this chapter by describing a robust open-source clone of Pregel we built from scratch called *GPS*, for *Graph Processing System*. In addition to being open-source, GPS has three new features that do not exist in Pregel, all of which are described in more detail later in the chapter:

1. Only “vertex-centric” algorithms can be implemented easily and efficiently with the Pregel API. The GPS API has an extension that enables efficient implementation of algorithms composed of one or more vertex-centric computations, combined with global computations.

2. Unlike Pregel, GPS can repartition the graph dynamically across compute nodes during the computation, to reduce communication.

3. GPS has an optimization called *large adjacency-list partitioning (LALP)*, which partitions the adjacency lists of high-degree vertices across compute nodes, again to reduce communication.
In the remainder of this introductory material we motivate GPS’s new features. Then we outline our experiments demonstrating how different ways of partitioning, and possibly repartitioning, graphs across compute nodes affects the performance of algorithms running on GPS. Finally we give a roadmap for the rest of the chapters of this thesis.

Master.compute()
Recall that the original API of Pregel consists of a single vertex.compute() function, inside which vertices can update their values and send messages to each other. Implementing a graph computation inside vertex.compute() is ideal for certain algorithms, such as computing PageRank, finding shortest paths, or finding connected components, all of which can be performed in a fully “vertex-centric” and hence parallel fashion. However, some algorithms are a combination of vertex-centric (parallel) and global (sequential) computations. As an example, consider the following k-means-like graph clustering algorithm that consists of four parts: (a) pick $k$ random vertices as “cluster centers,” a computation global to the entire graph; (b) assign each vertex to a cluster center, a vertex-centric computation; (c) assess the goodness of the clusters by counting the number of edges crossing clusters, a vertex-centric computation; (d) decide whether to stop, if the clustering is good enough, or go back to (a), a global computation. We can implement global computations inside vertex.compute() by designating a “master” vertex to run them. However, this approach has two problems: (1) The master vertex executes each global computation in a superstep in which all other vertices are idle, wasting resources. (2) The vertex.compute() code becomes harder to understand, since it contains some sections that are written for all vertices and others that are written for the special vertex. To incorporate global computations easily and efficiently, GPS extends the API of Pregel with an additional function, master.compute(), explained in detail in Section 5.1.4.
GPS’s Partitioning Features

In GPS, as in Pregel, messages between vertices residing in different compute nodes are sent over the network. Two new features of GPS in addition to `master.compute()` are designed to reduce the network I/O resulting from such messages. First, GPS can optionally repartition the vertices of the graph across compute nodes automatically during the computation, based on their message-sending patterns. GPS attempts to colocate vertices that send each other messages frequently. Second, in many graph algorithms, such as PageRank and finding connected components, each vertex sends the same message to all of its neighbors. If, for example, a high-degree vertex `v` on compute node `i` has 1000 neighbors on compute node `j`, then `v` sends the same message 1000 times between compute nodes `i` and `j`. Instead, GPS’s `LALP` optimization (explained in Section 5.2.6) stores partitioned adjacency lists for high-degree vertices across the compute nodes on which the neighbors reside. In our example, the 1000 messages are reduced to one.

Partitioning Experiments

By default GPS and Pregel distribute the vertices of a graph to the compute nodes randomly (typically round-robin). Using GPS we have explored the graph partitioning question: Can some algorithms perform better if we “intelligently” assign vertices to compute nodes before the computation begins? For example, how would the performance of the PageRank algorithm change if we partition the web-pages according to their domains, i.e., if all web-pages with the same domain names reside on the same compute node? What happens if we use the popular METIS [91] algorithm for partitioning, before computing PageRank, shortest-path, or other algorithms? Do we improve performance further by using GPS’s dynamic repartitioning scheme? We present extensive experiments demonstrating that the answer to all of these questions is yes, in certain settings. We will also see that maintaining workload balance across compute nodes, when using a sophisticated partitioning scheme, is nontrivial to achieve but crucial to achieving good performance.
Chapter Outline

- In Section 5.1 we describe GPS, our open-source Pregel-like distributed message passing system for large-scale graph algorithms. We present the architecture and the programming API.
- In Section 5.2 we study how different graph partitioning schemes affect the network and run-time performance of GPS on a variety of graphs and algorithms. We also describe our large adjacency-list partitioning feature (LALP) and report some experiments on it.
- In Section 5.3 we describe GPS’s dynamic repartitioning scheme. We repeat several of our experiments from Section 5.2 using dynamic repartitioning.
- In Section 5.4 we discuss several additional optimizations that reduce memory use and increase the overall performance of GPS.
- In Section 5.5 we cover related work.

Outline of Chapters on Large-scale Graph Processing

To set the stage for the remainder of the thesis, we provide an outline for the upcoming chapters that continue to focus on large-scale graph processing:

- Chapter 6 presents our work on implementing graph algorithms efficiently on Pregel-like systems. As we discuss in Chapter 6, although Pregel-like systems are more efficient than MapReduce, algorithms can still incur unnecessary inefficiencies such as executing very-large numbers of supersteps, or incurring high communication or computation costs. We describe several optimization techniques to address these inefficiencies and evaluate their effectiveness using GPS.
- Chapter 7 presents a debugger called Graft to help programmers debug their vertex.compute() and master.compute() functions. Graft is implemented for another open-source Pregel clone called Apache Giraph [8].
- Chapter 8 presents our work on programming distributed graph algorithms with a set of high-level graph primitives we call HelP. Although vertex.compute()
and `master.compute()` functions make it easier to implement graph algorithms than using MapReduce's `map()` and `reduce()` functions, they can still be too low-level when implementing some algorithms, yielding long and complex programs. To simplify expressing such graph algorithms, we present our HelP primitives that capture commonly appearing operations in large-scale graph computations. We describe the implementation of HelP on another distributed bulk synchronous graph system called GraphX [130], which is built on top of the Spark [134] data-processing system.

5.1 GPS System

GPS uses the distributed message-passing model of Pregel [87], which is based on bulk synchronous processing [123]. Broadly, the input is a directed graph, and each vertex of the graph maintains a user-defined `value`, and a `flag` indicating whether or not the vertex is active. Optionally, edges may also have values. The computation proceeds in iterations called `supersteps`, terminating when all vertices are inactive. Within a superstep \( i \), each active vertex \( u \) in parallel: (a) looks at the messages that were sent to \( u \) in superstep \( i - 1 \); (b) modifies its value; (c) sends messages to other vertices in the graph and optionally becomes inactive. Vertex \( u \) can send a message to any other vertex \( v \) in the graph as long as \( u \) knows \( v \)'s ID. A message sent in superstep \( i \) from vertex \( u \) to vertex \( v \) becomes available for \( v \) to use in superstep \( i + 1 \). The behavior of each vertex is encapsulated in a function `vertex.compute()`, which is executed exactly once in each superstep.

5.1.1 Overall Architecture

The architecture of GPS is shown in Figure 7.1. As in Pregel, there are two types of `processing elements (PEs)`: one master and \( k \) workers, \( W_0...W_{k-1} \). The master maintains a mapping of PE identifiers to physical compute nodes and workers use a copy of this mapping to communicate with each other and the master. PEs communicate using Apache MINA [12], a network application framework built on
java.nio, Java’s asynchronous network I/O package. GPS is implemented in Java. The compute nodes run HDFS (Hadoop Distributed File System) [59], which is used to store the input graph and the final vertex values at termination. We next explain how the input graph is partitioned across workers. The master and worker implementations are described in Section 5.1.3. Section 5.1.4 explains the API and provides examples.

5.1.2 Input Graph Partitioning Across Workers

The input graph $G$ is specified in HDFS files in a simple format: each line starts with the ID of a vertex $u$, followed by the IDs of $u$’s outgoing neighbors. The input file may optionally specify values for the vertices and edges. GPS assigns the vertices of $G$ to workers using the same simple round-robin scheme used by Pregel: vertex $u$ is assigned to worker $W_{(u \mod k)}$. When we experiment with more sophisticated partitioning schemes (Section 5.2), we run a preprocessing step to assign node IDs so that the round-robin distribution reflects our desired partitioning. GPS also supports optionally repartitioning the graph across workers during the
computation, described in Section 5.3.

5.1.3 Master and Worker Implementation

The master and worker PEs are again similar to Pregel [87]. The master coordinates the computation by instructing workers to: (a) start parsing input files; (b) start a new superstep; and (c) terminate computation. The master awaits notifications from all workers before instructing workers what to do next, and so serves as the centralized location where workers synchronize between supersteps. The master also calls a `master.compute()` function at the beginning of each superstep, described in Section 5.1.4.

Workers store vertex values, active flags, and message queues for the current and next supersteps. Each worker consists of three “thread groups,” as follows.

1. A *computation thread* loops through the vertices in the worker and executes `vertex.compute()` on each active vertex. It maintains an outgoing *message buffer* for all workers in the cluster, including itself. When a buffer is full it is either given to *MINA threads* for sending over the network, or passed directly to the local *message parser thread*.

2. *MINA threads* send and receive message buffers, as well as simple *coordination messages* between the master and the worker. When a message buffer is received, it is passed to the message parser thread.

3. A *message parser thread* parses incoming message buffers into separate messages and enqueues them into the receiving vertices’ message queues for the next superstep.

One advantage of this thread structure is that there are only two lightweight points of synchronization: when the computation thread passes a message buffer directly to the message parser thread, and when a MINA thread passes a message buffer to the message parser thread. Since message buffers are large (the default size is 100KB), these synchronizations happen infrequently.
class WCCVertex extends Vertex<IntWritable, IntWritable> {
    @Override
    void compute(Iterable<IntWritable> messages, int superstepNo) {
        if (superstepNo == 1) {
            setValue(new IntWritable(getId()));
            sendMessages(getNeighborIds(), getValue());
        } else {
            int minValue = getValue().value();
            for (IntWritable message : messages) {
                if (message.value() < minValue) {
                    minValue = message.value();
                }
            }
            if (minValue < getValue().value()) {
                setValue(new IntWritable(minValue));
                sendMessages(getNeighborIds(), getValue());
            } else {
                voteToHalt();
            }
        }
    }
}

Figure 5.2: Connected components in GPS.

5.1.4 API

Similar to Pregel, the programmer of GPS subclasses the `Vertex` class to define the vertex value, message, and optionally edge-value types. The programmer codes the vertex-centric logic of the computation by implementing the `vertex.compute()` function. Inside `vertex.compute()`, vertices can access their values, their incoming messages, and a map of `global objects`—our implementation of the `aggregators` of Pregel [87]. Global objects are used for coordination, data sharing, and statistics aggregation. At the beginning of each superstep, each worker has the same copy of the map of global objects. During a superstep, vertices can update objects in their worker’s local map, which are merged at the master at the end of the superstep, using a user-specified merge function. When ready, a vertex declares itself inactive by calling the `voteToHalt()` function in the API.

Algorithms whose computation can be expressed in a fully vertex-centric fashion are easily implemented using this API, as in our first example.
CHAPTER 5. GPS: A GRAPH PROCESSING SYSTEM

Figure 5.3: A simple k-means like graph clustering algorithm.

Example 5.1.1. WCC [74] is an algorithm to find the weakly connected components of an undirected graph: First, every vertex sets its value to its own ID. Then, in iterations, vertices set their values to the minimum value among their neighbors and their current value. When the vertex values converge, the value of every vertex \( v \) is the ID of the vertex that has the smallest ID in the component that \( v \) belongs to; these values identify the weakly connected components. WCC can be implemented easily using \texttt{vertex.compute()}, as shown in Figure 5.2.

A problem with this API (as presented so far) is that it is difficult to implement algorithms that include global as well as vertex-centric computations, as shown in the following example.

Example 5.1.2. Consider the simple k-means like graph clustering algorithm introduced in the beginning of this chapter and outlined in Figure 5.3. This algorithm has two vertex-centric steps, which we call “phases”:

1. Assigning each vertex to the closest “cluster center” (line 5 in Figure 5.3). This process is a simple extension of the algorithm from [87] to find shortest paths from a single source.

2. Counting the number of edges crossing clusters (line 6 in Figure 5.3). This computation requires two supersteps; it is shown in Figure 5.4.

Now consider lines 2 and 3 in Figure 5.3: checking the result of the latest clustering and terminating if the threshold has been met, or picking new cluster centers. With the API so far, we must put this logic inside \texttt{vertex.compute()} and designate a special “master” vertex to do it. Therefore, an entire extra superstep is spent at each iteration of the while loop (line 3 in Figure 5.3) to do this very short computation.
public class EdgeCountingVertex extends Vertex<IntWritable, IntWritable> {
    @Override
    public void compute(Iterable<IntWritable> messages, int superstepNo) {
        if (superstepNo == 1) {
            sendMessages(getNeighborIds(), getValue().value());
        } else if (superstepNo == 2) {
            for (IntWritable message : messages) {
                if (message.value() != getValue().value()) {
                    minValue = message.value();
                    updateGlobalObject("num-edges-crossing-clusters",
                                    new IntWritable(1));
                }
            }
            voteToHalt();
        }
    }
}

Figure 5.4: Counting the number of crossing edges with vertex.compute().

at one vertex, with others idle. Global objects cannot help us with this computation, since they only store values.

In GPS, we have addressed the shortcoming illustrated in Example 5.1.2 by extending the Pregel API to include an additional function, master.compute(). The programmer subclasses the Master class, and implements the master.compute() function, which gets called at the beginning of each superstep. The Master class has access to all of the merged global objects, and it can store its own global data that is not visible to the vertices. It can update the global objects map before it is broadcast to the workers.

Figure 5.5 shows an example master.compute(), used together with the vertex-centric computations already described (encapsulated in SimpleClusteringVertex, not shown) to implement the overall clustering algorithm of Figure 5.3. Lines 2 and 3 in Figure 5.3 are implemented in lines 24 and 25 of Figure 5.5. SimpleClustering-Master maintains a global object, phase, that coordinates the different phases of the algorithm. Using this global object, the master signals the vertices what phase of the algorithm they are currently in. By looking at the value of this object, vertices know what computation to do and what types of messages to send and receive. Thus, we are able to encapsulate vertex-centric computations in vertex.compute(), and coordinate them globally with master.compute().
public class SimpleClusteringMaster extends Master {
  @Override
  public void compute(int nextSuperstepNo) {
    if (nextSuperstepNo == 1) {
      pickKVerticesAndPutIntoGlobalObjects();
      getGlobalObjects().put("phase",
        new IntGlobalObject(CompStage.CLUSTER_FINDING_1));
    } else {
      int compStage = getGlobalObject("phase").value();
      switch(compStage) {
        case CompStage.CLUSTER_FINDING_1:
          getGlobalObjects().put("phase",
            new IntGlobalObject(CompStage.CLUSTER_FINDING_2));
          break;
        case CompStage.CLUSTER_FINDING_2:
          if (numActiveVertices() == 0) {
            getGlobalObjects().put("phase",
              new IntGlobalObject(CompStage.EDGE_COUNTING_1));}
          break;
        case CompStage.EDGE_COUNTING_1:
          getGlobalObjects().put("phase",
            new IntGlobalObject(CompStage.EDGE_COUNTING_2));
          break;
        case CompStage.EDGE_COUNTING_2:
          int numEdgesCrossing = getGlobalObject("num−edges−crossing").value();
          if (numEdgesCrossing > threshold) {
            pickKVerticesAndPutIntoGlobalObjects();
            getGlobalObjects().put("phase",
              new IntGlobalObject(CompStage.CLUSTER_FINDING_1));
          } else {
            terminateComputation(); }}}}

Figure 5.5: Clustering algorithm using master.compute().

We will use this general code pattern also in other algorithms in Chapter 6.
### Table 5.1: Data sets.

<table>
<thead>
<tr>
<th>Name</th>
<th>Vertices</th>
<th>Edges</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uk-2007-d</td>
<td>106M</td>
<td>3.7B</td>
<td>web graph of the .uk domain from 2007 (directed)</td>
</tr>
<tr>
<td>uk-2007-u</td>
<td>106M</td>
<td>6.6B</td>
<td>undirected version of uk-2007-d</td>
</tr>
<tr>
<td>sk-2005-d</td>
<td>51M</td>
<td>1.9B</td>
<td>web graph of the .sk domain from 2005 (directed)</td>
</tr>
<tr>
<td>sk-2005-u</td>
<td>51M</td>
<td>3.2B</td>
<td>undirected version of sk-2005-d</td>
</tr>
<tr>
<td>twitter-d</td>
<td>42M</td>
<td>1.5B</td>
<td>Twitter “who is followed by whom” network (directed)</td>
</tr>
<tr>
<td>uk-2005-d</td>
<td>39M</td>
<td>750M</td>
<td>web graph of the .uk domain from 2005 (directed)</td>
</tr>
<tr>
<td>uk-2005-u</td>
<td>39M</td>
<td>1.5B</td>
<td>undirected version of uk-2005-d</td>
</tr>
</tbody>
</table>

#### 5.2 Static Graph Partitioning

We next present our experiments on different static partitionings of the graph. In Sections 5.2.2-5.2.5 we show that by partitioning large graphs “intelligently” before computation begins, we can reduce total network I/O by up to 58.5x and runtime by up to 2.5x. The effects of partitioning depend on three factors: (1) the graph algorithm being executed; (2) the graph itself; and (3) the configuration of the worker tasks across compute nodes. We show experiments for a variety of settings demonstrating the importance of all three factors. We also explore partitioning the adjacency lists of high-degree vertices across workers; we report on those performance improvements in Section 5.2.6. We begin in Section 5.2.1 by explaining our experimental set-up.

#### 5.2.1 Experimental Setup

We describe our computing set-up, the graphs we use, the partitioning algorithms, and the graph algorithms used for our experiments.

We ran all our experiments on the Amazon EC2 cluster using large instances (4 virtual cores and 7.5GB of RAM) running Red Hat Linux OS. We repeated each experiment five times. The numeric results we present are the averages across all runs ignoring the initial data loading stage. Performance across multiple runs varied by only a very small margin.
The graphs we used in our experiments are specified in Table 7.2. We consider four different static partitionings of the graphs:

- **Random**: The default “mod” partitioning method described in Section 5.1, with vertex IDs ensured to be random.
- **METIS-default**: METIS [91] is publicly-available software that divides a graph into a specified number of partitions, trying to minimize the number of edges crossing the partitions. By default METIS balances the number of vertices in each partition. We set the \texttt{ufactor} parameter to 5, resulting in at most 0.5% imbalance in the number of vertices assigned to each partition [91].
- **METIS-balanced**: Using METIS’s multi-constraint partitioning feature [91], we generate partitions in which the number of vertices, outgoing edges, and incoming edges of partitions are balanced. We again allow 0.5% imbalance in each of these constraints. METIS-balanced takes more time to compute than METIS-default, although partitioning time itself is not a focus of our study.
- **Domain-based**: In this partitioning scheme for web graphs only, we locate all web pages from the same domain in the same partition, and partition the domains randomly across the workers.

Unless stated otherwise, we always generate the same number of partitions as we have workers.

Note that we are assuming an environment in which partitioning occurs once prior to executing the graph algorithms on GPS, while the graph algorithms may be run many times. Therefore we focus our experiments on the effect partitioning has on algorithms, not on the cost of partitioning itself.

We use four different graph algorithms in our experiments:

- **PageRank (PR) [25]
- Finding shortest paths from a single source (SSSP), as implemented in [87]
- The WCC [74] algorithm to find connected components
- **RW-n**, a pure random-walk simulation algorithm. Each vertex starts with an

\footnote{1These datasets were provided by “The Laboratory for Web Algorithmics” [81], using software packages WebGraph [22], LLP [21] and UbiCrawler [20].}
initial number of $n$ walkers. For each walker $i$ on a vertex $u$, $u$ randomly picks one of its neighbors, say $v$, to simulate $i$’s next step. For each neighbor $v$ of $u$, $u$ sends a message to $v$ indicating the number of walkers that walked from $u$ to $v$.

5.2.2 Performance Effects of Partitioning

Because of their bulk synchronous nature, the speed of Pregel-like systems is determined by the slowest worker to reach the synchronization points between supersteps. We can break down the workload of a worker into three parts:

1. **Computation**: Looping through vertices and executing `vertex.compute()`
2. **Networking**: Sending and receiving messages between workers
3. **Parsing and enqueuing messages**: In our implementation of GPS, where messages are stored as raw bytes, this involves byte array allocations and copying between byte arrays.

Although random partitioning generates well-balanced workloads across workers, almost all messages are sent across the network. We show that we can both maintain a balanced workload across workers and significantly reduce the network messages and overall run-time by partitioning the graph using our more sophisticated schemes.

With sophisticated partitioning of the graph we can obviously reduce network I/O, since we localize more edges within each worker compared to random partitioning. Our first set of experiments, presented in Section 5.2.3, quantifies the network I/O reduction for a variety of settings.

In Section 5.2.4 we present experiments measuring the run-time reduction due to sophisticated partitioning when running various algorithms in a variety of settings. We observe that partitioning schemes that maintain workload balance among workers perform better than schemes that do not, even if the latter have lower communication. In Section 5.2.5 we discuss how to fix the workload imbalance among workers when a partitioning scheme generates imbalanced partitions.
5.2.3 Network I/O

In our first set of experiments, we measured network I/O (network writes in GB across all workers) when running different graph algorithms under different partitioning schemes in a variety of settings. The reductions we report are relative to the performance of random partitioning. Overall, network I/O reductions varied between 1.8x to 2.2x for partitioning by domain, 13.3x and 36.3x for METIS-balanced, and 26.6x and 58.5x for METIS-default. We present two of our experiments in Figure 5.6. Figure 5.6a shows network I/O for different partitioning schemes when running PageRank on the sk-2005-d graph (recall Table 7.2), with 60 workers running on 60 compute nodes. Figure 5.6b shows network I/O for random and METIS-balanced partitioning when executing different algorithms on the uk-2007-u graph, also with 60 workers and 60 compute nodes. The graph plots per-superstep network I/O for PageRank and RW-800, and total network I/O for WCC and SSSP. We also experimented with different numbers of workers and compute nodes; we found that network I/O reduction percentages were similar. Of course, network I/O is not the only contributor to overall run-time, so the remainder of our experiments consider the effect of partitioning schemes and other parameters on run-time.
5.2.4 Run-time

In this section, we set out to test how much sophisticated partitioning improves overall run-time. We measure the run-time performance of four algorithms on different graphs, partitioning schemes, and worker and compute node configurations. We used between 15 and 60 nodes, and between one and four workers on each node.

Since the main benefit of sophisticated partitioning is reducing the number of messages sent over the network, we expect partitioning to improve run-time most in algorithms that generate a lot of messages and have low computational workloads. The computation and communication workloads of the graph algorithms we use can be characterized as:

- PageRank: short per-vertex computation, high communication
- WCC: short per-vertex computation, medium communication
- RW-800: long per-vertex computation (due to random number generation), medium communication
- SSSP: short per-vertex computation, low communication

A sample of our experimental results is shown in Figure 5.7. Figure 5.7a shows PageRank on the sk-2005-d graph on 60 compute nodes with 60 workers. In this experiment, improvements ranged between 1.1x for domain-based partitioning to 2.3x for METIS-balanced. In other experiments for PageRank, METIS-balanced consistently performed best, reducing run-time between 2.1x to 2.5x over random partitioning. Improvements for METIS-default varied from 1.4x to 2.4x and for domain-based partitioning from 1.1x to 1.7x.

Run-time reductions when executing other graph algorithms are less than PageRank, which is not surprising since PageRank has the highest communication to computation ratio of the algorithms we consider. Figure 5.7b shows four algorithms on the uk-2007-u graph using 30 workers running on 30 compute nodes. We compared the performance of random partitioning and METIS-balanced. As shown, METIS-balanced reduces the run-time by 2.2x when executing PageRank, and by 1.47x, 1.08x, and 1.06x for WCC, SSSP, and RW-800, respectively.
5.2.5 Workload Balance

In all of our experiments reported so far, METIS-default performed better than METIS-balanced in network I/O but worse in run-time. The reason for this counterintuitive performance is that METIS-default tends to create bottleneck workers that slow down the system. For all of the graph algorithms we are considering, messages are sent along the edges. Recall that METIS-default balances only the number of vertices in each partition and not the edges. As a result, some workers process a significantly higher number of messages than average. Figure 5.8 shows the number of messages processed by the slowest workers in each of the experiments of Figure 5.7a. The message counts for Random and METIS-balanced indicate fairly homogeneous workloads (perfect distribution would be about 63M messages per worker). But with METIS-default, one partition has more than twice the average load of other partitions, thus slowing down the entire system.

We discuss how to improve workload imbalance, and in turn improve the run-time benefits when using a sophisticated partitioning scheme that can generate imbalanced partitions. One approach is to generate more partitions than we have
workers, then assign multiple partitions to each worker, thus averaging the workloads from “heavy” and “light” partitions. For example, if we repeat the METIS-default experiment of Figures 5.7a and 5.8 but generate 240 partitions and assign each worker four partitions, the slowest worker processes 96M messages instead of the 132M in Figure 5.8, and run-time is reduced from 9.8 to 8.0 minutes. As a second experiment, we used the original 60 partitions generated by METIS-default, but with only 20 workers on 20 compute nodes, so three partitions assigned to each worker. The result of this experiment is shown in the METIS-default (3 partitions) bar in Figure 5.9. This set-up improves run-time by 2.2x over random partitioning, significantly better than the 1.4x METIS-default improvement in Figure 5.7a. We can obtain the same “averaging” effect by assigning one partition to each worker but running multiple workers per compute node. The METIS-default (3 workers) bar in Figure 5.9 shows the performance of using the 60 partitions generated by METIS-default from before, assigning one partition to each worker, and running three workers per compute node. We see that assigning one partition to each worker and three workers to each compute node performs similarly to assigning three partitions to each worker and one worker to each compute node.

Figure 5.8: Slowest worker, number of messages.
5.2.6 Large Adjacency-List Partitioning

GPS includes an optimization called LALP (large adjacency-list partitioning), in which adjacency lists of high-degree vertices are not stored in a single worker, but rather are partitioned across workers. This optimization can improve performance, but only for algorithms with two properties: (1) Vertices use their adjacency lists (outgoing neighbors) only to send messages and not for computation; (2) If a vertex sends a message, it sends the same message to all of its outgoing neighbors. For example, in PageRank each vertex sends its latest PageRank value to all of its neighbors, and that is the only time vertices access their adjacency lists. On the other hand, RW-n does not satisfy property 2: a message from vertex $u$ to its neighbor $v$ contains the number of walkers that move from $u$ to $v$ and is not necessarily the same as the message $u$ sends to its other neighbors.

Suppose a vertex $u$ is located in worker $W_i$ and let $N_j(u)$ be the outgoing neighbors of $u$ located in worker $W_j$. Suppose $|N_j(u)| = 10000$. During the execution of PageRank, $W_i$ sends 10000 copies of the same message to $W_j$ in each superstep, one for each vertex in $N_j(u)$. Instead, if $W_j$ stores $N_j(u)$, $W_i$ need send only a single message to $W_j$ for node $u$, and $W_j$ replicates this message 10000 times to the message queues of each vertex in $N_j(u)$. 

![Figure 5.9: Fixing workload imbalance of METIS-default.](image-url)
Many real-world graphs are known to have a skewed degree distribution, in which a small number of vertices’ adjacency lists contain a significant fraction of all the edges in the graph. For these graphs, \textit{LALP} can improve network traffic and run-time significantly. GPS programmers specify a parameter $\tau$ when using this optimization. If a vertex $u$ has more than $\tau$ outgoing neighbors, GPS partitions $u$’s adjacency list into $N_1(u), N_2(u), \ldots, N_k(u)$, and sends $N_j(u)$ to worker $W_j$ during the initial partitioning of the graph across workers. During execution, when $u$ sends a message to all its neighbors, GPS intercepts the message and sends a single a message to each worker $W_j$, with $W_j$ delivering the message to all vertices in $N_j(u)$.

To verify that LALP improves performance, we ran PageRank on the twitter-d graph, with different values of $\tau$, using 32 workers and 16 compute nodes. As we reduce $\tau$, GPS partitions more adjacency lists across all workers, and we expect network I/O to be reduced. On the other hand, as $\tau$ is reduced the map of $\langle u, N_j(u) \rangle$ pairs, which each worker $W_j$ maintains, grows, incurring some memory and computation overhead during message parsing. We expect there to be an optimal $\tau$ that achieves the best run-time performance. The results of our experiment are shown in Figure 5.10. Figure 5.10a shows that decreasing $\tau$ decreases network I/O. Figure 5.10b shows that for our experiment the optimal $\tau$ is around 60, and
achieves 1.41x run-time improvement over running without LALP.

5.3 Dynamic Repartitioning

To reduce the number of messages sent over the network, it might be helpful to reassign certain vertices to other workers dynamically during algorithm computation. There are three questions any dynamic repartitioning scheme must answer: (1) which vertices to reassign; (2) how and when to move the reassigned vertices to their new workers; (3) how to locate the reassigned vertices. Below, we explain our answers to these questions in GPS and discuss other possible options. We also present experiments measuring the network I/O and run-time performance of GPS when the graph is initially partitioned by one of our partitioning schemes from Section 5.2 then dynamically repartitioned during the computation.

5.3.1 Picking Vertices to Reassign

One option is to reassign vertex $u$ at worker $W_i$ to a new worker $W_j$ if $u$ sends/receives more messages to/from $W_j$ than to/from any other worker, and that number of messages is over some threshold. There are two issues with this approach. First, in order to observe incoming messages, we need to include the source worker in each message, which can increase the memory requirement significantly when the size of the actual messages are small. To avoid this memory requirement, GPS bases reassignment on sent messages only.

Second, using this basic reassignment technique, we observed that over multiple iterations, more and more vertices were reassigned to only a few workers, creating significant imbalance. Despite the network benefits, the “dense” workers significantly slowed down the system. To maintain balance, GPS exchanges vertices between workers. Each worker $W_i$ constructs a set $S_{ij}$ of vertices that potentially will be reassigned to $W_j$, for each $W_j$. Similarly $W_j$ constructs a set $S_{ji}$. Then $W_i$ and $W_j$ communicate the sizes of their sets and exchange exactly $\min(S_{ij}, S_{ji})$ vertices, guaranteeing that the number of vertices in each worker does not change through
CHAPTER 5. GPS: A GRAPH PROCESSING SYSTEM 145

dynamic repartitioning.

5.3.2 Moving Reassigned Vertices to New Workers

Once a dynamic partitioning scheme decides to reassign a vertex $u$ from $W_i$ to $W_j$ in superstep $x$, three pieces of data associated with $u$ must be sent to $W_j$: (a) $u$’s latest value; (b) $u$’s adjacency list; and (c) $u$’s messages for superstep $(x+1)$. One option is to insert a “vertex moving” stage between the end of superstep $x$ and beginning of superstep $x+1$, during which all vertex data is moved. GPS uses another option that combines vertex moving within the supersteps themselves: At the end of superstep $x$, workers exchange their set sizes as described in the previous subsection. Then, between the end of superstep $x$ and beginning of superstep $(x+1)$, the exact vertices to be exchanged are determined and the adjacency lists are relabeled, as described in the next subsection. Relabeling of the adjacency lists ensures that all messages that will be sent to $u$ in superstep $x+1$ are sent to $W_j$. However, $u$ is not sent to $W_j$ at this point. During the computation of superstep $(x+1)$, $W_i$ first calls $u$.compute() and then sends only $u$’s adjacency list and latest value to $W_j$. Thus, $u$’s messages for superstep $(x+1)$ are not sent to $W_j$, reducing the network overhead of dynamic repartitioning.

5.3.3 Locating Reassigned Vertices

When a vertex $u$ gets reassigned to a new worker, every worker in the cluster must obtain and store this information in order to deliver future messages to $u$. An obvious option is for each worker to store an in-memory map consisting of $<\text{vertex-id, new-worker-id}>$ pairs. Of course, over time, this map can potentially contain as many pairs as there are vertices in the original graph, causing a significant memory and computation bottleneck. In our experiments, up to 90% of vertices can eventually get reassigned. Thus, GPS instead uses an approach based on relabeling the IDs of reassigned vertices. Suppose $u$ has been reassigned to $W_j$. We give $u$ a new ID $u'$, such that $(u' \mod k) = j$. Since every pair $W_i$ and $W_j$ exchange the same number of vertices, vertex IDs can effectively be exchanged as
well. In addition, each worker must go through all adjacency lists in its partition and change each occurrence of $u$ to $u'$.

There are two considerations in this approach:

- If the application requires the original node IDs to be output at the end of the computation, this information must be retained with nodes whose IDs are modified, incurring some additional storage overhead.
- When a node $u$ is relabeled with a new ID, we modify its ID in the adjacency lists of all nodes with an edge to $u$. If the graph algorithm being executed involves messages not following edges (that is, messages from a node $u_1$ to a node $u_2$ where there is no edge from $u_1$ to $u_2$), then our relabeling scheme cannot be used. In most graph algorithms suitable for GPS, messages do follow edges.

### 5.3.4 Dynamic Repartitioning Experiments

Dynamic repartitioning is intended to improve network I/O and run-time by reducing the number of messages sent over the network. On the other hand, dynamic repartitioning also incurs network I/O overhead by sending vertex data between workers, and run-time overhead deciding which vertices to send and relabeling adjacency lists. It would not be surprising if, in the initial supersteps of an algorithm using dynamic repartitioning, the overhead exceeds the benefits. We expect that there is a crossover superstep $s$, such that dynamic repartitioning performs better than static partitioning only if the graph algorithm runs for more than $s$ supersteps. Obviously, $s$ could be different for network I/O versus run-time performance, and depends on the graph, graph algorithm, and initial partitioning.

In our first experiment, we ran PageRank on the $uk-2007-d$ graph for between 3 and 100 iterations, with random initial partitioning, and with and without dynamic repartitioning. We used 30 workers running on 30 compute nodes. In GPS, the master task turns dynamic repartitioning off when the number of vertices being exchanged is below a threshold, which is by default 0.1% of the total number of vertices in the graph. In our PageRank experiments, this typically occurred around
superstep 15-20. Our results are shown in Figure 5.11. The crossover superstep in this experiment was five iterations for network I/O and around 55 iterations for run-time. When running PageRank for long enough, dynamic repartitioning gives 2.0x performance improvement for network I/O and 1.13x for run-time.

We repeated our experiment, now initially partitioning the graph using METIS-balanced and domain-based, rather than random. When the initial partitioning is METIS-balanced, we do not see noticeable network I/O or run-time benefits from dynamic repartitioning. On the other hand, when we start with domain-based partitioning, the crossover iteration is 4 for network I/O and 36 for run-time. When running PageRank for long enough, dynamic repartitioning shows 2.2x and 1.2x performance improvement for network I/O and run-time, respectively.

In our setting, the run-time benefits of dynamic repartitioning seem to be modest at best. However, in settings where networking is slower, benefits from network I/O should yield significant run-time improvements as well.

5.4 Other System Optimizations

We describe two additional optimizations in GPS that reduce memory usage and increase overall performance.

- **Combining messages at the receiver worker:** Combiners [36] were introduced in the MapReduce framework to reduce the number of intermediate values sent from mappers to reducers, when the reducers use these values in
commutative and associative operations. Similarly, Pregel uses combinators at the sender and receiver workers to reduce both the number of messages sent between workers and the memory required to store messages in each worker. At the sender side, when multiple vertices from worker $W_i$ send messages to a vertex $v$ located in $W_j$, $W_i$ can combine some of these messages at certain intervals and send fewer messages to $W_j$, reducing network I/O. At the receiver side, when a message $m$ is received in $W_j$ for $v$, if the message list for $v$ is empty, $W_j$ can add $m$ to $v$’s message list. Otherwise, instead of appending $m$ to the list, $W_j$ can immediately combine its value with the current message in the list. Receiver-side combining reduces the total memory required to store messages for a particular superstep from $|E|$ to $|V|$—a significant reduction in most graphs where the number of edges is significantly higher than the number of vertices.

GPS supports only receiver-side combining. During our development of GPS, we implemented and experimented with sender-side combining. In order to combine messages at a sender worker $W_i$, $W_i$ needs to store an outgoing message list for each vertex $v$ that receives a message from $W_i$, which increases memory usage. Also, messages are buffered twice, once in the outgoing messages lists, and then in the message buffers for each worker, which slows down the rate at which buffers fill and are flushed. Overall, we did not observe significant performance improvements by combining messages at the sender and decided not to support this feature.

- **Single Vertex and Message objects**: GPS reduces the memory cost of allocating many Java objects by storing canonical objects. First, instead of storing the value and the adjacency list of each vertex $v$ inside a separate Vertex object, and calling vertex.compute() on each object, GPS workers use a single canonical Vertex object, with vertex values and adjacency lists stored in separate data structures. For each vertex $v$ in worker $W_i$, $W_i$ is configured so the canonical Vertex object has access to $v$’s value and adjacency list. $W_i$ then calls vertex.compute() on the canonical object. Similarly, GPS workers store a single canonical Message object. Incoming messages
are stored as raw bytes in the message queues, and a message is deserialized into the canonical `Message` object only when the canonical `Vertex` object iterates over it.

## 5.5 Related Work

We survey several different classes of systems designed to do large-scale graph computations.

- **Bulk synchronous message-passing systems**: Pregel [87] introduced the first bulk synchronous distributed message-passing system, which GPS has drawn from. Several other systems are based on Pregel, including Apache Giraph [8], Hama [10], and Phoebus [101]. Giraph is the most popular and advanced of these systems. Giraph programs run as Hadoop jobs without the reduce phase. Giraph leverages the task scheduling component of Hadoop clusters by running workers as special mappers, which communicate with each other to deliver messages between vertices and synchronize between supersteps. Since our introduction of GPS, Giraph has also adopted our `master.compute()` API extension and our optimization for using single vertex and message objects.

- **Hadoop-based systems**: Many graph algorithms, e.g., computing PageRank or finding connected components, are iterative computations that terminate when a vertex-centric convergence criterion is met. Because MapReduce is a two-phased computational model, iterative graph algorithms cannot be expressed in Hadoop easily. One approach to solve this limitation has been to build systems on top of Hadoop, in which the programmer can express a graph algorithm as a series of MapReduce jobs, each one corresponding to one iteration of the algorithm. Pegasus [74], Mahout [11], HaLoop [27], iMapReduce [135], Surfer [32] and Twister [40] are examples of these systems. Since these systems execute each iteration of graph algorithms as separate MapReduce jobs, they suffer from the same inefficiencies of MapReduce we described in
the beginning of this chapter, which do not exist in bulk synchronous message-passing systems: (1) The input graph, which does not change from iteration to iteration, may not stay in RAM, and is sent from mappers to reducers in each iteration. (2) They lack a graph-specific API to express graph algorithms.

- **Asynchronous systems**: GPS supports only bulk synchronous graph processing. PowerGraph [50] and Signal-Collect [119] support asynchronous vertex-centric graph processing. An advantage of asynchronous computation over bulk synchronous computation is that fast workers do not have to wait for slower workers. However, programming in the asynchronous model can be harder than synchronous models, as programmers have to reason about the non-deterministic order of vertex-centric function calls.

- **Spark and GraphX**: Spark [134] is a general cluster computing system, whose API is designed to express generic iterative computations. The Spark API consists of parallelizable data primitives such as join, filter, and groupBy. As a result, programming graph algorithms directly on Spark requires significantly more coding effort than on GPS. GraphX is a distributed graph engine built on top of Spark. GraphX supports a set of low-level parallelizable core graph primitives that can simulate messaging between vertices, and modify the graph and vertex values. We will review these primitives in detail in Chapter 8. Unlike the vertex-centric APIs of Pregel, PowerGraph, or Signal-Collect, GraphX algorithms can be written as a sequence of operations that modify an input graph. GraphX also contains two libraries on top of the core GraphX primitives that implement Pregel’s and PowerGraph’s vertex-centric APIs. As we will explain in Chapter 8 although we find the core GraphX API too low-level for expressing some graph algorithms, overall we welcome the direction of expressing graph algorithms as a sequence of graph operations.

- **Message Passing Interface (MPI)**: MPI is a standard interface for building a broad range of message passing programs. There are several open-source implementations of MPI [99, 94]. MPI-based libraries, e.g., [28, 57, 86], can also be used to implement parallel message-passing graph algorithms. These libraries can be very efficient, but they require users to reason about low-level
synchronization, scheduling, and communication primitives in their code in order to realize the efficiency.

- **Other systems**: Trinity [110] is both a graph database and a graph computation platform based on a distributed key-value store, but it is not open-source. Grail [42] is a vertex-centric graph system whose API is similar to a limited version of Pregel’s and GPS’s API, without global objects or `master.compute()`. Grail is built on top of Microsoft SQL Server 2014, an industry row-oriented relational database management system. Grail compiles a vertex-centric program into an equivalent SQL program, which gets executed by the underlying Microsoft SQL system. Vertexica [71] is similar to Grail and compiles a user provided vertex-centric program to an equivalent SQL program, which gets executed on top of a closed-source industry column store. Both Vertexica and Grail make it easy to express algorithms that are solely composed of vertex-centric operations. However, for more complex algorithms, say that require aggregating statistics over the graph, programmers need to mix vertex-centric code with SQL code, which we believe is difficult and not intuitive.

In addition to presenting GPS, we studied the effects of different graph partitioning schemes on the performance of GPS when running different graph algorithms. We also studied the effects of GPS’s dynamic repartitioning scheme on performance. There are previous studies on the performance effects of different partitionings of graphs on other systems. Reference [65] shows that by partitioning Resource Description Framework [105] (RDF) data with METIS and then “intelligently” replicating certain tuples, SPARQL [115] query run-times can be improved significantly over random partitioning. We study the effects of partitioning under batch algorithms, whereas SPARQL queries consist of short path-finding workloads. Reference [118] develops a heuristic to partition the graph across machines during the initial loading phase. They study the reduction in the number of edges crossing machines and run-time improvements on Spark when running PageRank. They do not study the effects of other static or dynamic partitioning schemes. Reference [29] also experiments with the run-time effects of different ways of repartitioning a sparse matrix.
representation of graphs when computing PageRank. *Sedge* \[131\] is a graph query engine based on a simple Pregel implementation. In Sedge, multiple replicas of the graph are partitioned differently and stored on different groups of workers; queries are routed to the group that will result in minimum communication.
Chapter 6

Optimizing Graph Algorithms For Pregel-like Systems

In this chapter we study the problem of implementing graph algorithms efficiently on Pregel-like systems, which can be surprisingly challenging and require careful optimizations. As we show in this chapter, standard graph algorithms in this setting can incur unnecessary inefficiencies such as slow convergence or high communication or computation cost, typically due to structural properties of the input graphs such as large diameters or skew in component sizes. We base our discussion on thorough implementations of several fundamental graph algorithms. We identify a variety of inefficiencies that arise when executing these algorithms on Pregel-like systems, and we describe optimization techniques, some of which are applicable across multiple algorithms, to address them. For reference, Tables 6.1 and 6.2 list the algorithms and optimization techniques covered in this chapter. In the remainder of this introductory material we provide a summary of our approach and results.

Costs of Computation

Broadly, there are four different costs involved when executing an algorithm on Pregel-like systems: (a) communication, i.e., number of messages transmitted between compute nodes; (b) number of supersteps; (c) memory, i.e., size of the local state stored per vertex; and (d) computation performed by vertices
## Table 6.1: Algorithms.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Algorithms</th>
<th>Inefficiency</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly Connected Components (SCC)</td>
<td>The Coloring algorithm from [100]</td>
<td>SCC, GC, MSF, MWM</td>
<td>Large-diameter graphs, skew in component sizes</td>
<td>6.2.1</td>
</tr>
<tr>
<td>Minimum Spanning Forest (MSF)</td>
<td>Parallel version of Boruvka’s algorithm [33]</td>
<td></td>
<td></td>
<td>6.2.2</td>
</tr>
<tr>
<td>Graph Coloring (GC)</td>
<td>Greedy algorithm based on Luby’s parallel maximal independent set algorithm [85]</td>
<td></td>
<td></td>
<td>6.2.3</td>
</tr>
<tr>
<td>Approximate Maximum Weight Matching (MWM)</td>
<td>1/2-approximation algorithm for general graphs [103]</td>
<td></td>
<td></td>
<td>6.2.4</td>
</tr>
<tr>
<td>Weakly Connected Components (WCC)</td>
<td>Algorithm from [74]</td>
<td></td>
<td></td>
<td>6.2.5</td>
</tr>
</tbody>
</table>

## Table 6.2: Optimization Techniques.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Algorithms</th>
<th>Inefficiency</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finishing Computations Serially (FCS)</td>
<td>Performs serial version of the algorithm, or a phase of the algorithm, inside master.compute()</td>
<td>SCC, GC, MSF, MWM, Backward-Traversal phase of SCC</td>
<td>Large-diameter graphs, skew in component sizes</td>
<td>6.3</td>
</tr>
<tr>
<td>Storing Edges At Subvertices (SEAS)</td>
<td>Stores the edges of supervertices in a distributed fashion among its subvertices</td>
<td>MSF</td>
<td>High-cost single phase</td>
<td>6.4</td>
</tr>
<tr>
<td>Edge Cleaning On Demand (ECOD)</td>
<td>Edges are cleaned only when they are used as part of the computation</td>
<td>MWM, MSF</td>
<td>High-cost single phase</td>
<td>6.5</td>
</tr>
<tr>
<td>Single Pivot (SP)</td>
<td>Detects giant component efficiently by starting the computation from a single vertex</td>
<td>SCC, WCC</td>
<td>Skew in component sizes</td>
<td>6.6</td>
</tr>
</tbody>
</table>
in each superstep. The optimization techniques we offer in this chapter focus on reducing the first two costs: communication and number of supersteps. For the algorithms we consider, memory size and local computation are not dominant in overall run-time, nor do they appear to have room for significant improvement.

Sometimes, system-level optimizations—those that do not require any changes to the graph algorithm itself—can be used to reduce communication and memory. For example, the LALP optimization of GPS that we discussed in Chapter 5 is a graph partitioning technique that can reduce the communication cost significantly when executing some algorithms on input graphs with skewed degree distributions. Our work focuses on optimizations that are algorithmic and do not appear to have any system-level equivalents. Whether system-level alternatives to some of our optimization techniques can be designed is an interesting question for future work.

**Optimization Techniques**

We propose four optimization techniques in this chapter:

- **Finishing Computations Serially (FCS)**: Some algorithms or phases of algorithms may converge very slowly (i.e., execute many supersteps), while working on a tiny fraction of the input graph. FCS monitors the size of the “active” graph on which the computation is executing. If the active graph becomes small enough, FCS sends it to the master, which performs the end of the computation serially inside `master.compute()`, then sends the results back to the workers. In the algorithms we consider, slow convergence typically stems from structural properties of the graph, such as skew in components sizes and small-size maximal independent sets. From our experiments, we find that FCS can eliminate between 20% to 60% of total superstep executions (up to 16713 supersteps) when applied to several of our algorithms and their phases on large graphs.

- **Storing Edges At Subvertices (SEAS)**: SEAS is an optimization we apply primarily to our MSF algorithm (Table 6.1), in which groups of vertices (called subvertices) are merged to form supervertices. In the natural Pregel
implementation of supervertex formation, subvertices send their adjacency lists to the supervertex, then become inactive. SEAS instead retains the adjacency lists of subvertices and keeps them active. SEAS effectively avoids the cost of sending adjacency lists to the supervertex, at the expense of incurring some communication cost between subvertices and supervertices, with overall run-time benefits.

- **Edge Cleaning On Demand (ECOD):** “Edge cleaning” is a common operation in graph algorithms, in which vertices delete some neighbors from their adjacency lists based on some or all of their neighbors’ values. The natural Pregel implementation of edge cleaning can be expensive: vertices send each other their values in one superstep, then clean their adjacency lists in another superstep. ECOD avoids the edge cleaning phase completely, removing “stale” edges only when they are discovered later in the computation.

- **Single Pivot (SP):** The SP optimization technique was proposed originally in [104] for the WCC algorithm (Table 6.1). SP avoids unnecessary communication when detecting very large components in graphs with skewed component sizes. We show that SP is also applicable to the SCC algorithm. While SP shows only modest improvements for WCC in the setting of [104], it shows good improvements for both SCC and WCC in our setting.

**Chapter Outline**

- **Section 6.1** describes our experimental setup and datasets.
- **Section 6.2** describes the algorithms covered in this chapter. The Pregel implementations of some of our algorithms, to the best of our knowledge, have not been described before, and were quite challenging.
- **Section 6.3** covers our FCS optimization technique, which can eliminate many superstep executions when algorithms converge slowly. FCS is based on performing serial computation inside `master.compute()` on a small fraction of the input graph.
- **Section 6.4** covers our SEAS optimization for Boruvka’s minimum spanning
forest algorithm.

• Section 6.5 covers our ECOD optimization technique, which can be used to eliminate the edge cleaning phases in some algorithms.

• Section 6.6 reviews the SP optimization from [104] and discusses how skew in component sizes can yield unnecessarily high communication cost in the strongly and weakly connected components algorithms we study.

• Sections 6.7 discusses related work.

Sections 6.3–6.6 all include extensive experiments demonstrating the benefits of our optimization techniques. All of our algorithms and optimizations are fully implemented on GPS and have been made available for public download [54].

### 6.1 Experimental Setup

The directed (d) and undirected (u) graphs we used in our experiments are specified in Table 6.3. We used three different clusters for our experiments with varying number of compute nodes (m) and workers (w): (1) Large-EC2(m, w): Amazon EC2’s large instance machines (four virtual cores and 7.5GB of RAM); (2) Medium-EC2(m, w): Amazon EC2’s medium instance machines (two virtual cores and 3.75GB of RAM); (3) Local(m, w): our local cluster’s machines (32 cores and 64GB of RAM). The machines in all our setups were running Red Hat Linux OS.

Table 6.3: Graph datasets.

<table>
<thead>
<tr>
<th>Name</th>
<th>Vertices</th>
<th>Edges</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sk-2005</td>
<td>51M</td>
<td>1.9B (d), 3.5B (u)</td>
<td>Web graph of the .sk domain from 2005</td>
</tr>
<tr>
<td>twitter</td>
<td>42M</td>
<td>1.5B (d), 2.7B (u)</td>
<td>Twitter “who is followed by who” network</td>
</tr>
<tr>
<td>friendster</td>
<td>125M</td>
<td>1.8B (d), 3.1B (u)</td>
<td>Friendster social network</td>
</tr>
<tr>
<td>uk-2005</td>
<td>39M</td>
<td>750M (d), 1.4B (u)</td>
<td>Web graph of the .uk domain from 2005</td>
</tr>
<tr>
<td>random-2.5B</td>
<td>500M</td>
<td>2.5B (d), 4.3B (u)</td>
<td>Graph with uniformly random edges</td>
</tr>
</tbody>
</table>

1The Web and the Twitter graphs were provided by “The Laboratory for Web Algorithmics” [81], using software packages WebGraph [22], LLP [21], and UbiCrawler [20]. The original Friendster social graph is undirected; we assign a random direction to each edge. For algorithms that take as input weighted graphs, we assign each edge a weight between 0 and 1 uniformly at random.
We ran our experiments with fault-tolerance off and we ignore the initial data loading stage in our measurements.

We note that the run-time results we report may vary on a Pregel-like system other than GPS. However, except for our randomized SP optimization, the number of supersteps our algorithms and optimizations take will be exactly the same across systems. The relative network I/O effects of our optimizations will also be similar in all Pregel-like systems if the graph is partitioned randomly across workers as we do in this chapter. The differences in the actual network I/O across systems would be due to the differences in the encoding and serialization of the IDs and messages between vertices.

We also note that we have not repeated our experiments in every cluster and every possible compute node and worker configuration. However, the experiments we conducted suggest that the relative performance benefits of our optimizations do not vary significantly across configurations.

6.2 Algorithms

We next describe the five graph algorithms we study in this chapter (Table 6.1). Most of our algorithms consist of multiple computational steps. For example, an algorithm might prune the graph in one phase and traverse it in another. As we did in Chapter 5, we refer to the different steps of algorithms as “phases” (not to be confused with the multiple supersteps that occur within each phase). In our implementations of these algorithms, we use the code pattern that we presented in Chapter 5 for implementing algorithms that consist of multiple phases. Briefly, a phase global object stores the current phase of the algorithm that is executing. The master.compute() function contains the logic of which phase should be executed in the next superstep, depending on the current phase and possibly other global objects. As a simple example, in the SCC algorithm (Section 6.2.1), when the current phase is “Forward-Traversal-Rest,” the code fragment in Figure 6.1 is used by the master class to determine whether to switch the phase to “Backward-Traversal-Start.”
CHAPTER 6. ALGORITHMIC OPTIMIZATIONS

159

1 int numVerticesWithUpdatedColorIDs = getGlobalObject("num-updated").value();
2 if (numVerticesWithUpdatedColorIDs > 0) {
3    setGlobalObject("phase", FW_TRAVERSAL_REST.value());
4 } else { setGlobalObject("phase", BW_TRAVERSAL_START.value()); }

Figure 6.1: SCCMaster’s code for switching phases.

1 public class SCCVertex extends Vertex<SCCVertexValue, SCCMessage> {
2     public void compute(Iterable<SCCMessage> messages) {
3         ComputationPhase phase = getPhase(getGlobalObject("phase").value());
4         switch(phase) {
5             case TRANSPOSE_GRAPH_FORMATION_1: doTransposeGraphFormation1();
6             case TRANSPOSE_GRAPH_FORMATION_2: doTransposeGraphFormation2(messages);
7             case TRIMMING: doTrimming();
8             case FW_TRAVERSAL_START: doFwTraversalStart();
9             case FW_TRAVERSAL_REST: doFwTraversalRest(messages);
10             case BW_TRAVERSAL_START: doBwTraversalStart();
11             case BW_TRAVERSAL_REST: doBwTraversalRest(messages);}
4 }}

Figure 6.2: Skeleton code for SCCVertex.compute().

For each phase, the Vertex class contains one subroutine implementing the vertex-centric logic of the phase. The vertex.compute() function calls the appropriate subroutine according to the value of the phase global object. As an example, Figure 6.2 shows the skeleton of the vertex.compute() function for SCC.

6.2.1 Strongly Connected Components (SCC)

We implement the parallel Coloring algorithm from [100] for finding strongly connected components. Figure 6.3 shows the original algorithm, with four phases:

1. Transpose Graph Formation: The algorithm first constructs the transpose of the input graph $G$ (line 2).
2. Trimming: In the Trimming phase (line 4), the algorithm identifies trivial SCCs: vertices with only incoming or only outgoing edges (or neither).
1 Coloring(G(V,E))
2 \( G^T = \text{constructTransposeGraph}(G) \)
3 while \( V \neq \emptyset \):
4 \hspace{1cm} \text{Trim } G \text{ and } G^T
5 // colors vertices into disjoint color sets
6 \hspace{1cm} \text{MaxForwardReachable}(G, \text{start from every } v \in V)
7 \hspace{1cm} \text{foreach } p \in P \text{ in parallel:}
8 \hspace{2cm} \text{if color}(p) == p:
9 \hspace{3cm} \text{let } S_p \text{ be the vertices colored } p
10 \hspace{3cm} \text{SCC}_p = S_p \cap \text{BackwardReachable}(G^T, p)
11 \hspace{3cm} \text{remove } \text{SCC}_p \text{ from } G \text{ and } G^T

Figure 6.3: Original Coloring algorithm for computing SCCs [100].

3. **Forward-Traversal:** In the Forward-Traversal phase, which is encapsulated in the `MaxForwardReachable()` subroutine call on line 6, the algorithm traverses \( G \) in parallel from each vertex. During the traversals, each vertex \( v \) is colored by the maximum ID of the vertex that can reach \( v \) (possibly \( v \) itself). The Forward-Traversal phase has two properties: (1) \( G \) is partitioned into disjoint sets of vertices according to their colors, called *color sets*. (2) If \( S_i \) is the color set containing vertices colored \( i \), then \( \text{SCC}_i \), the SCC that vertex \( i \) belongs to, is entirely contained in \( S_i \).

4. **Backward-Traversal:** In the Backward-Traversal phase (lines 7–11), the algorithm detects one SCC for each color set \( S_i \), by doing a traversal from vertex \( i \) in the transpose of \( G \) and limiting the traversal to only the vertices in \( S_i \). The detected SCCs are then removed from the graph.

The algorithm repeats the Trimming, Forward-Traversal, and Backward-Traversal phases, each time detecting and removing from the graph one or more SCCs. It terminates when there are no vertices left in the graph.

In our distributed implementation of the algorithm, vertices contain two fields: (1) `colorID` stores the color of a vertex \( v \) in the Forward-Traversal phase and identifies \( v \)'s SCC at the end of the computation, i.e., vertices with the same `colorID` after termination are in the same SCC. (2) `transposeNeighbors` stores the IDs of \( v \)'s neighbors in the transpose of the input graph. The four phases...
1. **Transpose Graph Formation**: Requires two supersteps. In the first superstep, each vertex sends a message with its ID to all its outgoing neighbors, which in the second superstep are stored in `transposeNeighbors`.

2. **Trimming**: Takes one superstep. Every vertex with only incoming or only outgoing edges (or neither) sets its `colorID` to its own ID and becomes inactive. Messages subsequently sent to the vertex are ignored.

3. **Forward-Traversal**: Figure 6.4 shows the subroutines implementing the Forward-Traversal phase in the `Vertex` class. There are two subphases: `Start` and `Rest`. In the Start phase, each vertex sets its `colorID` to its own ID and propagates its ID to its outgoing neighbors. In the Rest phase, vertices update their own `colorIDs` with the maximum `colorID` they have seen, and propagate their `colorIDs`, if updated, until the `colorIDs` converge. The Master sets the phase global object to Backward-Traversal when the `colorIDs` converge.

4. **Backward-Traversal**: We again break the phase into `Start` and `Rest`. In Start, every vertex whose ID equals its `colorID` propagates its ID to the vertices in `transposeNeighbors`. In each of the Rest phase supersteps, each vertex receiving a message that matches its `colorID`: (1) propagates its `colorID` in the transpose graph; (2) sets itself inactive; (3) sets the `converged-vertex-exists` global object (false at the start of the superstep) to true. Messages subsequently sent to the vertex are ignored. The Master sets the phase global object back to Trimming when `converged-vertex-exists` remains false at the end of a superstep.

### 6.2.2 Minimum Spanning Forest (MSF)

We implement the parallel version of Boruvka’s MSF algorithm from [33], referring the reader to [95, 33] for details. Figure 6.5 shows the original algorithm. The algorithm repeats four phases of computation in iterations, each time adding a set of edges to the MSF $S$ it constructs, and removing some vertices from the graph until there are no vertices left.
public void doFwTraversalStart() {
    value().colorID = getId();
    sendMessages(getOutgoingNeighbors(), new SCCMessage(getId()));
}

public void doFwTraversalRest(Iterable<SCCMessage> messages) {
    int maxColorID = findMaxColorID(messages);
    if (maxColorID > value().colorID) {
        sendMessages(getOutgoingNeighbors(), new SCCMessage(value().colorID));
        updateGlobalObject("updated-vertex-exists", true);
    }
}

Figure 6.4: SCCVertex subroutines for the Forward-Traversal phase.

Boruvkas_MSF(G(V, E))
S = Ø
while V ≠ Ø:
    foreach v ∈ V in parallel:
        e = PickMinWeightEdge(v)
        S = S ∪ e
        v.sv = FindSupervertexOfConjoinedTree()
        removeNeighborsInSameConjoinedTree()
        relabelIDsOfNeighborsInDifferentConjoinedTrees()
    foreach sv r ∈ V in parallel:
        r.adjacencyList = mergeAdjacencyListsOfConjoinedTree();
    remove all subvertices from V

Figure 6.5: Parallel version of Boruvka’s MSF algorithm [33].

1. **Min-Edge-Picking**: In parallel, each vertex v picks its current minimum-weight edge (v, u). Ties are broken by picking the edge with minimum destination ID. Each picked edge (v, u) is added to S. As proven in [33], the vertices and their picked edges form disjoint subgraphs T₁, T₂, ..., Tₖ, each of which is a conjoined-tree: two trees, the roots of which are joined by a cycle. Figure 8.3 shows an example of a conjoined-tree. In the figure, vertex 0 picks vertex 1 as its minimum-weight edge, vertex 1 picks vertex 3, vertex 2 picks vertex 3, etc. We refer to the vertex with the smaller ID in the cycle of Tᵢ as the supervertex of Tᵢ, for example vertex 5 in Figure 8.3. All other vertices in Tᵢ are called subvertices. The following phases merge all of the subvertices of each Tᵢ into the supervertex of Tᵢ.
2. **Supervertex-Finding**: Each vertex finds the supervertex of the conjoined-tree it belongs to (line 7).

3. **Edge-Cleaning-and-Relabeling**: Each vertex $v$ performs one of two operations for each of its neighbors $u$: $v$ removes $u$ if $v$ and $u$ are in the same conjoined-tree (line 8), or relabels $u$ with the supervertex of the conjoined-tree $u$ belongs to, possibly $u$ itself (line 9). For correctness of the final output, the algorithm stores the original source and destination IDs on each edge, which remain unchanged throughout the computation.

4. **Supervertex-Formation**: The algorithm merges the relabeled adjacency lists of each $T_i$ at its supervertex, keeping the minimum-weight edges for duplicates (lines 10–12). All subvertices as well as supervertices with no edges are removed from the graph.

   In our distributed implementation, each vertex stores a type and a pointer field, which are used in the Supervertex-Finding phase (explained below). Each vertex $v$ also has a pickedEdgeSrcID and a pickedEdgeDstID field (initialized to null), which respectively store the original source and destination IDs of the last edge $v$ picks, i.e., the edge $v$ picks in the last Min-Edge-Picking phase it participates in. At the end of the computation, these fields identify the edges that are in $S$.

1. **Min-Edge-Picking**: Each vertex picks its minimum-weight edge and writes its pickedEdgeSrcID and pickedEdgeDstID fields.
2. **Supervertex-Finding**: For finding supervertices we implement the *Simple Pointer-Jumping Algorithm* from \[33\]. There are two subphases: *Question* and *Answer*. We explain the subphases using the example conjoined-tree from Figure 8.3. Initially each vertex \(v\) sets its type to *Unknown* and pointer to the neighbor \(v\) picked in Min-Edge-Picking. In the first Question phase every vertex \(v\) sends a *question message* to \(v\).pointer. In our example, vertex 0 sends a message to 1, and 5 and 6 send messages to each other. In the first Answer phase, 5 and 6 see that they’ve sent each other messages and discover that they are part of the cycle of the conjoined-tree. 5 sets its type to *Supervertex* and 6 to *PointsAtSupervertex*. Every other vertex sets its type to *PointsAtSubvertex*. In addition, if a vertex \(v\) receives a question, it replies with an answer that contains the ID of \(v\).pointer and whether \(v\).pointer is the supervertex. For example, 1 sends 0 the message <3, *isSupervertex*:false>, while 6 sends 7 and 8 the message <5, *isSupervertex*:true>.

From then on, we execute the Question and Answer phases in iterations until every vertex points to the supervertex of its conjoined-tree. Let \(d_i\) be the longest distance of any leaf vertex in the conjoined-tree \(T_i\) to its supervertex, and let \(d_{\text{max}}\) be the maximum over all \(d_i\). Supervertex-Finding takes \(\log(d_{\text{max}})\) supersteps.

3. **Edge-Cleaning-and-Relabeling**: Takes two supersteps. First, each vertex \(v\) sends its ID and supervertex ID to all of its neighbors. The supervertex ID is effectively the new ID for \(v\). In the second superstep, vertex \(v\) for each of its edges \(e = (v, u)\) either deletes \(e\) if \(u\) has the same supervertex ID, or relabels \(e\) to point to \(u\)’s new ID.

4. **Supervertex-Formation**: Takes two supersteps. First, every subvertex sends its edges to its supervertex and becomes inactive. Then, each supervertex merges and stores these edges, keeping the minimum-weight for duplicates.
6.2.3 Graph Coloring (GC)

Graph coloring is the problem of assigning a color to each vertex of an undirected graph such that no two adjacent vertices have the same color. We implement the greedy algorithm from [47]. The algorithm iteratively finds a maximal independent set (MIS) of vertices, i.e., a maximal set of vertices such that no pair of vertices are adjacent. The algorithm assigns the vertices in each MIS a new color, then removes them from the graph, until there are no vertices left in the graph.

For finding an MIS we use Luby’s classic parallel algorithm [85]. The algorithm maintains three sets of vertices:

- $S$: The MIS being constructed. Starts empty and grows in iterations.
- $NotInS$: Vertices that have at least one edge to a vertex in $S$ and as a result cannot be in $S$.
- $Unknown$: Vertices that do not have an edge to any vertex in $S$ but are not yet in $S$.

In each iteration of MIS, each $Unknown$ vertex $v$ is first tentatively added to $S$ with $\frac{1}{2 \times \text{degree}(v)}$ probability. Suppose $v$ and some of its neighbors, $u_1, u_2, ..., u_k$, are added to $S$. Then, $v$ is kept in $S$ only if its ID is less than all IDs of $u_1, u_2, ..., u_k$. Otherwise, $v$ is put back to $Unknown$. By putting only the minimum ID vertex into $S$, the algorithm guarantees that two neighbor vertices are not added to $S$, i.e., vertices in $S$ are independent. If a vertex $w$ is in $Unknown$ and has an edge to any of the vertices that were added to $S$, then $w$ is put into $NotInS$. Finally, vertices that remain in $Unknown$ decrement their degree counts by the number of their neighbors that were put into $NotInS$. The iterations continue until there are no $Unknown$ vertices.

In our distributed implementation of GC, vertices have type and degree fields that are used in the MIS construction, and a color field to store the color of the vertex (initialized to null). There are six phases:

- **MIS-Degree-Initialization**: Executed once for each MIS the algorithm constructs. Takes two supersteps. First, each vertex remaining in the graph sets its type to $Unknown$ and sends an empty message to all its neighbors. In the
second superstep, vertices set their degree fields to the number of messages they receive.

- **Selection:** Takes one superstep. Each vertex \( v \) sets its type to TentativelyInS with \( \frac{1}{2 \times \text{degree}(v)} \) probability, then notifies its neighbors with a message containing its ID.

- **Conflict-Resolution:** Takes one superstep. Each vertex \( v \) that is tentatively in \( S \) inspects the IDs of its messages. If \( v \) has the minimum ID among its messages, it sets its type to InS, then sends an empty “neighbor-in-set message” to its neighbors. Otherwise, \( v \) sets its type back to Unknown.

- **NotInS-Discovery-and-Degree-Adjusting-1:** If \( v \) receives a neighbor-in-set message, \( v \) sets its type to NotInS, becomes inactive, and sends an empty “decrement degree” message to its neighbors.

- **Degree-Adjusting-2:** Every vertex \( v \) that is of type Unknown decreases its degree by the number of messages it receives. If there are remaining Unknown vertices, the master sets the phase back to Selection. Otherwise, the MIS construction is complete and the master sets the phase to Color-Assignment.

- **Color-Assignment:** Each vertex that is of type InS sets its color field to a new color, which is broadcast by the master inside a global object, and becomes inactive. Vertices of type NotInS set their types back to Unknown.

### 6.2.4 Approximate Maximum Weight Matching (MWM)

A maximum weight matching (MWM) of a weighted undirected graph \( G \) is a set of edges \( M \) such that each vertex \( v \) is part of at most one edge in \( M \) (i.e., \( M \) is a matching) and there is no other matching that has higher weight than \( M \). We implement the approximate MWM algorithm from [103]. In each iteration of MWM, vertices select their MWM neighbors. If \( u \) and \( v \) select each other, the edge \((u, v)\) is added to \( M \), and \( u \) and \( v \) (along with all edges pointing to them) are removed from the graph. The iterations continue until there are no vertices left in the graph. A proof that the algorithm computes a \( 1/2 \)-approximation to the maximum matching in the graph can be found in [103]. In our implementation
of the algorithm, we store a \texttt{pickedEdge} field per vertex, which is initially null. There are three phases:

- **Max-Weight-Edge-Picking**: Every vertex $v$ picks its maximum weight neighbor $u$ (ties are broken by picking the neighbor with minimum ID), stores it tentatively in its \texttt{pickedEdge} field, and sends a message to $u$ containing $v$’s ID.

- **Match-Discovery**: If $v$ and $u$ have picked each other, they send a notification message to their neighbors that they are matched and become inactive.

- **Removing-Matched-Neighbors**: Each unmatched vertex $v$ receives messages from its matched neighbors, and removes them from $v$’s adjacency list. The master sets the phase back to Max-Weight-Edge-Picking if there are unmatched vertices, otherwise terminates the computation.

### 6.2.5 Weakly Connected Components (WCC)

We implement the distributed algorithm from [74], which we had also reviewed in Chapter 5. WCC consists of a single phase, which is identical to the Forward-Traversal phase of the SCC algorithm from Section 6.2.1. In iterations, vertices propagate their IDs and keep the maximum ID they have seen until convergence. Similar to our implementation of Forward-Traversal, we break the phase into two subphases. In the Start phase, which takes one superstep, vertices initialize their \texttt{wccIDs} to their own IDs and propagate their IDs to their neighbors. In the Rest phase, vertices update their own \texttt{wccIDs} with the maximum \texttt{wccID} they have seen, and propagate their \texttt{wccIDs} (if updated). The Rest phase continues until \texttt{wccIDs} converge.

### 6.3 Finishing Computations Serially (FCS)

We now describe our first optimization technique, *Finishing Computations Serially* (FCS). FCS addresses slow convergence in algorithms by performing some serial computation on a tiny fraction of the input graph. Sections 6.3.1 and 6.3.2 give
a high-level description and explain the implementation of FCS, respectively. Section 6.3.3 discusses the benefits and overheads of FCS. In Sections 6.3.4 and 6.3.5, we apply FCS to the SCC algorithm from Section 6.2.1, addressing slow convergence due to large diameters and skewed component sizes, respectively. In Section 6.3.6, we apply FCS to the GC algorithm (Section 6.2.3), which converges slowly on graphs with small-size maximal independent sets. Sections 6.3.4–6.3.6 include experimental results.

### 6.3.1 High-level Description

Sometimes, an algorithm or a phase of the algorithm may converge very slowly, i.e., execute for a large number of supersteps, while executing on a very small fraction of the input graph, which we refer to as the *active-subgraph*. Since Pregel-like systems synchronize and exchange coordination messages at each superstep, slow convergence can significantly degrade performance. The premise of FCS is to avoid a large number of these small superstep executions by finishing the computation on a small active-subgraph serially, inside `master.compute()`. FCS monitors the size the active-subgraph. Once the size of the active-subgraph is below a threshold (5M edges by default), it sends the active-subgraph to the master, which performs the rest of the computation serially, and sends the results back to the workers. In the remainder of this section we call the vertices that are in the active-subgraph, i.e., those that can contribute to the computation in the remainder of the algorithm or a phase of the algorithm, as *potentially-active* (not to be confused with Pregel’s active/inactive flag). We call a vertex *certainly-inactive* otherwise.

FCS can be applied to algorithms in which the size of the active-subgraph, i.e., the subgraph that contains only the potentially-active vertices and the edges between them, shrinks throughout the computation. All of the algorithms we study in this chapter, except for WCC, have this “shrinking active-subgraph” property. For example, the SCC algorithm (Section 6.2.1) detects several components in each iteration and removes them from the graph, decreasing the size of the active-subgraph.
FCS can also be applied to individual phases, if the active-subgraph of the phase is shrinking. As an example, consider the Backward-Traversal phase of SCC. A vertex $v$ becomes certainly-inactive in two ways: (1) $v$ receives a message that contains its colorID, then $v$ discovers its component; or (2) no vertex in the graph propagates a message containing $v$'s colorID. In the second case, $v$ will not discover its component in the rest of the phase and will be assigned a new colorID in the next iteration. In contrast, recall the Forward-Traversal phase, in which vertices update and propagate their colorIDs by the maximum until convergence. In this phase, no vertex becomes certainly-inactive until the phase is complete, as we cannot tell with certainty that a vertex will not update its colorID later without observing the entire graph—a larger colorID may be propagating from another part of the graph to the vertex.

### 6.3.2 Implementation

Our implementation of FCS for an algorithm always uses three global objects, to store: (1) the number of edges in the active-subgraph; (2) the active-subgraph when serial computation is triggered; (3) the results of the serial execution (e.g., the component IDs in SCC), which are used by the potentially-active vertices to update their values in a superstep following serial execution. Implementations of FCS in specific algorithms and phases differ in the way the potentially-active vertices are identified, and the way the serial computation is performed inside `master.compute()`. In Sections 6.3.4–6.3.6 we explain the implementation differences among the algorithms and phases we apply FCS to.

### 6.3.3 Cost Analysis

FCS avoids additional superstep executions after serial propagation is triggered. On the other hand, it incurs the overhead of: (a) monitoring the size of the active-subgraph, which involves potentially-active vertices incrementing a global object; (b) serial computation at the master; (c) communication cost of sending the active-subgraph to the master and the results of the computation back to the workers;
(d) one superstep execution for vertices to read the results. The actual benefits and overheads depend on the algorithm and the graph, and we report experimental results in Sections 6.3.4–6.3.6. We note that we expect FCS to yield good benefits only when the algorithm or a phase converges very slowly. For example, FCS can be applied to MSF, in which the active-subgraph shrinks, but convergence is not slow.

We also note that in general one can trigger FCS as soon as the active subgraph is as large as the size of the memory of the master worker. Triggering FCS earlier would reduce the number of superstep executions and communication, at the expense of parallel computation. In practice, there will be an optimal threshold that minimizes the run-time of the computation. However, we advise keeping a very low threshold (such as 5M edges) instead of trying to find an optimal one. The motivation for FCS is to detect and avoid situations in which the algorithm cannot parallelize a computation due to the existence very small subgraphs, instead of performing serial computations on large subgraphs.

6.3.4 FCS for Backward-Traversal Phase of SCC (FCS-BT)

It has been observed that some real-world graphs have very large actual diameters but very small “effective diameters” 26 82. In other words, although most vertices are very close to each other, a small fraction of the vertices are far apart. On a graph with diameter $d$, the Backward-Traversal phase of SCC can take $d$ supersteps: a vertex $v$ might be at distance $d$ from the vertex that starts propagating $v$’s colorID. However, if the graph has a short effective diameter, most vertices may become certainly-inactive after a few supersteps, with only a small fraction of the vertices participating in the rest of the supersteps. To avoid some of these final superstep executions, we apply FCS to Backward-Traversal, and refer to this optimization as FCS-BT.

In order to monitor the size of the subgraph, we identify a vertex $v$ as potentially-active if: (1) $v$ has not yet received a message containing its colorID; and (2) there is at least one vertex propagating a message containing $v$’s colorID. We use a
global object \textit{propagating-colorIDs-set}, which is updated by vertices that propagate messages. Vertices that have not found their components look at this object to check whether their colorIDs are being propagated. For serial computation, we do a simple serial breadth-first search traversal at the master.

\textbf{Experiments}

To evaluate the overall run-time benefits, we applied FCS-BT to the SCC algorithm and ran experiments with and without the optimization. Figure \textbf{6.7} (ignore the “FCS-SCC” bars for now; see next section) shows the results. FCS-BT yields 1.3x and 2.3x run-time improvements on the \textit{uk-2005} and \textit{sk-2005} web graphs, respectively, when compared with the baseline SCC algorithm. In terms of supersteps, FCS-BT reduced the total number of supersteps by 28\% in \textit{uk-2005} (from 4546 to 3278) and 56\% in \textit{sk-2005} (from 6509 to 2857). FCS-BT does not show much improvement on non-web graphs, which have significantly smaller diameters than web-graphs. As a result, on non-web graphs the total number of supersteps the Backward-Traversal phases take is small ($\leq 50$) so eliminating them does not significantly improve performance. Over all five experiments, monitoring the size of the active-subgraph slowed down the run-time of supersteps before serial computation.
on average by 1.3%, which was minor compared to the benefits of FCS-BT.

### 6.3.5 FCS for SCC (FCS-SCC)

As observed in reference [82], real graphs can have skewed component sizes and exhibit a large number of very small components. Even if the small-size components comprise a small fraction of the original graph, detecting them may take a large number of iterations. Suppose 100 (say) small-size components are connected to each other. Then, in the Forward-Traversal phase, a vertex with a large ID from one component can color many other components and prevent their detection. (Recall the algorithm from Section 6.2.1) In an entire iteration, the algorithm may detect only a few of the 100 connected components. Therefore, we may have to run many iterations of SCC before detecting all 100 components. Applying FCS to the SCC algorithm as a whole (called *FCS-SCC*) can eliminate some of these iterations. In our implementation of FCS-SCC, we monitor the size of the active-subgraph simply by counting the edges of the vertices whose components have not been discovered. For serial computation, we use Kosaraju’s classic SCC algorithm [111].

**Experiments**

To evaluate the additional run-time benefits of FCS-SCC over FCS-BT, we repeated our experiments from Section 6.3.4 using FCS-BT in the Backward-Traversal phase and FCS-SCC for the overall algorithm. Figure 6.7 shows that FCS-SCC yields between 1.1x to 2.4x additional run-time improvement across our five datasets. FCS-SCC decreased the number of superstep executions by up to an additional 39%. Similar to FCS-BT, FCS-SCC performs better on web-graphs, which have very large diameters. Because of the diameter, executing the forward and backward traversals can take a very large number of supersteps; avoiding them increases performance significantly. We note that the time it took to monitor the active-subgraph and execute Kosaraju’s serial SCC algorithm was negligible compared to the rest of the computation in all of our experiments.
6.3.6 FCS for GC (FCS-GC)

The GC algorithm from Section 6.2.3 iteratively finds a maximal independent set (MIS) $M$ in the graph, gives all of the vertices in $M$ a color, and removes them from the graph. Thus, the active-subgraph shrinks throughout the computation. When executing GC on our datasets, we also observed that over time the active-subgraph gets denser, and as a result the independent sets get smaller. Nearing the end of the algorithm, we can be left with a small clique and need to find as many independent sets as there are vertices in the clique. Each iteration takes at least five supersteps (see Section 6.2.3), and hence on a clique of only 100 vertices, the algorithm executes at least 500 supersteps. We apply FCS to GC (called FCS-GC) to eliminate some of these supersteps. In our implementation, we use a serial version of the algorithm also based on finding independent sets: We compute the sets greedily, putting each vertex in a queue, then one by one putting each vertex $v$ in the set if none of $v$'s neighbors is already in the set.

Experiments

To evaluate the benefits of FCS-GC, we ran the GC algorithm with and without the optimization. Figure 6.8 shows a sample of our experiments. FCS-GC yields
between 1.1x to 1.4x run-time benefits, reducing the superstep executions between 10% to 20%. (For example, on the sk-2005 graph, FCS-GC reduced the number of superstep executions from 85980 to 69267.) On active-subgraphs of fewer than 5M edges, the serial computation takes under 5 seconds and is negligible compared to the rest of the computation. Similar to FCC-SCC, the cost of monitoring the active-subgraph was also negligible.

6.4 Storing Edges At Subvertices (SEAS) in MSF

6.4.1 High-level Description

Recall the MSF algorithm from Section 6.2.2. In the Supervertex-Formation phase, a supervertex $s$ receives and merges the adjacency lists of its subvertices, a high-cost operation. Our SEAS optimization instead stores the edges of a supervertex $s$ in a distributed fashion among all of its subvertices. With the adjacency lists of supervertex $s$ distributed, $s$ must pick its minimum-weight edge in a distributed fashion. Moreover, if $s$ is merged into another supervertex in a particular iteration, it has to notify its subvertices of the new supervertex they belong to. Nevertheless, the added work is offset by the improved performance of avoiding the very costly phase of sending and merging the adjacency lists of all subvertices. We note that SEAS can be applied to other algorithms that form supervertices during the computation but are not covered in this chapter, e.g., Karger’s classic randomized minimum cut algorithm [76] or the METIS graph partitioning algorithm [91].

6.4.2 Implementation

In our implementation of SEAS, subvertices store a pointer to their latest supervertices. The Min-Edge-Picking phase is now performed in two supersteps. In the first superstep, subvertices send their local minimum-weight edges to the supervertex. In the second superstep, a supervertex $s$ picks the minimum of its local edge and the edges it receives. The Supervertex-Finding and Edge-Cleaning-and-Relabeling phases are performed as usual. Instead of Supervertex-Formation, we perform a new
phase called \textit{New-Supervertex-Notification}, which takes three supersteps: (1) Suppose we are in iteration $i$. Every subvertex from iteration $i - 1$ sends a message to its latest supervertex containing its ID. (2) Every supervertex $s$ from iteration $i - 1$ sends a message back to its supervertices containing the ID of its new supervertex (possibly $s$ itself). (3) Subvertices from iteration $i - 1$ update their supervertices with the ID they receive.

\textbf{6.4.3 Cost Analysis}

Let $G_i(V_i, E_i)$ be the remaining graph in the $i$th iteration of the baseline MSF, and let $SV_i$ be the number of subvertices that have at least one edge remaining in the $i$th iteration when SEAS is on. In iteration $i$, SEAS avoids the computation of merging edge lists and inserting merged edges at supervertices. However, SEAS’s effects on communication are twofold. If we assume for simplicity that all of the edges of supervertices in iteration $i+1$ come from their subvertices, then SEAS roughly avoids $|E_i|$ amount of communication in iteration $i$. On the other hand SEAS incurs $3*|SV_i|$ additional communication: each subvertex sends one message to its supervertex in the Min-Edge-Picking and sends one and receives one message in New-Supervertex-Formation. Therefore, whether SEAS increases or decreases communication depends on how the sizes of $E_i$ and $SV_i$ change in each iteration. In effect, we avoid the communication and computation performed in the Supervertex-Formation phase, which is proportional to the number of edges in the graph, at the expense of increasing the cost of Min-Edge-Picking and incurring the costs of New-Supervertex-Formation, which are proportional to the number of vertices in the graph.

\textbf{6.4.4 Experiments}

To evaluate the effects of our SEAS optimization, we ran the MSF algorithm both with and without the optimization. Figure 6.9 shows the results on our five datasets (ignore the MSF+SEAS+ECOD bars for now). We find with SEAS the overall performance increases between 1.15x and 3x across our data sets. We also note
that SEAS decreases the total communication cost of the algorithm by up to 1.4x. Recall from the previous section that we expect SEAS to perform better when $|E_i|$ is large (SEAS avoids more communication and computation) and $|SV_i|$ is low (SEAS incurs less extra communication). Therefore SEAS’ performance depends on the ratio of $|E_i|$ to $|SV_i|$. Random-2.5B is the sparsest graph we experimented with (on average each vertex has only 5 vertices) and $|SV_i|$ was very high in each iteration of MSF (always $\geq 400$M vertices). As a result, SEAS improved performance less significantly for random-2.5B than other graphs.

6.5 Edge Cleaning On Demand

6.5.1 High-level Description

“Edge cleaning” is a common graph operation in which vertices delete some of their neighbors according to the neighbors’ values. For example, in the Removing-Matched-Neighbors phase of the MWM algorithm from Section 6.2.4, unmatched vertices remove edges to their matched neighbors. Other examples include the (not surprisingly named) Edge-Cleaning-and-Relabeling phase in SCC, as well as phases...
from other algorithms that are not covered in this chapter. (For example, Karger’s classic randomized minimum cut algorithm forms supervertices in iterations and “cleans” edges within supervertices [76].) In the natural implementation of edge cleaning on Pregel-like systems, vertices send messages to their neighbors in one superstep, and remove neighbors in another, possibly based on the contents of the messages. This implementation incurs a communication cost proportional to the number of edges in the graph, which might be expensive. In addition, sometimes an edge $e$ might be deleted unnecessarily: even if we keep $e$ around, $e$ may never be used again or otherwise affect the computation.

Our ECOD optimization technique keeps stale edges around instead of deleting them. Vertices “pretend” every edge is valid, and a stale edge $e$ is discovered and deleted only when a vertex attempts to use $e$ as part of the computation. Among the algorithms we studied in this chapter, ECOD can be applied to MWM and MSF, although implementations of ECOD on different algorithms differ in the way stale edges are discovered. We next describe the implementations and cost analysis of ECOD for each algorithm.

### 6.5.2 Implementation for MWM

Recall that in the Match-Discovery phase of our implementation of MWM in Section 6.2.4, when a vertex $v$ finds its match $u$ (say), $v$ sends all its neighbors a notification message and becomes inactive. In the following Removing-Matched-Nighbors phase, all of $v$’s unmatched neighbors remove $v$ from their adjacency lists. With ECOD, instead $v$ sends a message only to its neighbors that requested a match with $v$, and only those neighbors remove $v$ from their adjacency lists in the Removing-Matched-Nighbors phase; $v$’s other unmatched neighbors keep their “stale” edges to $v$. In addition, $v$ now stays active in order for its neighbors to discover their stale edges to $v$. If in a future iteration of the algorithm a vertex $z$ uses its stale edge to $v$ and requests to match with $v$, $v$ sends $z$ a notification message and $z$ removes its stale edge to $v$; otherwise, edge $(z, v)$ is never removed. As we discuss below, ECOD may decrease the number of vertices that converge in
each iteration; i.e. vertices that find a match or clean all their edges and discover that they will remain unmatched in the remainder of the computation. To avoid very slow convergence we switch to regular edge-cleaning if the number vertices that converge is below a threshold number of vertices (1% of all the vertices in the graph by default).

### 6.5.3 Cost Analysis for MWM

ECOD decreases the amount of communication and computation a matched vertex generates: instead of sending a message to all neighbors, matched vertices send messages only to those neighbors that request to match with them. As a result, similar to SEAS, ECOD effectively avoids costs that are proportional to the number of edges in the graph, at the expense of incurring costs that are proportional to the number of vertices. In our experiments (discussed momentarily), we observed significant performance benefits from this tradeoff. In addition, ECOD can avoid the communication and computation of deleting some edges unnecessarily. For example, consider an edge \((v, u)\), and assume that \(v\) and \(u\) both match with other vertices in the first iteration. Without ECOD, they would send each other unnecessary notification messages.

On the other hand, ECOD may slow down the convergence of the algorithm, decreasing the number of vertices that match (or that discover not to match any other vertex) in each iteration, which increases the number of iterations. Consider
the simple graph in Figure 6.10. Without ECOD, the algorithm takes two iterations. In the first iteration, vertex 2 matches with 3, and 4 matches with 5. In the second iteration, 1 matches with 0. Using ECOD, the algorithm takes five iterations. In the first four iterations, vertex 1 attempts to match with its heaviest edges 5, 4, 3, and 2 one by one and fails. After removing these edges, it finally matches with 0 in the fifth iteration.

6.5.4 Implementation for MSF

Recall that in the Edge-Cleaning-and-Relabeling phase of MSF (Section 6.2.2), vertices send their supervertex IDs to their neighbors in the first superstep. In the second superstep, vertices remove their neighbors that have the same supervertex ID from their adjacency lists. With ECOD, we omit this phase completely. As a result, during Min-Edge-Picking, vertices cannot pick their minimum weight edges directly in one superstep, as some of their edges may be stale. Instead, we execute a Stale-Edge-Discovery phase, during which vertices discover whether or not their minimum weight edges are stale. In the first superstep, each vertex $v$ tentatively picks its minimum weight edge $u$ (say) and sends a “question” message to $u$ containing $v$’s ID and $v$’s supervertex ID, $i$ (say). In the second superstep, if $u$ belongs to a different supervertex $j$ (say), it sends an answer message back to $v$ containing the value $j$. In the third superstep, if $v$ receives an answer message, it successfully picks $u$ as its minimum-weight edge and relabels it to $j$; otherwise $v$ removes $u$. We run the Stale-Edge-Discovery phase a threshold number of times (two by default). We then run a final All-Stale-Edges-Discovery phase, in which vertices that have not yet successfully picked their minimum-weight edges send a question message to all their neighbors and clean all of their stale edges.

6.5.5 Cost Analysis for MSF

ECOD avoids the communication and computation of the Edge-Cleaning-and-Relabeling phase. On the other hand, ECOD incurs the extra communication and computation cost of question and answer messages in the new Stale-Edge-Discovery and
All-Stale-Edges-Discovery phases. Without ECOD, in each Edge-Cleaning-and-Relabeling phase, a message is sent over each remaining edge in the graph. As a result, if an edge \((v, u)\) is not picked as a minimum-weight edge or cleaned for \(j\) iterations, \(v\) will send exactly \(j\) messages over \((v, u)\). In contrast with ECOD, \(v\) can exchange as few as one question and one answer message with \(u\), and certainly no more than \(j\) messages. There are two cases:

1. \((v, u)\) is picked as a minimum-weight edge in the \(j\)th iteration: Then, in the \(j\)th iteration, \(v\) exchanges one question and one answer message with \(u\) in the Stale-Edge-Discovery phase, and successfully picks \(u\) as its minimum-weight edge. In the first \(j - 1\) iterations, \(v\) sends a message to \(u\) only if \(v\) cannot pick its minimum-weight edge in the Stale-Edge-Discovery phases, and has to clean all its stale edges in All-Stale-Edges-Discovery.

2. \((v, u)\) is cleaned in the \(j\)th iteration: \(v\) exchanges one questions and answer message with \(u\) and cleans \((u, v)\). For the first \(j - 1\) iterations, the situation is exactly as case (1) and \(v\) sends a message to \(u\) only if \(v\) has to run All-Stale-Edges-Discovery in the first \(j - 1\) iterations.
As a result $v$ cumulatively sends between 1 and $j$ messages to $u$ in both cases. Therefore the amount of communication of ECOD is less than or equal to baseline MSF’s. We have observed in our experiments that with ECOD, significantly fewer messages are sent over edges in general.

On the negative side, ECOD increases the overall number of superstep executions. It avoids two superstep executions of the Edge-Cleaning-and-Relabeling phase but executes the three superstep Stale-Edge-Discovery phase several times.

### 6.5.6 Experiments

We evaluated ECOD on both MWM and MSF. As shown in Figures 6.11 and 6.12, ECOD improves run-time of MWM by up to 1.45x and decreases the total communication cost between 1.3x to 3.1x across our data sets. ECOD increased the number of supersteps between 1.7x to 2.2x. For MSF, we applied ECOD in combination with our SEAS optimization (Section 6.4). The results were included in Figure 6.9. When used in combination with SEAS, ECOD yields between 1.2x and 3.3x additional run-time benefit. ECOD also decreased the total communication cost of MSF by up to 1.9x across our experiments. For MSF, the overall
performance improvements on uk-2005 were modest because ECOD improved the communication cost only by 1.03x.

6.6 Single Pivot Optimization (SP)

In this section we describe how skew in component sizes can yield unnecessarily high communication cost in the component detection algorithms we study. We review the Single Pivot (SP) optimization, which was originally described in reference [104] to improve the performance of the WCC algorithm (Section 6.2.5). We show that the optimization can also be applied to the SCC algorithm from Section 6.2.1.

While reference [104] reports minor performance benefits running WCC on two small synthetic graphs (fewer than 240M edges), we report major benefits on our larger real-world graphs for both WCC and SCC.

6.6.1 High-level Description and Implementation

In addition to a large number of small-size components, graphs with skewed component sizes typically exhibit a single “giant” component, which contains a significant fraction of the vertices in the graph [26]. The SP optimization, originally described in [104] for WCC, is designed to detect giant components efficiently. Initially, the optimization picks a single vertex \( r \) (called the pivot) and finds the component that \( r \) belongs to, by propagating \( r \)'s ID along its neighbors (either in WCC or the Forward-Traversal phase of SCC). The process is repeated until a large component is found, or a threshold number of iterations is reached. At that point the original algorithm is used for the remainder of the graph. To implement SP, we added a new initial phase, Random-Pivot-Picking, to the WCC and SCC algorithms. In this phase, every vertex updates a custom global object that picks one of the vertices as pivot uniformly at random. Then, only the pivot starts propagating its ID in the WCC algorithm and the Forward-Traversal phase of the SCC algorithm.
6.6.2 Cost Analysis

The existence of a giant component can incur unnecessary costs in the WCC algorithm: Let $w$ be the vertex with maximum ID in the giant component. Then all propagation messages inside the giant component, except those that contain the ID of $w$, are unnecessary: they will not contribute to the final $wccID$ values.

The situation is potentially worse for the SCC algorithm. Let $CON$ denote the (usually) larger set of vertices that are connected to the giant (strongly-connected) component, and let $w$ be the vertex with maximum ID in $CON$. There are two possibilities: (1) If $w$ is in the giant component, then in the Forward-Traversal phase, we incur the same unnecessary propagation messages as in WCC. (2) If $w$ is not in the giant component, then we will detect the SCC that $w$ belongs to, which typically is much smaller than the giant one. As a result, much of the computation and messages will be repeated in a later iteration.

If SP picks a pivot vertex from the giant component, it avoids all of the unnecessary propagation messages and computation costs of detecting the giant component. On the other hand, for each iteration failing to pick a pivot $r$ from the giant component, SP decreases the parallelism of the algorithms: instead of detecting multiple components in an iteration, SP detects only $r$’s component. As a result, SP might increase the total number of superstep executions. Also, in the Forward-Traversal phase of SCC, picking a pivot from outside the giant component can result in unnecessary propagation messages among many vertices that do not belong to the pivot’s component. However, SP always incurs less communication cost than baseline WCC. This follows from the observation that with SP there will be exactly one message sent over each edge in the component of pivot $r$, which is the cost incurred only in the first iteration of baseline WCC.

6.6.3 Experiments

Figure 6.13 shows the results of our experiments adding SP to WCC. Because SP introduces randomness, we repeated each experiment three times and report the average of our measurements. As shown, SP improves the run-time on average
between 2.7x to 7.4x across all of our input graphs. Note because the giant component in our undirected graphs is very large, consisting of more than 90% of the vertices, in all of our trials SP detected the giant-component in the first try.

For SCC, SP can be used in combination with FCS-BT and FCS-SCC. To measure the additional benefits of SP, we repeated our experiments from Section 6.3.5 with SP. Figure 6.14 shows the results. We again ran each experiment three times and report the average run-times. SP yields between 1.1x to 2.1x additional runtime benefits. In all of our trials SP detected the giant component within two tries. We note that the improvements were modest on our web graphs. The runtime of SCC is dominated by the large number of supersteps the Forward-Traversal phases execute due to large diameters of the web graphs. None of the optimization techniques we know of can avoid these superstep executions. Finally, we note that considering all our optimizations together (FCS-BT, FCS-SCC, and SP), the run-time of the baseline SCC algorithm improved between 1.45x and 3.7x.

Figure 6.13: SP on WCC, Large-EC2(100, 100).

6.7 Related Work

The SP optimization was introduced originally in [104] for finding weakly-connected components in undirected graphs. They evaluate the optimization on two small
synthetic graphs that exhibit skew. They report minor performance benefit for one graph and minor performance loss for the other. We report significant performance improvements when finding both strongly and weakly-connected components in large graphs.

A variety of parallel algorithms exist for the graph problems we cover in this chapter: SCC [44, 117], MSF [37, 78], MWM [41, 103], and GC [47, 68]. Some of these algorithms are designed for the PRAM model and are not suitable for vertex-centric implementations, and some have been observed to not perform well on large-scale real graphs [63, 117]. To the best of our knowledge, none have been implemented on Pregel-like systems.

A variety of other graph algorithms have been implemented on Pregel-like systems. The original Pregel paper [87] describes several algorithms including PageRank, single-source shortest paths to all vertices, a randomized maximal matching in a bipartite graph, and an algorithm to cluster similar vertices in a graph. Reference [104] describes Pregel implementations of several social network analysis algorithms: computing the degrees of separation and the clustering coefficients of vertices, computing the diameter of a graph, and finding the k-cores, triangles, and k-trusses in a graph. Reference [64] describes Pregel implementations of two other algorithms used in social network analysis: computing the conductance [75] of a
graph and an approximation algorithm to compute the betweenness centrality [24] of vertices in a graph. Aside from the SP optimization in [104], none of these references describe algorithm-level optimizations to their baseline implementations.
Chapter 7

Debugging Graph Algorithms For Pregel-like Systems

In this chapter we tackle the challenge of debugging programs written for Pregel-like systems. Despite being a core component of programmers’ development cycles, very little work has been done on debugging in these systems. We based our work on interviews with several Apache Giraph and GPS programmers (hereafter referred to as “users”) and studied how they currently debug their `vertex.compute()` functions. As we explain momentarily, we found users’s current debugging cycles to be tedious and time-consuming. Based on our observations, we designed and developed Graft, a new replay-style debugger that is tailored specifically for the needs of Giraph users. Although we have developed Graft for the open-source Apache Giraph system so that it would achieve wide usage, with only minor modifications Graft could be used for other Pregel-like systems, such as our GPS system. In the remainder of this section, we first describe the way users currently debug their programs and then give a summary of Graft’s approach.

In our interviews with users we found that the following three steps were common when debugging Giraph or GPS programs: (1) Users add print statements to their code to capture information about a select set of potentially “buggy” vertices, e.g., vertices that are assigned incorrect values, send incorrect messages, or throw exceptions. The captured set of vertices is typically quite small, sometimes
containing as little as a single vertex with its neighbors, because it is slow to log, and difficult to inspect the information of a large number of vertices. (2) Then, by parsing the log files of the Pregel-like system, users inspect the captured vertex information and mentally “replay” their graph algorithms superstep by superstep, until they narrow in on the most suspicious vertices and supersteps. (3) Finally, they return to their code and try to identify the part of vertex.compute() that must have executed on the suspicious vertices and supersteps, hoping to find the bug.

Pregel-like systems sometimes run on hundreds of machines and produce gigabytes of log files. As a result, producing textual information with print statements, and then parsing and reasoning about this information can be tedious and slow. Unfortunately, existing replay debuggers, e.g., [6, 48], also do not help users. These debuggers capture and replay all low-level system calls made by a distributed application, such as memory reads and writes to the network drivers, which are usually not relevant for diagnosing bugs inside vertex.compute() functions. They also do not provide any replay functionality specific to Pregel’s vertex-centric graph computations. Graft’s approach is motivated by the three manual steps we observed in users’ current debugging cycles, which we call capture, visualize, and reproduce, respectively:

- **Capture**: Users describe programmatically which vertices they are interested in capturing (details in Section 7.1.1). Graft captures the entire context information for these vertices, across all supersteps or a user-defined selection of supersteps. It is expected that the selected set of vertices will be relatively small, and the rich API encourages applying selective criteria.

- **Visualize**: Graft includes a graph-specific and superstep-based visual interface for users to replay the algorithm’s effects on the vertices whose contexts have been captured. Users can see how the values and messages of these vertices change from superstep to superstep, narrowing in on suspicious values, messages, or exceptions.

- **Reproduce**: The last step involves code inspection, for which we rely on the user’s integrated development environment (IDE), such as Eclipse [39] or
IntelliJ [66]. The context that Graft captures is sufficient to generate code that can reproduce exactly those lines of \texttt{vertex.compute()} that executed for a specific vertex and superstep. The user copies this code into the IDE and uses its line-by-line debugger. The code that Graft generates is a unit test file, which the user may wish to turn into a real unit test case for his \texttt{vertex.compute()} function.

Graft similarly helps users debug their \texttt{master.compute()} functions and has features to construct end-to-end tests for Giraph programs (Section 7.1.4). Graft is open-source and fully integrated into Apache Giraph’s main code base. The rest of this chapter is organized as follows:

- In Section 7.1 we describe the architecture and implementation of Graft.
- In Section 7.2 we give three example debugging scenarios with Graft. One of our examples is based on a report from an actual Graft user.
- In Section 7.3 we present experiments evaluating Graft’s performance overhead by comparing running Giraph programs with and without Graft.
- In Section 7.4 we discuss a limitation of our approach when a program has external dependencies.
- In Section 7.5 we discuss related work.

7.1 The Graft Debugging Tool

Figure 7.1 gives an overview of Graft’s architecture. In the following subsections we explain the architecture and components in terms of the capture, visualize, and reproduce functionalities they implement.

7.1.1 Capture: The DebugConfig File and Graft Instrumenter

Users extend and implement a \texttt{DebugConfig} class to specify the vertices they are interested in capturing. Users can instruct Graft to capture all vertices in five categories: (1) vertices specified by their IDs, and optionally their neighbors; (2) a random set of a given number of vertices, and optionally their neighbors; (3) vertices
that violate a specified constraint on vertex values; (4) vertices that send a message value that violates a specified constraint; and (5) vertices that raise exceptions. Alternatively, a user may specify that all active vertices should be captured. Users can also limit in which supersteps Graft captures vertices; by default Graft captures vertices in each superstep. For example, the DebugConfig shown in Figure 7.2 instructs Graft to capture 5 random vertices and their neighbors, and all vertices that send negative-valued messages, across all supersteps.

The Graft Instrumenter takes as input the user’s DebugConfig file and vertex.compute() function. It uses Javassist [70] to wrap the vertex.compute() around a new instrumented one, which is the final program that is submitted to Giraph. When Giraph calls compute() on the instrumented code of a vertex \( v \), the code calls the user’s original vertex.compute() function, intercepting messages and value updates so it can check constraints. After the user’s
public class RWDebugConfig {
    public int numRandomVerticesToCapture() { return 5; }
    public boolean captureNeighborsOfVertices() { return true; }
    public boolean messageValueConstraint(Message msg, ID srcID, ID dstID,
        int superstep) {
        return msg.value ≥ 0;
    }
}

Figure 7.2: A DebugConfig file.

The vertex.compute() function returns, the instrumented function checks whether \( v \) should be captured: (1) if \( v \) is in one of the five possible categories of DebugConfig (above); or (2) if the user instructed Graft to capture all active vertices.

To capture \( v \), the instrumented code logs the context of \( v \), along with the messages that \( v \) sent, to a trace file in the Hadoop Distributed File System (HDFS)—the distributed file system that is used by Giraph [9]. The context of a vertex.compute() function consists of the five pieces of data that the Giraph API exposes to a vertex: (1) the vertex ID; (2) its outgoing edges; (3) its incoming messages; (4) a set of aggregators (see below); and (5) default global data consisting of the current superstep number and the total number of vertices and edges in the graph. The Giraph API is an implementation of the original Pregel API; we refer the reader to Chapter 5 for a description of the Pregel API. We note that there are some limitations to capturing only this small set of data, which we discuss in Section 7.4. We also note that the design of our debugger assumes that the captured set of vertices is relatively small, both for usability and performance. As a “safety net”, we have built in an adjustable threshold, specifying a maximum number of captures, after which Graft stops capturing.

7.1.2 Visualize: Graft GUI

Users inspect the captured vertices through the Graft GUI in their browsers; screenshots are shown in Figures 7.3-7.5. Figures 7.3 and 7.4 are from the GC graph coloring algorithm (also described in Chapter 6), so the vertex values are color
Figure 7.3: Node-link View.

Figure 7.4: Tabular View.

assignments. Figure 7.5 is from the \textit{WCC} connected components algorithm (described in Chapter 6), where the values are vertex IDs. The GUI’s Node-link View is for visualizing a small number of selected vertices and their graph structure. The Tabular View is for visualizing larger numbers of selected vertices. The Violations and Exceptions View is for inspecting those vertices that violate constraints or raise exceptions. We explain these views in detail, and explain the Reproduce Context buttons.
• **Node-link View (Figure 7.3):** Shows the vertices that were captured by ID or by random selection as a node-link diagram. The IDs and values of vertices are displayed on the nodes, and edge values (if any) are displayed on the links. If a vertex is inactive in the displayed superstep, its color is dimmed. For example, in Figure 7.3 vertex 567890 is active in superstep 231, while vertex 567891 is inactive. If \( u \) is a neighbor of a captured vertex \( v \) but \( u \) is not captured, then \( u \) is shown as a small node and only \( u \)'s ID is displayed. By clicking on a captured vertex, users can further see the incoming and outgoing messages of the vertex (omitted in the figure). In the upper-right corner, we see the aggregator values and default global data for the displayed superstep. Users can replay how vertex values, active/inactive states, incoming and outgoing messages, and aggregators change superstep by superstep by pressing the Next and Previous superstep buttons. On the left side, there are three boxes labeled as \( M \) for message value constraint, \( V \) for vertex value constraint, and \( E \) for exception. Green labels indicate that no violation or exception has occurred on any vertex during the superstep that is being displayed, and red labels indicate otherwise. When a label is red, users can click on the label and go to the Violations and Exceptions View to see the vertices that violated the constraint or raised an exception (see below).

• **Tabular View (Figure 7.4):** If the user is debugging a large number of

![Figure 7.5: Violations and Exceptions View.](image-url)
vertices, then the node-link diagram becomes difficult to use. The Graft GUI alternatively provides a tabular view of vertices, shown in Figure 7.4, where each row displays the summary of a captured vertex and can be expanded to see the entire context. This view also provides a simple search feature to find vertices by their IDs or their neighbors’ IDs, their values, or messages they have sent and/or received. Similar to the Node-link View, the Tabular View allows stepping through vertex contexts superstep by superstep.

- **Violations and Exceptions View (Figure 7.5):** Tabular view of vertices that have violated a vertex value or message constraint, or have raised an exception. Shows the constraint-violating vertex or message value, or the error message and stack trace of the exception.

- **Reproduce Context Buttons:** When the user wants to further investigate a vertex \( v \), he clicks the “Reproduce Vertex Context” button that is available in all of the views. The GUI displays a piece of Java code the user can copy into his IDE to replay line-by-line which lines of \( v \text{.compute}() \) executed for the selected vertex. We will explain the details of this Java code in the next section.

### 7.1.3 Reproduce: Context Reproducer

When a user clicks on the Reproduce Vertex Context button for a vertex \( v \) in superstep \( i \), Graft’s Context Reproducer takes the trace file of \( v \) from superstep \( i \) and generates a JUnit test file \([73]\), with the help of Apache Velocity’s template-based code generation \([124]\). The generated code replicates the context of \( v \) through mock objects, then calls \( v \text{.compute}() \). The user can copy this code into his IDE and use the IDE’s line-by-line debugger to see exactly which lines of \( v \text{.compute}() \) executed in superstep \( i \). Using Mockito \([93]\), the mock objects emulate the behavior of the different objects that the \( v \text{.compute}() \) code depends on, such as \( v \)'s data and ID, aggregators in superstep \( i \), or the default global data exposed to \( v \), thus replicating the context under which \( v \text{.compute}() \) executed in the cluster. (We will discuss a limitation of this approach in Section 7.4.)
public class GCVertexGraftTest {
  @Test
  public final void testCompute() {
    // mock default global data
    GraphState graphState = mock(GraphState.class);
    when(graphState.getSuperstep()).thenReturn(41);
    when(graphState.getTotalNumVertices()).thenReturn(1000000000L);
    when(graphState.getTotalNumEdges()).thenReturn(3000000000L);
    // mock aggregators
    WorkerAggregatorUsage aggr = mock(WorkerAggregatorUsage.class);
    when(aggr.getAggregatedValue("phase")).thenReturn(
      new Text("CONFLICT-RESOLUTION"));
    // Reproduce the vertex’s value.
    Vertex<LongWritable, GCVertexValue> vertex = conf.createVertex();
    vertex.initialize(new LongWritable(672), new GCVertexValue(null, "TENTATIVELY_IN_SET");
    // Reproduce the vertex’s outgoing neighbors.
    ReusableEdge<LongWritable, NullWritable> edge =
      conf.createReusableEdge();
    edge.setTargetVertexId(new LongWritable(671));
    vertex.addEdge(edge);
    edge.setTargetVertexId(new LongWritable(673));
    vertex.addEdge(edge);
    // Reproduce the vertex’s incoming messages.
    ArrayList<GCMessage> inMsgs = new ArrayList();
    inMsgs.add(new GCMessage("NBR_IN_SET", 671))
    inMsgs.add(new GCMessage("NBR_IN_SET", 673))
    // Finally, call vertex.compute().
    ((GCVertex) conf.createVertexClass()).compute()(vertex, inMsgs);}

Figure 7.6: JUnit test case generated by Graft Context Reproducer.

Figure 7.6 shows an example JUnit test file replicating the context of a vertex with ID 672 in superstep 41 for a program called GCVertex (graph coloring). In lines 5-8, the mock GraphState object emulates the default global data that Giraph exposed to vertex 672, which consists of the superstep number (41), the total number of vertices (1 billion), and the total number of edges (3 billion).
Lines 10-12 emulate the aggregators that were exposed to vertex 672. Lines 14-28 emulate the value, outgoing edges, and incoming messages of the vertex. Finally line 30 calls the user’s `GCVertex.compute()` function. We believe the ability to replay the line-by-line execution of `vertex.compute()` for a specific vertex and superstep is a very powerful feature of Graft. It was also the most challenging component of Graft to implement.

Although JUnit is a standard testing framework to unit-test pieces of Java code, our primary use of JUnit is not to unit-test `vertex.compute()`, but to reproduce the context of `vertex.compute()` for a particular vertex in a particular superstep. However, users can edit the JUnit test code generated by Graft and turn it into a real unit test.

### 7.1.4 Other Features

#### Debugging Master.compute()

If the Giraph program submitted to Graft contains a `master.compute()` function, Graft automatically captures its context—just the aggregator values—in every superstep. By clicking on the “Reproduce Master Context” button in the GUI, users can generate a JUnit test file reproducing `master.compute()` execution in a particular superstep.

#### Small Graph Construction and End-To-End Tests

We discussed that the JUnit test files Graft produces can be retained as real unit tests. In addition to unit tests, which test `compute()` functions in a single superstep, users may wish to write “end-to-end” tests. In an end-to-end test, a user may construct a small graph, then execute the program locally from first superstep until termination, to verify that Giraph’s final output is correct on the small graph. The Node-link View of the Graft GUI has an “offline” mode to help users construct small graphs for testing purposes. Users can add vertices and draw edges between vertices, and edit the values of the vertices and edges in an easy fashion. Users can also select premade graphs from a menu. After constructing a graph in the GUI,
the user can obtain either a text file that contains the adjacency list representation of the graph and use it in an end-to-end test, or can obtain an end-to-end test code template, which contains code that constructs the graph programatically.

7.2 Debugging Scenarios

To illustrate Graft’s debugging capabilities, we describe three debugging scenarios that cover different features of Graft. Two of the scenarios demonstrate how Graft helps users debug their `vertex.compute()` functions, while the third demonstrates how Graft can also help users find problems in their input graphs. In each scenario, we first describe a bug inside a Giraph program or the input graph and then walk through how a hypothetical Graft user “Bob” can fix the bug. We note that although our first example is drawn from a real user, our second and third examples are hypothetical and meant to demonstrate features of Graft not covered in the first example.

7.2.1 Graph Coloring Scenario

Our first scenario is based on a report from an actual Graft user and illustrates the feature of capturing and inspecting a random set of vertices for debugging `vertex.compute()`. It is based on the Giraph implementation of the graph coloring algorithm GC from Chapter 6. We give a brief overview of the algorithm here; for details see Chapter 6.

Recall that graph coloring is the problem of assigning a color to each vertex of an undirected graph so that no two adjacent vertices have the same color, using as
few colors as possible. GC iteratively finds a maximal independent set (MIS) $S$ of vertices, i.e., a maximal set of vertices such that no pair of vertices are adjacent. The algorithm assigns the vertices in $S$ a new color, then removes them from the graph, until there are no vertices left in the graph. GC uses Luby’s algorithm for finding an MIS. We describe a simplified version of the algorithm. Initially each vertex has a type value *Unknown*. Each iteration of Luby’s algorithm \[85\] consists of three phases, each taking a single superstep:

- **Selection**: Each *Unknown* vertex $v$ randomly sets its type to *TentativelyInS* and sends its outgoing neighbors a message containing $v$’s ID.

- **Conflict-Resolution**: Each *TentativelyInS* vertex inspects the IDs of its messages. If $v$ has the minimum ID among its messages, it sets its type to *InS* and sends an empty message to its neighbors. Otherwise, $v$ sets its type back to *Unknown*. As proved in reference \[85\], putting only the minimum ID vertex into $S$ guarantees that two neighbor vertices are not added to $S$.

- **NotInS-Discovery**: If $v$ receives a message, $v$ sets its type to *NotInS*.

The iterations continue until all vertices have type either *InS* or *NotInS*.

Figure 7.7 shows a snippet of an implementation of GC, which contains a bug in the Conflict-Resolution phase when constructing an MIS. The bug is between lines 8 and 13. In this code snippet, a vertex incorrectly puts itself into the MIS when its ID is not the minimum among its messages. This bug can cause multiple adjacent vertices to put themselves into the MIS together and eventually be assigned the same color. As a simple example, consider a clique of three *Unknown* vertices and suppose that in the Selection phase, all of the vertices tentatively put themselves into the MIS. Then in the Conflict-Resolution phase, the vertex with the minimum ID sets its type to *Unknown* and the other two vertices, which are adjacent, set their types to *InS*.

Our hypothetical user Bob can detect this bug with Graft as follows. As a sanity check for his code, Bob writes a simple DebugConfig file and captures a set of ten random vertices and their neighbors. Bob then goes to the final superstep from the GUI to verify that no two adjacent vertices have been assigned the same
public class BuggyGCVertex {
    public void compute(Iterable<GCMessage> msgs) {
        ....
        case CONFLICT_RESOLUTION:
        long myID = getID();
        long minID = findMinID(myID, msgs);
        // Bug below: if condition should be (myID != minID)
        if (myID == minID) {
            getValue().type = Unknown;
        } else {
            getValue().type = InS;
            // Send an empty notification msg to nbrs that the current vertex is
            // in set.
            sendMessageToNbrs(new Message());
        }
    }
}

Figure 7.7: Buggy implementation of GC.

color. Upon inspection, he sees that some vertices and their neighbors are assigned the same color, indicating a bug in his implementation. He then focuses on two such vertices in the GUI, say 672 and 673, and replays the computation superstep by superstep, and investigate how they end up with the same color. He notices that in superstep 41 both vertices change their types from Unknown to InS. He then generates a JUnit test case from the GUI replicating the lines of code that executed for vertex 672 in superstep 41. During line-by-line replay inside an IDE, he identifies that the condition inside the if statement on line 8 is incorrect and fixes the bug.

7.2.2 Random Walk Scenario

Our second scenario illustrates debugging using message value constraints, using the RW-n random walk simulation algorithm from Chapter 5. Recall that in RW-n, each vertex starts with an initial number of $n$ walkers. In each superstep, each vertex declares a local counter for each of its neighbors, randomly increments one
of the counters by one for each of its walkers, then sends the counters as messages to its neighbors.

Figure 7.8 shows a buggy implementation of RW-n. To optimize the memory consumption, the code in the figure declares the counters as 16-bit short primitive types on line 6. However, if a vertex $v$ has a large number of walkers, either because we started with a high number of initial walkers or $v$ has a large number of incoming neighbors, then the number of walkers from $v$ to $u$ can exceed Java’s max short value of 32767. In this case $v$ might send $u$ a negative number of walkers making the algorithm incorrect.

Bob can detect this bug using Graft’s message value constraint feature. As a sanity check, Bob writes a DebugConfig file with the constraint that messages are non-negative. After the run, he inspects the GUI and sees that the message value constraint icon is red in some supersteps. In the Violations and Exceptions View he identifies that some vertices are sending negative messages. He generates a JUnit test case from a vertex $v$ that has sent a negative message, and detects that the bug is due to overflowing of the short type counters.
7.2.3 Maximum-Weight Matching Scenario

Our third scenario illustrates that Graft can also be used to find errors in the input graph. We use the MWM algorithm from Chapter 6 for computing an approximate maximum-weight matching of an undirected graph. Recall that in each iteration of MWM, vertices select their maximum-weight neighbors. If $u$ and $v$ select each other, the edge $(u, v)$ is added to $M$, and $u$ and $v$ (along with all edges incident to them) are removed from the graph. The iterations continue until there are no vertices left in the graph.

Recall that graphs are directed in Pregel-like systems. As a result, undirected graphs are specified as directed graphs in which each edge $(u, v)$ has its corresponding edge $(v, u)$ stored in vertex $v$. The MWM algorithm can enter an infinite loop if the input graph has asymmetric weights on two copies of the same same edge. For example, consider the triangle graph shown in Figure 7.9. Running MWM on this graph will result in an infinite loop: vertex 1 will select vertex 2, vertex 2 will select vertex 3, and vertex 3 will select vertex 1. Therefore no two vertices will select each other and the same selection will continue indefinitely.

Bob can detect this problem with an input graph that contains asymmetric edge weights as follows. We assume that the number of asymmetric edges are small. He runs MWM on the erroneous graph and sees that the algorithm enters an infinite loop. He then runs MWM with Graft and captures all active vertices after a very large superstep number, say 1000, by which point most vertices are matched. Therefore the active graph is fairly small and only contains the edges

![Figure 7.9: Graph with asymmetric edge weights.](image)
### CHAPTER 7. GRAFT

#### Table 7.2: Graph datasets for performance experiments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Vertices</th>
<th>Edges</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sk-2005</td>
<td>51M</td>
<td>1.9B (d), 3.5B (u)</td>
<td>Web graph of the .sk domain from 2005</td>
</tr>
<tr>
<td>twitter</td>
<td>42M</td>
<td>1.5B (d), 2.7B (u)</td>
<td>Twitter “who is followed by who” network</td>
</tr>
<tr>
<td>bipartite-2B-6B</td>
<td>2B</td>
<td>12B (u)</td>
<td>A 3-regular bipartite graph</td>
</tr>
</tbody>
</table>

#### Table 7.3: DebugConfig configurations.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC-sp</td>
<td>Captures 5 specified vertices</td>
</tr>
<tr>
<td>DC-sp+nbr</td>
<td>Captures 5 specified vertices and their neighbors</td>
</tr>
<tr>
<td>DC-msg</td>
<td>Specifies constraint that message values are non-negative</td>
</tr>
<tr>
<td>DC-vv</td>
<td>Specifies constraint that vertex values are non-negative</td>
</tr>
<tr>
<td>DC-full</td>
<td>Captures 10 specified vertices and their neighbors, specifies message and vertex constraints, and checks for exceptions</td>
</tr>
</tbody>
</table>

with asymmetric edges. In the GUI, he notices that some of the edge weights in the small remaining graph are asymmetric, which is the cause of the algorithm not converging.

### 7.3 Performance

To evaluate the performance overhead of Graft, we ran the three algorithms from our scenarios with and without Graft, over large data sets. The graphs and the DebugConfig configurations we used are shown in Tables 7.2 and 7.3, respectively. We used a cluster of 36 machines, each with 50 GB of RAM, running Red Hat Linux OS and Giraph 1.0.0. We used the 3X software tool to automate and manage our experiments [112]. Our 3X repository, which contains the exact configurations of our experiments and can be used to reproduce them, is available to download [55].

Figure 7.10 shows a sample of our experiments. In the figure, each cluster of bars corresponds to one algorithm and dataset as indicated on the x-axis. Individual bars of the cluster show the overhead of running the algorithm and dataset on a particular DebugConfig or without Graft (“no-debug” bars). The heights of the
Figure 7.10: Graft’s performance overhead.

bars indicate the relative total run-time of a specific experiment against the no-debug configuration (normalized to 1.0 for each experiment). The number on each bar indicates the total number of vertex captures in the experiment. For example, the last bar of the RW-tw cluster indicates that the overhead of running RW on the twitter graph with the DC-full configuration was 26% compared to running without Graft, and Graft captured 24213 vertices during the experiment. We ran each experiment five times and report the average runtimes for each experiment. The variances across runs were small and are shown by the error bars.

Overall, Graft’s overhead when capturing a set of 5 vertices randomly or by their IDs was less than 16% (DC-sp), and less than 17% when also capturing the neighbors of these vertices (DC-sp+nbr), across all of our experiments. The overhead was less than 20% when checking message and vertex value constraints (DC-vv and DC-msg). Finally, Graft’s overhead when capturing a specified set
of vertices and their neighbors, and also the vertices that violate a message and vertex value constraint was less than 29% (DC-full). The number of captures varied between 1 and 1,246,151.

7.4 Limitation: External Dependencies

Because the APIs of Pregel-like systems are implemented in standard programming languages such as Java or C++, users can, in principle, write compute() functions that depend on “external” data beyond the five pieces of data that are exposed by the API. For example, programs can read external data from an HDFS file in Giraph, or vertices that are executed on the same Giraph worker thread can share information amongst themselves through static Java objects. If a user’s vertex.compute() code has such external data dependencies, then the limited context information that Graft captures will not be enough to reproduce exactly what happened, which is a limitation of our approach. That said, we believe that such external data dependencies should be avoided in general, and every algorithm we have seen implemented in the code repositories of Pregel-like systems either had no external dependencies or could be implemented without such dependencies. There are ways, however, to capture external dependencies: we could, for example, employ techniques in some replay debuggers [6, 18] that capture enough context information to replay the entire execution of processes, which includes all function calls. However, Graft is intended to be a very lightweight debugger, and employing techniques to capture and reproduce the complete data dependencies of compute() functions is beyond the scope of the initial system.

7.5 Related Work

No previous work we know of proposes and implements a debugger designed specifically for the needs of Pregel-like systems, or any other distributed graph-processing system. We review some existing debuggers and explain why they cannot be used or adapted easily for Pregel-like systems.
7.5.1 Remote Debuggers

Remote debuggers in existing development environments, e.g., [39, 66], attach to pieces of code running on a specific port of a specific machine—information that is usually unknown to the users before launching their Giraph programs. Thus, it is difficult or impossible to attach to a specific `vertex.compute()` function that the user is interested in. In addition, remote debuggers are interactive debuggers that halt the execution of the processes that they attach to. Thus, if a user puts a breakpoint into `vertex.compute()` and attaches to a Giraph worker, the worker halts its execution. This behavior may have two side effects. First, because the attached worker is not making progress, the master worker may assume that the attached worker has failed and switch to fault-recovery mode. Second, in Giraph and other Pregel-like systems, a pause of a single worker in a particular superstep causes all other workers to pause as well. Therefore as the user debugs his `vertex.compute()` code, all workers of Giraph are idle and keep consuming resources across the cluster. If there are multiple users of the cluster, those resources will not be available to the other users during the entire debugging process.

7.5.2 Replay Debuggers

Replay debuggers provide offline debugging functionality. As we mentioned in the introduction and Section 7.5 of this chapter, existing replay debuggers, e.g., [6, 48], record low-level information such as all OS calls, network messages, and memory accesses throughout the execution of a distributed application. Then, using this recorded information, the user can replay the execution of any group of processes offline to find bugs in the distributed application. Some replay debuggers are designed to debug any distributed program and thus could be used to debug Giraph programs as well. However, when debugging Giraph programs, existing replay debuggers would incur unnecessarily high overhead, recording a high volume of low-level information, most or all of which is irrelevant for bug-finding inside `vertex.compute()`. Also, existing replay debuggers do not provide specific visualizations for graphs, which we believe is very important for debugging in Pregel-like
systems.

Graft can be thought of as a very lightweight replay debugger tailored for the needs of Giraph users. Instead of replaying the line-by-line executions of different Giraph workers, which existing replay debuggers would do, Graft can replay specific \texttt{vertex.compute()} calls. Thus, Graft only needs to capture a small amount of data, often in the kilobytes, even when debugging a computation across hundreds of workers. In addition, the Graft GUI is tailored to visualize graphs as opposed to generic distributed applications.

7.5.3 Debuggers of Dataflow Systems

Recently, two debuggers \cite{1, 69} and one framework \cite{97} for building debuggers for distributed dataflow systems have been introduced. We briefly review each of these systems, then explain why they could not be adapted easily for Giraph:

- **Inspector Gadget (IG):** IG \cite{97} is a framework for building debugging tools for Pig \cite{98}, a dataflow system on top of Hadoop \cite{9}. Some of the tools that have been programmed by IG allow users to trace input data records across operators to inspect how they are modified, build histograms over the output records of operators, and receive alerts when a records take a long time to process inside operators. The IG framework is not designed to build tools that enable users to do line-by-line debugging within operators.

- **Daphne:** Daphne \cite{69} is a set of debugging tools for DryadLINQ \cite{133}, a dataflow system on top of the Dryad \cite{67} data-processing system. Daphne allows users to visualize and profile the resource consumption of different operators of their workflows. It also allows users to replay and line-by-line debug the processes that crash on their local machines.

- **Arthur:** Arthur \cite{1} is a new debugger for Spark \cite{134} and Hadoop \cite{9} programs. Similar to some IG tools, Arthur lets users trace records across operators. Users can also use a line-by-line debugger to replay any process that they are interested in on their local machines, and users can inspect intermediate outputs of their dataflow programs. When a user issues a query over an
intermediate output, Arthur first re-runs part or all of the dataflow program from the beginning to compute the relevant part of the intermediate output for the query and then runs the query. The query is a Spark program written by the user.

The functionalities provided by these debuggers are not a good fit for the needs of programmers of Pregel-like systems for several reasons. First, users of Pregel-like systems do not cast their graph computations as a workflow of operators that modify a set of input records. Second, \texttt{vertex.compute()} functions can be quite complex, and debugging them necessitates line-by-line debugging of specific \texttt{vertex.compute()} functions that users are interested in. Line-by-line debugging is either not provided by these tools, or is only provided at the worker level, as opposed to vertex level. Finally, similar to replay debuggers, debuggers of dataflow systems do not provide any specific visualizations for understanding graph algorithms.
Chapter 8

Higher-level Primitives For Distributed Graph Algorithms

As introduced in Chapter 5, the APIs of Pregel-like systems expose a `vertex.compute()` function and an optional `master.compute()` function. Other distributed graph processing systems also expose a small set of vertex-centric functions, such as the `gather()`, `apply()`, and `scatter()` functions of PowerGraph [50]. The benefit of these APIs is that programmers concentrate on implementing a few specific functions, and the framework automatically scales the computation by executing these functions in parallel across machines. However, sometimes the functions are too low-level or restricted for the algorithmic task. For example, as we discussed in Chapters 5 and 6, additional code and workarounds that use global data structures can be required to control the flow of some algorithms, yielding long and complex programs. In addition, custom code can be required for some commonly-appearing operations, such as initializing vertex values or checking for convergence of computations. Similar issues with the MapReduce framework have led to widely-used languages such as Pig Latin [98] and Hive [121]. These languages express data analysis tasks using higher-level constructs and primitives, such as filters and grouping, and compile to the lower-level `map()` and `reduce()` functions of the Hadoop open-source MapReduce system. In this chapter, we introduce an analogous approach for graph processing: we describe a set of high-level primitives
we designed and implemented, which capture commonly appearing operations in large-scale graph computations.

In Chapter 6, we described our detailed implementation of several graph algorithms on GPS. In the process of implementation, we observed certain patterns emerge that we believed could be abstracted to a useful set of higher-level graph processing primitives. We implemented an additional suite of algorithms to verify our sense of the most generally useful primitives. This chapter covers the primitives we identified, called HelP (for High-Level Primitives), our implementation of these primitives using the functions available on the GraphX system [130], and the implementation of many graph algorithms using the primitives. Our experience has been that implementing algorithms using our primitives is more intuitive and much faster than using the APIs of existing distributed systems.

We picked the GraphX system, which is built on top of the Spark data processing system [134], to implement our primitives because it required significantly less effort than would have been needed to implement them on top of other existing distributed graph engines and because programming with the HelP primitives in the Spark setting is more natural. Specifically, many graph algorithms are composed of multiple primitives executed one after another and possibly inside a loop. In Spark, programs can be written as a sequence of HelP primitives inside a main function with access to loop constructs. Each HelP primitive translates to a sequence of Spark’s data primitives, which are executed by Spark in the correct order automatically. In contrast, the natural way to implement our primitives on top of Giraph or GPS would be to have a separate compute() function implementing each primitive. However, for algorithms that consist of multiple primitives, we would need to implement either a separate compiler module to merge multiple compute() functions into a single one, or a scheduler to execute a sequence of compute() functions in the correct order.

In the remainder of this introductory material, we give a summary of our primitives and then the outline of the rest of the chapter.
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Update of Vertices (LUV)</td>
<td>Updates vertex values, possibly using global information or local edges of each vertex. Used mainly for initializing vertex values.</td>
</tr>
<tr>
<td>Aggregating Neighbor Values (ANV)</td>
<td>Some vertices aggregate some or all of their neighbor values to update their own values. Appears both as a one-step computation and as an iterative process. In the iterative version, some or all of the vertices start propagating a value to their local neighbors, which are aggregated and propagated further by receiving vertices in the next iteration. The propagations continue until vertex values converge.</td>
</tr>
<tr>
<td>Update Vertices Using One Other Vertex (UVUOV)</td>
<td>Updates vertex values by using a value from one other vertex (not necessarily a neighbor). Commonly used in matching-like algorithms.</td>
</tr>
<tr>
<td>Filter</td>
<td>Removes some vertices or edges from the graph.</td>
</tr>
<tr>
<td>Form Supervertices (FS)</td>
<td>Merges groups of vertices.</td>
</tr>
<tr>
<td>Aggregate Global Value (AGV)</td>
<td>Computes a single global value over the graph. A commonly used special case is picking a random vertex from the graph.</td>
</tr>
</tbody>
</table>

Table 8.1: HelP Primitives.

Overview of HelP Primitives

Table 8.1 summarizes our HelP primitives. We group the primitives broadly into three areas, according to the operations they perform:

1. **Vertex-centric Updates**: The Local Update of Vertices (LUV), Aggregating Neighbor Values (ANV), and Update Vertices Using One Other Vertex (UVUOV) primitives all update the values of some or all of the vertices of the graph, in parallel. An update of a vertex value may use information from edges incident to the vertex (hereafter “local edges”), or from the values of other vertices. Updates can happen in iterations or in a single iteration.

2. **Topology Modifications**: The Filter and Form Supervertices (FS) primitives modify the topology of the graph by removing some vertices or edges based on filtering conditions, and merging multiple vertices together to form new vertices, respectively.
3. Global Aggregations: The Aggregate Global Value (AGV) primitive performs a global aggregation operation over some or all of the graph (e.g., find the average degree, or find the vertex furthest from a given one).

All of the primitives in HelP abstract parallel operations that are suitable for scalable distributed implementations. The HelP primitives and algorithms described in this chapter are fully implemented on top of GraphX and publicly available [61]. We note that some of the primitives in HelP were already part of GraphX; we specify those primitives clearly later in the chapter.

Chapter Outline

- Section 8.1 provides necessary background on GraphX [130] and the Spark system [134].
- In Section 8.2, we describe three primitives and their variations, all of which perform vertex-centric updates: (1) Local Update of Vertices; (2) Aggregating Neighbor Values; and (3) Update Vertices Using One Other Vertex.
- In Section 8.3, we describe the Filter and Form Supervertices primitives and their variations, all of which modify the topology of the graph.
- In Section 8.4, we describe the Aggregate Global Value primitive, which performs a global aggregation operation over the vertices of the graph.
- In Section 8.5, we give examples of algorithms that benefit from using additional “data primitives” such as joins and grouping, either instead of or in addition to our HelP primitives. These examples are naturally expressed in an “edge-centric” fashion (rather than “vertex-centric”), such as triangle-finding, clustering coefficient, or finding k-trusses.
- Section 8.6 covers related work.

Table 8.2 lists the algorithms we have implemented and the primitives that are used in each algorithm. The last column of the table indicates whether any data primitives are used in each algorithm.

Notably absent from this chapter is any experimental evaluation. Evaluating programmer productivity is difficult to do objectively and convincingly. Lines of
<table>
<thead>
<tr>
<th>Name</th>
<th>LUV</th>
<th>ANV</th>
<th>UVUOV</th>
<th>Filter</th>
<th>FS</th>
<th>AGV</th>
<th>Data Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank [87]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HITS [49]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Source Shortest Paths [87]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weakly Connected Components [74]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conductance [23]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semi-clustering [87]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random Bipartite Matching [87]</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Approx. Betweenness Centrality [16]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diameter Estimation (Double Fringe) [104]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly Connected Components (see Chapter 6)</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Minimum Spanning Forest [95] (see Chapter 6)</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Approx. Maximum Weight Matching (see Chapter 6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Graph Coloring (see Chapter 6)</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximal Independent Set (see Chapter 6)</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-core [104]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Triangle Finding [104]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Clustering Coefficient [104]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>K-truss [104]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Simple METIS [91]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Multilevel clustering [96]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Table 8.2: Implemented Algorithms.
code (LOC) is one possible metric to evaluate productivity gains of new APIs and languages, however the shortcomings of LOC as a measure have long been observed [106]. For the algorithms in Table 8.2, we saw up to 2x code reduction using our primitives against coding in GraphX without them. We could also evaluate the performance of algorithms implemented using our primitives against using other libraries on top of GraphX, or programming in GraphX directly. However, all of these approaches, when programmed carefully, translate to similar GraphX and Spark calls at the lowest levels, and thus similar expected performance. Finally, evaluating HelP against other graph processing systems would yield the same comparison as those systems against GraphX, as covered in the original GraphX paper [130].

8.1 Background

We review the Spark computation engine and the GraphX system built on top of Spark.

8.1.1 Spark

Spark [114, 134] is a general open-source distributed system for large-scale data processing. Spark was developed initially at UC Berkeley, and later became an Apache project. Spark’s programming model consists of two parts: (1) an in-memory data abstraction called resilient distributed datasets (RDDs); and (2) a set of deterministic parallel operations on RDDs that can be invoked using data primitives, such as map, filter, join, and groupBy. RDDs can either be created from data residing in external storage, such as a distributed file system, or can be derived from existing RDDs by applying one of Spark’s data primitives. For example, the map primitive transforms each element $t$ in an RDD by applying a user-specified mapF function to $t$, and returns a new RDD. RDDs are partitioned across a cluster of machines and each data primitive on RDDs executes in parallel across these machines. Spark is implemented in Scala and RDD elements can
contain arbitrary Java objects. A Spark program is a directed acyclic graph of Spark’s primitives, which are applied on a set of initial and derived RDDs. Here is a simple example program from the original Spark paper [134], which counts the number of lines that contain errors in a large log file:

```scala
val file = spark.textFile("hdfs://... ")
val errs = file .filter (_.contains("ERROR"))
val ones = errs.map(_ => 1)
val count = ones.reduce(AggrFn.SUM)
```

Values `file`, `errs`, and `ones` in the above code are RDDs and `count` is an integer numeric value. Spark stores the lineage of RDDs, i.e., the chain of operations that were applied to construct an RDD. In case of failures, Spark uses the lineage information to rebuild lost partitions by performing each operation in the lineage consecutively. Programmers can perform iterative computations very easily on Spark by using Scala’s loop constructs, such as `for` and `while`. In addition to iterative computations, Spark supports interactive computations through the Spark interpreter.

### 8.1.2 GraphX

GraphX is a distributed graph engine built on top of Spark. GraphX stores a graph in RDDs and supports a set of operations on the graph that can be invoked using GraphX’s core graph primitives. In addition, GraphX contains Pregel and PowerGraph libraries, which are built on top of the core graph primitives. These libraries implement the Pregel and PowerGraph APIs, which programmers can use to implement graph algorithms. Our HelP primitives are built as another library on top of the core GraphX primitives. Figure [8.1] shows the hierarchy of primitives belonging to Spark, GraphX, and the libraries on top of GraphX. To set the stage for explaining our library and how it was implemented, we next review the RDDs GraphX uses to store a graph, and some of the core graph primitives in GraphX.
GraphX RDDs
GraphX contains three core RDDs to store a graph:

- **EdgesRDD**: Stores the edges of the graph along with edge values. EdgesRDD is partitioned evenly across machines.
- **VerticesRDD**: Stores the vertices of the graph along with vertex values. VerticesRDD is also partitioned evenly across the machines. Note that VerticesRDD does not contain information about the edges incident on the vertices.
- **RoutingTableRDD**: Stores for each vertex \( v \) the edge partitions that contain the edges where \( v \) appears as a source or a destination vertex.

Core GraphX Primitives

We review the core GraphX primitives that we used for implementing our high-level graph primitives. A comprehensive list of all core GraphX primitives appears in [56].

- **filter**: Takes a vertex predicate \( \nuP \) and an edge predicate \( \epsilonP \). Returns a new graph that contains each vertex \( v \) for which \( \nuP \) evaluates to true, and
each edge \( e \) of the original graph if: (1) \( e \vDash P \) evaluates to true on \( e \); and (2) \( \forall v \vDash P \) evaluates to true on both the source and destination vertex of \( e \).

- **mapVertices**: Takes as input a \( \mathit{mapF} \) function and applies it to each vertex \( v \) of the graph to generate a new value for \( v \). The edges of the graph remain unchanged.

- **collectEdges**: Takes as input an edge direction, one of \( \text{In} \), \( \text{Out} \), or \( \text{Either} \), and returns a new RDD that contains, for each vertex \( v \) in the graph, a tuple with \( v \)'s ID, data, and edges in the user-specified direction. The returned RDD is called \( \text{ExtendedVerticesRDD} \).

- **getTriplets**: Returns an RDD called \( \text{TripletsRDD} \) that contains an edge triplet of the form \( (\mathit{eval}, \mathit{srcObj}, \mathit{dstObj}) \) for each edge \( e=(u,v) \). \( \mathit{eval} \) is \( e \)'s value, while \( \mathit{srcObj} \) and \( \mathit{dstObj} \) contain the ID and value of the source \( u \) and destination \( v \), respectively.

- **mapReduceTriplets**: Takes three inputs:
  - \( \mathit{tripletMapF} \): A function whose input is an extended edge triplet: an edge triplet as above but with an additional field on the \( \mathit{srcObj} \) and \( \mathit{dstObj} \) indicating whether \( u \) and \( v \) are in the user-specified \( \text{optVertices} \) set (see below). The output of \( \mathit{tripletMapF} \) is a list of “messages” of the form \( (\mathit{vertexID}, \mathit{msg}) \). The first field is the ID of a vertex, and the second field is the message, which can be any Java object.
  - \( \mathit{reduceMessagesF} \): A function that takes as input a list of messages that were “sent” to the same vertex by \( \mathit{tripletMapF} \), and returns an aggregated message of the same type.
  - \( \text{optVertices} \): A possibly empty set of vertex IDs. If the ID of a vertex \( v \) is in \( \text{optVertices} \), this information is exposed in the extended edge triplet input to \( \mathit{tripletMapF} \).

\( \text{mapReduceTriplets} \) is generally used in graph operations that are naturally expressed by vertices sending and receiving messages to and from their neighbors. For each edge \( e=(u,v) \) in the graph, \( \text{mapReduceTriplets} \) applies \( \mathit{tripletMapF} \) to \( e \)'s extended edge triplet and generates a message to \( u \), \( v \), or
both u and v. Then reduceMessagesF is applied to the list of messages sent to each vertex v in the graph to generate a single aggregated message for v. The output of mapReduceTriplets is an RDD called MessagesRDD, that contains for each vertex v that was sent at least one message, one tuple of the form (vertexID, aggregatedMsg). We use mapReduceTriplets in our implementation of the propagateAndAggregate primitive (Section 8.2.2).

- **diff**: Unlike the previous primitives, which operate on the graph, diff operates on two VerticesRDDs, old and new. Given old and new, diff returns a new VerticesRDD that contains each vertex v in old that exists and contains a different value in new, along with v’s new value.

### 8.2 Vertex-Centric Updates

In this section and the following two we present our HelP primitives. For each primitive we first give a high-level description of the primitive, then we give one or more examples from our algorithms that use the primitive. Finally, we explain how the primitive is implemented in GraphX. We note that except for global aggregations (Section 8.4), all of our primitives return a new graph. In our code snippets, we omit the assignment of the returned graph to a new variable.

#### 8.2.1 Local Update of Vertices

One of the most commonly appearing graph operations is to initialize the values of some or all of the vertices, either at the beginning of an algorithm, or at the start of each phase of an algorithm. Depending on whether the local edges of the vertices are used in the updating of vertices, we provide two primitives for this operation: updateVertices and updateVerticesUsingLocalEdges.

- **updateVertices** takes two inputs:
  - vP: A predicate to select which vertices to update.
  - updateF: A function that takes a vertex and returns a new value for the vertex.
Not surprisingly, the behavior of `updateVertices` is to update the values of all vertices for which \( vP \) evaluates to true, with the updated vertex value returned by `updateF`. Other vertices remain unchanged. `updateVerticesUsingLocalEdges` takes an additional edge direction input `dir` which specifies whether the incoming, outgoing, or both types of incident edges are used in `updateF`. The programmer can access the specified edges in a list format inside the `updateF` function.

### Examples of Use

Consider the first step of PageRank [25], which initializes the value of each vertex to \( \frac{1.0}{|V|} \). Using `updateVertices`, we can express this operation as follows:

```scala
val updateFn = v => {
  v.val.pageRank = 1.0 / g.numVertices;
  v
}

val updatedVertices = g.updateVertices(v => true, updateFn)
```

As an example use of `updateVerticesUsingLocalEdges`, consider the bipartite matching algorithm from [87]. The vertices of the input bipartite graph are divided into \( L \) and \( R \), for left and right, respectively. In each iteration of the algorithm, every unmatched vertex \( v_l \) in \( L \) randomly picks one of its neighbors from \( R \), say \( v_r \), and stores the ID of \( v_r \) in the `pickedNbr` field of its value. We can express this operation using `updateVerticesUsingLocalEdges` as follows:

```scala
val updateFn = (v, edges) => {
  v.val.pickedNbr = edges.get(random(edges.length)).dstID;
  v
}

val updatedVertices = g.updateVerticesUsingLocalEdges(EdgeDirection.Out, v => v.isLeft, updateFn)
```

### Implementation

GraphX’s core primitive `mapVertices` (Section 8.1.2 above) already contains the functionality of `updateVertices`, but we added a new primitive for convenience, since `mapVertices` does not take predicate `vP` as an explicit input. We implement `updateVertices` simply by pushing predicate `vP` into the `updateF` function of `mapVertices`. 
CHAPTER 8. HELP

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>The direction of the local neighbors whose values will be aggregated</td>
</tr>
<tr>
<td>nbrP</td>
<td>A predicate to select which neighbors to aggregate</td>
</tr>
<tr>
<td>vP</td>
<td>A predicate to select which vertices to update</td>
</tr>
<tr>
<td>aggregated-ValueF</td>
<td>A function that takes a neighbor vertex value and returns the relevant part to be aggregated, possibly the entire neighbor value itself</td>
</tr>
<tr>
<td>aggregateF</td>
<td>The aggregation function</td>
</tr>
<tr>
<td>updateF</td>
<td>A function that takes a vertex and an aggregated value of the neighbors and returns a new value for the vertex</td>
</tr>
</tbody>
</table>

| Table 8.3: Inputs to aggregateNeighborValues. |

Implementation of updateVerticesUsingLocalEdges is a bit more complex. We first call GraphX's collectEdges primitive on the graph with the user-specified edge direction. Recall from Section 8.1.2 that collectEdges returns ExtendedVerticesRDD, an extension of the VerticesRDD containing the local edges of each vertex in the specified direction. Similar to updateVertices, we push predicate vP inside updateF, and we call Spark's map primitive with updateF on ExtendedVerticesRDD to generate a new VerticesRDD.

8.2.2 Aggregating Neighbor Values

In another common form of vertex-centric update operation, vertices aggregate some or all of their neighbors' values to update their own values. PageRank, HITS, finding shortest paths from a single source, finding weakly-connected components, or computing the conductance of a graph, are some of the example algorithms that perform this operation. Aggregating neighbor values appears in algorithms as a one-step computation, or as an iterative process that continues until vertex values converge. We provide the aggregateNeighborValues primitive for the one-step version, and propagateAndAggregate for the iterative version.

aggregateNeighborValues: The inputs to aggregateNeighborValues are listed in Table 8.3. vP is a predicate that selects which vertices to update. If v is a vertex whose value should be updated, the input dir and the nbrP predicate determine the neighbors of v whose values should be aggregated in updating the
<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>The direction of the local neighbors to propagate values to</td>
</tr>
<tr>
<td>startVP</td>
<td>A predicate to select which vertices to start propagating from</td>
</tr>
<tr>
<td>propagated-ValueF</td>
<td>A function that takes a vertex value and returns the relevant part to be propagated</td>
</tr>
<tr>
<td>propagate-AlongEdgeF</td>
<td>A function that takes two inputs: (1) the propagated value of a vertex; and (2) an edge value, through which the vertex will propagate its value to a neighbor; and computes the final propagated value along the edge</td>
</tr>
<tr>
<td>aggregateF</td>
<td>A function to aggregate propagated values</td>
</tr>
<tr>
<td>updateF</td>
<td>A function that takes a vertex and an aggregated value of the propagated values from neighbors and returns a new value for the vertex</td>
</tr>
</tbody>
</table>

Table 8.4: Inputs to propagateAndAggregate.

value of $v$. Function aggregatedValueF is applied to the values of these neighbors, the outputs of aggregatedValueF are aggregated using aggregateF, and finally updateF is applied on $v$ and the output of aggregateF to compute the new value for $v$.

**propagateAndAggregate**: In some computations the aggregation of neighbor values continues in iterations until all vertex values converge. The common pattern of such computations is the following: In the first iteration, one or more vertices propagate a value to their neighbors. Vertices that receive propagated values aggregate them and update their own values. In the next iteration, all vertices whose values have changed propagate a new value to their neighbors. The propagation of values continues in iterations until all vertex values are stable.

The inputs of propagateAndAggregate are listed in Table 8.4. Inputs dir, aggregateF, and updateF are the same as in aggregateNeighborValues. The startVP predicate selects which vertices propagate values in the first iteration. (In the weakly-connected components example, which we will discuss in Section 8.2.2, this predicate always returns true. However, in other algorithms, such as single source shortest paths [87] or approximate betweenness centrality [16],...
this predicate returns true only for a subset of the vertices.) Similar to the aggregatedValueF input to aggregateNeighborValues, propagatedValueF extracts the relevant value to be propagated from a vertex. In addition, there is a propagateAlongEdgeF function that takes a propagated value val from a vertex, and an edge through which val will be propagated, and computes a possibly modified value to be propagated along the edge.

Examples of Use

As an example use of aggregateNeighborValues, consider executing a fixed number of iterations, 10 say, of the HITS algorithm [79] for ranking web-pages. In HITS, each vertex has a hub and an authority value. In one iteration of the algorithm, vertices first compute the sum of all of their incoming neighbors’ hub values to update their authority values. Then, vertices aggregate their outgoing neighbors’ authority values to update their hub values. We can express this computation using aggregateNeighborValues as follows:

```c
for (i = 0; i < 10; ++i) {
    // Aggregate in−neighbors’ hub values to update authorities
    g.aggregateNeighborValues(
        EdgeDirection.In, /* direction of neighbors */,
        nbr → true /* which neighbors to aggregate */,
        v → true /* which vertices to update */,
        nbrVal → nbrVal.hub /* relevant neighbor value to aggregate */,
        AggrFnc.SUM /* aggregate neighbors’ hub values by summation */,
        (v, aggrHubVal) → { v.val.authority = aggrHubVal; v;})

    // Aggregate out−neighbors’ authorities to update hub values
    g.aggregateNeighborValues(
        EdgeDirection.Out,
        nbr → true,
        v → true,
        nbrValue → nbrValue.authority,
```
As an example use of propagateAndAggregate, consider the weakly-connected components algorithm WCC from reference [74], which we covered in Chapter 6. The input is an undirected graph, and vertices store a \( wccID \) value, initialized to their own IDs. Initially each vertex \( v \) propagates its \( wccID \) to all of its neighbors. In iterations, each vertex \( v \) updates its \( wccID \) to the maximum of its own \( wccID \) and the \( wccID \) values propagated by its neighbors, then propagates its \( wccID \) further if it has changed. At convergence, all vertices with the same \( wccID \) value belong to the same connected component. We can express this computation using propagateAndAggregate as follows:

\[
g.\text{updateVertices}(v \rightarrow \text{true}, v \rightarrow \{ v.\text{val}.wccID = v.\text{ID} \})
\]

\[
g.\text{propagateAndAggregate}(\text{EdgeDirection.Either} /* \text{direction of neighbors} */ ,
  v \rightarrow \text{true} /* \text{which vertices to start propagating from} */ ,
  vVal \rightarrow vVal.wccID /* \text{propagated value} */ ,
  /* \text{do not change propagated value along each edge} */
  (\text{propagatedVal}, \text{edgeVal}) \rightarrow \text{propagatedVal} ,
  \text{AggrFnc.MAX} /* \text{aggregate wccIDs by taking their max} */ ,
  (v, \text{aggrWccIDVal}) \rightarrow \{ 
  v.\text{val}.wccID = \text{AggrFnc.MAX}(v.\text{val}.wccID, \text{aggrWccIDVal}); v; \})
\]

**Implementation**

We explain our implementation of aggregateNeighborValues for aggregating in-neighbors of vertices, i.e., input dir is In. Implementations of the other directions are similar. The implementation consists of three steps:

1. Call GraphX’s mapReduceTriplets primitive. Recall from Section 8.1.2 that mapReduceTriplets applies a tripletMapF function to the extended edge triplet \((\text{eval}, \text{srcObj}, \text{dstObj})\) associated with each edge \((u,v)\) in the
graph and generates a list of messages. The list of messages for each vertex is then aggregated by the `reduceMessagesF` function to compute a final `MessagesRDD`. For `tripletMapF` we use a function that sends a message to the destination vertex, if the source vertex satisfies the `nbrP` predicate and the destination vertex satisfies the `vP` predicate. The message contains the output of applying `aggregatedValueF` on the source vertex’s value. For `reduceMessagesF` we use `aggregateF`. It returns a `MessagesRDD` that contains, for each vertex `v` that satisfies `vP`, the aggregation of the values of `v`’s in-neighbors.

2. Join the `MessagesRDD` with `VerticesRDD` to obtain `VerticesMessagesRDD`, which contains for each vertex `v` its aggregated message `m`.

3. Apply Spark’s `map` primitive on `VerticesMessagesRDD` using `updateF`, which takes `v` and its aggregated message `m` and generates a new value for `v`. Finally, return a new graph using `v`’s new value.

Figure 8.2 shows the implementation of `propagateAndAggregate` pictorially on an example. The implementation consists of four steps performed in iterations. The first three steps of each iteration are similar to the three steps of `aggregateNeighborValues` as above, but with a few differences. We again explain our implementation for the case when the input `dir` is `In`.

1. Call `mapReduceTriplets` as in `aggregateNeighborValues`. However, now we send a message from the source `u` to `v` only if `u` satisfies the `startVP` predicate. Moreover, we apply `propagateAlongEdgeF` to the output of `propagatedValueF` and the edge value `eval` in the extended edge triplet to possibly generate a new message.

2. Join the `MessagesRDD` with `VerticesRDD` to obtain `VerticesMessagesRDD`.

3. Apply Spark’s `map` primitive on `VerticesMessagesRDD` using `updateF` to generate a new `VerticesRDD`.

4. Call GraphX’s `diff` primitive (Section 8.1.2) on the old and new `VerticesRDD`. We refer to the output of `diff` as `ChangedVerticesRDD`. 
In iteration $i$, send a message from $src$ to $dst$ if: (1) $i=1$ and $startVP(src) = true$; or (2) $i>1$ and $src$ is in ChangedVerticesRDD. Messages are computed by $propagateAlongEdgeF(edgeVal, propagatedValueF(src))$, and aggregated by $aggregateF$.

In subsequent iterations, we repeat the three steps of the first iteration with two differences: (1) we pass ChangedVertices to $mapReduceTriplets$ as the $optVertices$ input (see Section 8.1.2); and (2) instead of sending a message from $u$ to $v$ if $u$ satisfies $startVP$, we send a message if $u$ is in ChangedVertices. Recall that $mapReduceTriplets$ exposes this information in a field of the $srcObj$ when $optVertices$ input is specified. We continue the iterations until ChangedVertices is empty.

### 8.2.3 Updating Vertices Using One Other Vertex

In some algorithms, vertices store a pointer to (actually an ID of) one other vertex, not necessarily a neighbor, in a field of their vertex value, and either update their
CHAPTER 8. HELP

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vP</td>
<td>A predicate to select which vertices will update themselves</td>
</tr>
<tr>
<td>otherVertexIDF</td>
<td>A function that takes the vertex value, say of v, and returns the ID of the other vertex that v points to</td>
</tr>
<tr>
<td>relevantPointedVertexValueF</td>
<td>A function that takes the pointed vertex’s value and returns the relevant part of it that will be used in updating the pointer vertex</td>
</tr>
<tr>
<td>updateF</td>
<td>Takes the vertex v and the relevant value of the vertex that v points to and returns a new value for v</td>
</tr>
</tbody>
</table>

Table 8.5: Inputs to updateSelfUsingOneOtherVertex.

own value using the value of the vertex they point to, or vice-versa. This operation appears commonly in matching algorithms, but it also appears in Boruvka’s minimum spanning forest algorithm [95] (covered in Chapter 6), the METIS graph partitioning algorithm [108], and the clustering algorithm from [96]. If v stores the ID of w in its value, we refer to v as the pointer vertex and w as the pointed vertex. Depending on whether the pointer or the pointed vertex is updated, we provide two primitives for this operation: updateSelfUsingOneOtherVertex and updateOneOtherVertexUsingSelf. The inputs and behavior of the two primitives are very similar, so we specify them only for updateSelfUsingOneOtherVertex.

The inputs of updateSelfUsingOneOtherVertex are listed in Table 8.5. The behavior of updateSelfUsingOneOtherVertex is to update each vertex v for which vP evaluates to true in three steps: (1) compute the ID of the vertex w that v points to by applying otherVertexIDF on v; (2) apply relevantPointedVertexValueF to w to extract the relevant value of w that will be used in updating v; and (3) apply updateF on v and the output of relevantPointedVertexValueF from the second step to compute the new value for v. We note that in updateOneOtherVertexUsingSelf, multiple vertices might point to and update the same vertex. As a result updateOneOtherVertexUsingSelf takes an additional input function aggregateF, which aggregates the relevant pointer vertex values, before applying updateF.
Examples of Use

Consider Boruvka’s minimum spanning forest (MSF) algorithm that we covered in Chapter 6. In each iteration of this algorithm, each vertex \( v \) points to its minimum-weight neighbor, which is stored in a `pointedVertex` field. As shown in [33], the vertices and their picked neighbors form disjoint subgraphs called *conjoined-trees*: two trees joined by a cycle. Figure 8.3 shows an example conjoined tree. We refer to the vertex with the smaller ID in the cycle of a conjoined tree \( T \) as the *root* of \( T \), for example vertex 5 in Figure 8.3. After picking their neighbors, vertices find the root of the conjoined-tree they are part of iteratively as follows. In the first iteration, each vertex \( v \) discovers whether it is the root by checking whether \( v \)'s `pointedVertex` \( u \) points back to \( v \), and whether \( v \)'s ID is smaller than \( u \). If both conditions hold, \( v \) is the root and sets its `pointsAtRoot` field to true. In the later iterations, each vertex \( v \) copies its `pointedVertex` \( u \)'s `pointedVertex` and `pointsAtRoot` values to itself until every vertex points to the root. We can express this computation using `updateSelfUsingOneOtherVertex` as follows (we will describe the `aggregateGlobalValue` primitive in Section 8.3):

```c
// discover the root of each conjoined-tree
g.updateSelfUsingOneOtherVertex(
    v → true, /* which vertices to update */
    v → v.val.pointedVertex /* ID of the other vertex */,
```

Figure 8.3: Example of a conjoined-tree.
// relevant part of the other vertex
otherV \rightarrow (otherV.val.pointedVertex, otherV.ID),

// updateF: set pointsAtRoot to true if otherV also points
// at v and v's ID is smaller than otherV's ID
(v, (otherVsPointedVertex, otherVID) \rightarrow 
\begin{cases}
  \text{true} & \text{if } (v.ID == otherVsPointedVertex \text{ and } v.ID < otherVID) \\
  \text{false} & \text{otherwise}
\end{cases}
\}
)

// count the number of vertices that do not point to a root
var numNotPointing = g.aggregateGlobalValue(
  v \rightarrow \begin{cases}
    0 & \text{if } v.val.pointsAtRoot \\
    1 & \text{otherwise}
  \end{cases}, \text{AggrFnc.SUM}
)
while (numNotPointing > 0) 
{
g.updateSelfUsingOneOtherVertex(
  v \rightarrow \neg v.val.pointsAtRoot,
  v \rightarrow v.val.pointedVertex,
  otherV \rightarrow (otherv.val.pointedVertex, otherv.pointsAtRoot),
  (v, (otherVsPointedVertex, otherVPointsAtRoot) \rightarrow 
\begin{cases}
  otherVsPointedVertex & \text{if } otherVPointsAtRoot \text{ is true}
  \end{cases}
\)
numNotPointing = g.aggregateGlobalValue(
  v \rightarrow \begin{cases}
    0 & \text{if } v.val.pointsAtRoot \\
    1 & \text{otherwise}
  \end{cases}, \text{AggrFnc.SUM}
)
}

---

Implementation

Figure 8.4 shows the implementation of updateSelfUsingOneOtherVertex pictorially on an example, which we also use to guide our explanation. (We omit the description of updateAnotherVertexUsingSelf, which is similar.) In the figure, vertex values consist of two fields, the first of which stores the ID of the pointed vertex, e.g., \(v_1\) points to \(v_3\), and \(v_4\) points to \(v_1\). In our example, if \(u\) points to \(v\), \(u\) copies over the second field of \(v\) to itself.

1. We apply otherVertexIDF on each vertex by calling Spark’s map primitive
on VerticesRDD, generating an intermediate RDD containing (pointedID, pointerID) tuples. Since $v_1$ and $v_2$ point at $v_3$ in the figure, two of the tuples in this RDD are $(v_3.ID, v_1.ID)$ and $(v_3.ID, v_2.ID)$. We then group these tuples by pointedID to obtain PointingRDD, which in our example contains $(v_3.ID, \{v_1.ID, v_2.ID\})$.

We join each pointed vertex $v$'s value with the IDs of the vertices that point to $v$, by joining PointingRDD with the original VerticesRDD. We obtain in our example $(v_3.ID, (v_4.ID, c), \{v_1.ID, v_2.ID\})$, where the second field is $v_3$'s value. We then use Spark's flatMap primitive\footnote{flatMap is a generalized version of the map primitive that can output multiple elements instead of just one.} to output a set of

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure8_4.png}
\caption{Implementation of $updateSelfUsingOneOtherVertex$.
}\end{figure}
(pointerID, relevantPointedVValue) tuples. In our example, we output \((v_1.\text{ID}, c)\) and \((v_2.\text{ID}, c)\), where \(c\) is the relevant value from \(v_3\). We call this RDD RelevantValuesRDD.

3. We join each vertex \(v\)'s value with the relevant value from the vertex \(v\) points to (by joining RelevantValuesRDD with the original VerticesRDD). In our example, we obtain \((v_1.\text{ID}, (v_3.\text{ID}, a), c)\) and \((v_2.\text{ID}, (v_3.\text{ID}, b), c)\). Finally, using Spark's map primitive, we apply \text{updateF} on the join tuples to obtain new values for each vertex. We get \((v_1.\text{ID}, (v_3.\text{ID}, c))\) and \((v_1.\text{ID}, (v_3.\text{ID}, c))\), effectively copying over the \(c\) from \(v_3\)'s value to \(v_1\) and \(v_2\).

### 8.3 Topology Modifications

We describe two primitives that change the topology of the graph, either by removing certain vertices and edges, or by merging multiple vertices together to form supervertices (explained below).

#### 8.3.1 Filtering

Filtering operations remove certain vertices and/or edges from the graph. These operations appear especially in algorithms that iteratively find partial solutions on a subset of the vertices or edges of the graph, remove them, then continue until there are no vertices and edges left. We provide two primitives for filtering vertices and one primitive for filtering edges.

- **filterVertices**: Takes as input a predicate \(vP\) that accepts a vertex value and an ID. The behavior is to remove all vertices from the graph for which \(vP\) evaluates to false and all edges that are incident on such vertices.

- **filterVerticesUsingLocalEdges**: Same as **filterVertices** except predicate \(vP\) takes as additional input the local edges of the vertex in a user-specified direction \(dir\).

- **filterEdges**: Takes as input a predicate \(eP\) that accepts an edge value and a source and destination vertex ID and values. The behavior is to remove from
the graph all edges for which $eP$ evaluates to false.

**Examples of Use**

Consider the approximate maximum weight matching algorithm from [103]. In each iteration of this algorithm, vertices that are unmatched pick their maximum-weight edge neighbor and store it in a pickedNbrID field. Then, if two vertices have picked each other, i.e., they have successfully matched, they set their isMatched field to true and are removed from the graph. We can express the removal of matched vertices from the graph using the `filterVertices` primitive simply as:

$$g.\text{filterVertices} \ (v \rightarrow v.\text{val.isMatched} == \text{false})$$

For an example use of `filterEdges`, consider Luby’s maximal independent set algorithm [85], which we used as a subroutine in the graph coloring algorithm GC in Chapter 6. Recall that in Luby’s algorithm each vertex sets its type value as InSet, NotInSet, or Unknown. In each iteration, some of the Unknown vertices are marked as InSet, and some as NotInSet, while the types of others remain unchanged. At the end of each iteration, the algorithm removes all edges incident on InSet and NotInSet vertices, i.e., keeps only the edges between Unknown vertices. This operation is expressed using `filterEdges` as:

$$g.\text{filterEdges} \ ((\text{edgeValue}, \text{srcVertex}, \text{dstVertex}) \rightarrow \text{(srcVertex.value.setType == Unknown)} \ (\text{dstVertex.value.setType == Unknown}))$$

**Implementation**

GraphX’s core primitive `filter` can already perform the functionalities of `filterVertices` and `filterEdges`, since `filter` takes both an edge and vertex predicate (Section 8.1.2). We nonetheless added new `filterVertices` and `filterEdges` primitives for convenience and clarity: in all of our algorithms either vertices or edges are filtered, never both. `filterVertices` simply calls `filter` with $vP$ and a dummy edge predicate that always evaluates to true; similarly for `filterEdges` with $eP$ and a dummy vertex predicate.
For filterVerticesUsingLocalEdges, we first call GraphX’s collect-Edges primitive with the user-specified direction to obtain an extended VertexRDD that contains a list of edges for each vertex. Then we filter this extended VertexRDD with the given vP to compute a new VerticesRDD.

8.3.2 Forming Supervertices

Another operation that modifies the topology of the graph is to merge groups of vertices into supervertices. Recall from Chapter [6] that this operation appeared in Boruvka’s MSF algorithm. This operation also appears in some graph partitioning and clustering algorithms, such as the METIS partitioning algorithm [91, 108] and the clustering algorithm from reference [96]. In these algorithms, after some computation every vertex identifies a supervertex (possibly itself) that it will merge into. Then:

- All vertices and their values that belong to the same supervertex are merged into a single vertex. How the vertex values are merged is specified by an input function.
- Consider an edge \((u,v)\) and assume that vertices \(u\) and \(v\) are merged into supervertices \(s_1\) and \(s_2\), respectively. If \(s_1 = s_2\), then \((u,v)\) is removed from the graph. Otherwise, \((u,v)\) becomes an edge between \(s_1\) and \(s_2\). If there are multiple edges between \(s_1\) and \(s_2\), then edges are merged. How the edge values are merged is specified by an input function.

The inputs of our formSupervertices primitive are listed in Table [8.6]

Examples of Use

As an example, consider Boruvka’s MSF algorithm, which we discussed in Section [8.2.3]. In each iteration, vertices discover the root of the conjoined-tree they are part of and store its ID in a pointedVertex value. The algorithm uses these values to merge all vertices in each conjoined-tree into a single supervertex. For this algorithm, function mergeVertexValuesF simply returns a new empty vertex value because the algorithm does not need to merge the values of vertices that
<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>supervertexIDF</td>
<td>A function that takes the vertex value, say of v, and returns the ID of v's supervertex</td>
</tr>
<tr>
<td>mergeVertexValuesF</td>
<td>A function that takes a set of vertex values and returns a single merged value for the supervertex</td>
</tr>
<tr>
<td>mergeEdgeValuesF</td>
<td>A function that takes a set of edge values for edges between the same pair of supervertices and returns a single merged value for the edge</td>
</tr>
</tbody>
</table>

Table 8.6: Inputs to formSupervertices.

form the supervertex. However, edges are weighted; function mergeEdgeValuesF takes the minimum of the edge values. We can express this computation using formSupervertices as:

\[
g.\text{formSupervertices}(\text{v} \rightarrow \text{v.val.pointedVertex} \quad \text{/** ID of the supervertex */} \\
\text{vVals} \rightarrow \text{new EmptyMSTVertexValue()} \quad \text{/** mergeVertexF */} \\
\text{eVals} \rightarrow \text{AggrFnc.MIN(eVals)} \quad \text{/** mergeEdgeF */})
\]

Implementation

The implementation of formSupervertices generates a new graph from scratch by first computing a new VerticesRDD and then a new EdgesRDD, as follows.

- We map each vertex in the old VerticesRDD to a (supervertexID, vertexValue) tuple. We extract the supervertex ID of each vertex by calling supervertexIDF on the vertex. We then use Spark’s reduceByKey primitive to group the (supervertexID, vertexValue) tuples by supervertexID and aggregate them using mergeVertexValuesF, which yields a new VerticesRDD.
- For computing the new EdgesRDD we use GraphX’s getTriplets primitive to get the TripletsRDD. Recall from Section 8.1.2 that TripletsRDD associates each edge e=(u,v) with the IDs and values of u and v. We then
apply Spark’s map primitive on TripletsRDD. If \( u \) and \( v \) belong to different supervertices, map outputs a tuple of the form \(((\text{newSrcID}, \text{newDstID}), \text{eval})\), where newSrcID and newDstID are the supervertex IDs of \( u \) and \( v \), respectively, and eval is the value of \( e \). Otherwise, if \( u \) and \( v \) belong to the same supervertex, map does not output any tuples. Finally, we use Spark’s reduceByKey primitive to group the resulting tuples by \((\text{newSrcID}, \text{newDstID})\) pairs and aggregate the associated edge values of each pair using mergeEdgeValuesF, which yields a new EdgesRDD.

8.4 Global Aggregations

Many graph algorithms need to compute a global value over the vertices of the graph, such as counting the number of vertices with a particular value, or finding the maximum vertex value. Specifically, each vertex emits a value, and the values are aggregated in some fashion to produce a single value. These computations can thus be seen as special MapReduce computations with a single reducer.

We abstract global aggregation in our aggregateGlobalValue and aggregateGlobalValueUsingLocalEdges primitives. aggregateGlobalValue takes two inputs:

- mapF: A function that takes a vertex and produces a value.
- reduceF: A function that takes a pair of values and aggregates them into a single value.

Notice that function reduceF combines a pair of values, rather than the set of all emitted values. Function reduceF is first applied to a pair of mapped values to produce a single value, the function is then applied to the result value with another mapped value, and so on until the entire set of mapped values has been processed to produce the final aggregated value. The order of application is unpredictable, so for correctness of the aggregation operation, reduceF should be commutative and associative. Our requirement of a pairwise reduceF function is primarily for efficiency: mapF is applied to every vertex in the graph, so the set of values to be processed by reduceF has size equal to the number of vertices of
the graph. Applying reduceF to the entire set at once would require collecting all of the values in a single machine. Also, the most efficient implementation of global aggregation in GraphX is to use Spark’s map and reduce primitives directly (Section 8.4.2), and Spark likewise limits its reduce primitive to pairwise operations for efficiency [116].

Function aggregateGlobalValueUsingLocalEdges takes an additional dir input, and in this case mapF additionally takes the set of edges incident to the vertex in the specified direction.

We note that a frequently used special case of a global aggregation operation is to pick a random vertex from the graph, which we expose as a separate pickRandomVertex primitive in GraphX.

8.4.1 Examples of Use

One example use of aggregateGlobalValue is to detect termination of the root-finding phase of Boruvka’s MSF algorithm. Recall that the algorithm iterates in this phase until all vertices find their roots. To compute whether all vertices have found their roots, the algorithm counts the number of vertices whose pointsAtRoot value is false, using primitive aggregateGlobalValue as follows:

\[ \text{numNotPointing} = g.\text{aggregateGlobalValue}(v \rightarrow \{ \text{(!v.val.pointsAtRoot) ? 1 : 0} \}, \text{AggrFnc.SUM}) \]

Another example is the approximate betweenness-centrality algorithm from reference [16], which performs a breadth-first search (BFS) from a source vertex and labels each vertex with its level in the BFS tree. Then, the algorithm computes the maximum depth of the tree using primitive aggregateGlobalValue as follows:

\[ \text{maxDepth} = g.\text{aggregateGlobalValue}(v \rightarrow v.\text{val.level}, \text{AggrFnc.MAX}) \]

8.4.2 Implementation

The implementation of aggregateGlobalValue simply applies Spark’s map and reduce primitives on the VerticesRDD with mapF and reduceF respectively.
For aggregateGlobalValueUsingLocalEdges, we call GraphX’s collectNeighbors with the user-specified direction before applying flatMap and reduce.

8.5 Supplementing Graph Primitives with Data Primitives

In Sections 8.2–8.4 we described our HelP primitives that abstract several commonly-appearing operations in distributed graph computations. However, for certain algorithms, our primitives alone may not be sufficient or the most suitable way of implementing the algorithm. Sometimes, using record-oriented data primitives instead of or in addition to our graph primitives can be beneficial. In the next two subsections, we show example algorithms based on finding subgraphs, such as triangles, rectangles, or k-trusses (defined momentarily), that benefit from this approach.

As observed in the original GraphX paper [130], integration of graph and data primitives enables programmers to express the three components of a typical large-scale graph computation pipeline, through one interface: (1) graph construction, i.e., extract, transform, and load (ETL), using data primitives; (2) implementing a graph algorithm, using graph primitives; and (3) analyzing the output of the algorithm and/or feeding the output to another pipeline, using data primitives. We show that combining graph and data primitives can be helpful not only in providing a uniform interface across (1), (2), and (3), but also for implementing algorithms within step (2).

8.5.1 Using Data Primitives Instead of Graph Primitives

With the exception of filterEdges, our library consists of vertex-centric graph primitives: some computation is performed for each vertex, such as an update, a filter, or emitting a value to be aggregated. Some subgraph finding algorithms can easily be expressed using our vertex-centric primitives, such as finding the $k$-core
of an undirected graph: a subgraph in which each vertex has at least \( k \) edges. We can find the \( k \)-core of a graph by using our \texttt{filterVerticesUsingLocalEdges} primitive iteratively, removing vertices with fewer than \( k \) edges:

```java
var previousNumVertices = Long.MaxValue
while (previousNumVertices != g.numVertices) {
    previousNumVertices = g.numVertices
    g.filterVerticesUsingLocalEdges(EdgeDirection.Either,
        (vId, vval, edges) → edges.get.size >= k })
```

In contrast, some subgraph finding algorithms are “edge-centric”. They search for particular edge patterns, such as three or four edge cycles when finding triangles or rectangles, or they require computing a value for each edge, such as finding the \( k \)-truss of a graph: a subgraph in which each edge participates in at least \( k - 2 \) triangles. Such edge-centric algorithms cannot be expressed easily with our primitives. We take finding triangles as a simple example.

The standard approach to finding triangles \cite{72, 104, 120} takes two steps: (1) compute open-triads, i.e., pairs of edges of the form \((u,v) (v,w)\); then (2) search for the closing edges \((u,w)\). If we used our graph primitives, we would likely perform this computation as follows (to simplify the code, we ignore overcounting of the triangles):

```scala
val localEdges = edges.map(edge => {edge.otherVertexId(vid)});
new Value(localEdges, null /* no nbs’ adjacency lists yet */))
```

```
// copy each neighbor’s adjacency lists to local value
```

```scala
g.aggregateNeighborValues(
    EdgeDirection.Either,
    (vId, vval) => true /* nbrP */, (vId, vval) => true /* vP */,
    (nbrId, nbrdata) => Array((nbrId, nbrdata.adjList)),
    // concatenate all (nbrId, nbrAdjList)
```
First, each vertex \( u \) copies its adjacency list to its value using \texttt{updateVerticesUsingLocalEdges}. Then, each vertex \( u \) copies over the adjacency list of each of its neighbors \( v \) using \texttt{aggregateNeighborValues}. The aggregation operation in the above code appends the \((\text{nbrId}, \text{nbrAdjList})\) pairs into an array. At this point, \( u \)'s value contains its own adjacency list and an array containing the adjacency lists of its neighbors. Assume \( v \) is a neighbor of \( u \), and \( v \)'s adjacency list contains \( \{w, z\} \). Then \( u \) can infer its open triads with \( v \), namely \((u,v) (v,w)\) and \((u,v) (v,z)\). Finally, \( u \) finds all the triangles it is part of by searching the closing edges of its open triads, such as \((u,w)\) and \((u,z)\), in its own adjacency list using the \texttt{updateVertices} primitive. Notice that the UDFs and vertex values we use inside our primitives are quite complex, and they get even more complex when finding larger subgraph shapes.

A simpler approach is to view the edges as a table with schema \((\text{src}, \text{dst})\), and perform a three-way join on this table using data primitives. Suppose we store the edges table in an RDD called \texttt{edges}, and that an edge \((u,v)\) exists in \texttt{edges} both as \((u,v)\) and \((v,u)\). We again ignore overcounting in the following code:

\[
\text{val openTriads = edges.join(edges)}
\]

\[
\text{val openTriadsClosingEdgesFirstColumn are of the form: } (((u, w), v))
\]
We first self-join edges to compute the openTriads table. We then join openTriads with edges to close the triangles. Note that Spark only allows joining tables on the first column, so in between the two join operations we rearrange the columns of openTriads. Finding larger subgraphs can similarly be expressed as a multi-way join using data primitives.

### 8.5.2 Using Data Primitives Together With Graph Primitives

Now we show the benefits of directly combining data primitives and graph primitives. Suppose we want to compute the clustering coefficient \[ \frac{2}{k(k-1)} \] of each vertex \( v \): the ratio of the actual number of triangles \( v \) participates in to the maximum number of triangles \( v \) can participate in. The maximum is the number of open triads \( v \) is in, or \( \frac{k(k-1)}{2} \) where \( k \) is the degree of \( v \). We can perform this computation using data and graph primitives together as follows: (1) compute the number of triangles for each vertex from triangles by using Spark’s flatMap and reduce primitives, yielding a vertexIDNumTriangles RDD; (2) construct a graph object, using edges as EdgesRDD, and vertexIDNumTriangles as VerticesRDD; (3) compute the clustering coefficient of each vertex using updateVerticesUsingLocalEdges. The code is:

```scala
val vertexIDNumTriangles = triangles.flatMap(
    tri → Iterator((tri._1, 1), (tri._2, 1), (tri._3, 1)).reduceByKey(AggrFn.SUM)
val graph = Graph(vertexIDNumTriangles, edges)
// vval contains an integer value containing the number of triangles
// we return vval/(k(k-1)/2)
g.updateVerticesUsingLocalEdges(v → true,
    (vId, data, edges) => vval/((edges.length*edges.lengh−1)/2))
```
Continuing with our example, suppose we next want to compute the *global clustering coefficient* of the graph: average clustering coefficient of the vertices. We can perform this computation with `aggregateGlobalValue`:

```scala
val sumCoeff = g.aggregateGlobalValue((vId, vval) → vval, AggrFn.SUM)
val globalClusteringCoeff = sumCoeff / g.numVertices
```

If we wanted to compute the global clustering coefficient without computing local coefficients explicitly, instead of `updateVerticesUsingLocalEdges`, we could use `aggregateGlobalValueUsingLocalEdges` directly.

### 8.6 Related Work

No other work we know of at this time proposes and implements high-level primitives for distributed graph computations. Related work divides roughly into four categories: APIs of existing distributed systems for large-scale graph computations, high-level languages for distributed data analysis, domain-specific languages for graph computations, and MPI-based graph libraries. We note that Chapter 5 covered some of the related work on the APIs of existing distributed systems for graph computations and MPI-based graph libraries. In Chapter 5, we compared these references against GPS’s API. Here, we contrast them against our HelP primitives.

#### 8.6.1 APIs of Existing Distributed Systems

We first consider existing distributed systems for large-scale graph computations, reviewing specifically the different APIs exposed by these systems. The APIs of Spark (version 0.8.1) and the earlier version of GraphX on which we built our primitives were discussed in Section 8.1. APIs of existing distributed systems fall into two categories: vertex-centric and MapReduce-based.
Vertex-centric APIs

PowerGraph [50] is a distributed message-passing system that can execute computations both synchronously and asynchronously, in the latter case without any clearly defined supersteps. The PowerGraph API is a set of three vertex-centric functions: \texttt{gather()}, \texttt{apply()}, and \texttt{scatter()}. Using these functions, programmers define the messaging behavior of the vertices and how to update vertex values using received messages.

As discussed in the beginning of this chapter, vertex-centric functions can be too low-level for certain operations, such as forming supervertices or updating vertices using values from another vertex, resulting in complex vertex values and messaging between vertices. Also, as observed in [60, 62] and Chapter 5 of this thesis, the APIs of these systems do not offer mechanisms for controlling the flow of programs, which makes it challenging to write algorithms composed of multiple operations: workarounds using global data structures are needed to control the flow from one operation to the next. In contrast, multiple operations in our primitives are expressed naturally by writing the primitives one after another.

MapReduce-based APIs

MapReduce [36], Hadoop [9], and systems that extend Hadoop can also be used for graph computations. To perform graph processing in these systems, the graph structure and the vertex and edge values are encoded in distributed files or in a distributed database. In basic MapReduce or Hadoop, programmers implement one or more pairs of \texttt{map()} and \texttt{reduce()} functions to simulate each graph operation in their algorithms. For computations that iterate until vertex values converge, programmers implement additional \texttt{map()} and \texttt{reduce()} functions to check convergence criteria. The APIs of iterative extensions of Hadoop, such as HaLoop [27] and iMapReduce [135], provide special additional functions for checking convergence criteria.

MapReduce-based APIs have similar shortcomings to vertex-centric ones, and in some sense are even lower level, since they are not part of a graph-specific
abstraction. Likewise they do not provide mechanisms for controlling the flow of operations, so an extra script needs to be written to assemble all of the \texttt{map()} and \texttt{reduce()} functions into a single program.

### 8.6.2 Higher-level Data Analysis Languages

Another way to implement graph algorithms for execution on MapReduce or Hadoop is through a higher-level language that compiles to these systems, such as \textit{Pig Latin} \cite{98}, \textit{Hive} \cite{121}, or \textit{Scalding} \cite{107}. When using these languages, the graph structure and vertex and edge values again are encoded in distributed storage. However, instead of writing \texttt{map()} and \texttt{reduce()} functions, programmers use higher-level constructs. In Pig Latin and Scalding, just as in Spark, programmers use data primitives such as \texttt{filter}, \texttt{group}, and \texttt{join}; in Hive programmers use a declarative SQL-like language, including constructs such as \texttt{select}, \texttt{update}, and \texttt{groupBy}. SparkSQL \cite{14} is similar to Hive, exposing a SQL layer over Spark that can be used to implement graph algorithms that execute on Spark.

Unlike our HelP primitives, which are graph-specific and operate on a graph abstraction, these languages expose relational data primitives and operate on tables consisting of tuples or key-value pairs. Thus, graph operations that we abstract in a single primitive can require a large number of data primitives. For example, if we were to bypass GraphX and use only Spark’s primitives, programmers would need to use eight data primitives for \texttt{propagateAndAggregate} and six for \texttt{updateSelfUsingAnotherVertex}.

### 8.6.3 Domain-Specific Languages

Recently, Green-Marl \cite{62}, a domain-specific graph language, was introduced specifically for implementing distributed and parallel graph algorithms. Green-Marl is an imperative language with graph-specific constructs, such as \texttt{Graph} and \texttt{Node} types to represent a graph, and \texttt{InBFS} and \texttt{InDFS} constructs to express breadth-first and depth-first graph traversals. Programs are written as if the graph is stored in a single machine, but compile to different parallel or distributed backends. Currently
Green-Marl compiles to parallel C++ code and to two distributed graph-processing systems: Giraph and GPS. Only a subset of Green-Marl, called *Pregel-canonical* programs, can be compiled to run on GPS or Giraph [64], so programmers need to be aware of which code patterns and constructs violate the restrictions of the Green-Marl compiler. In contrast, all of our primitives abstract distributable operations.

8.6.4 MPI-based Graph Libraries

Finally, there are several graph libraries based on *Message Passing Interface* (MPI). MPI is a standard interface for building parallel and distributed message-passing programs, which can be used for graph processing. All MPI-based systems and libraries can compile to and execute on distributed architectures that have an MPI implementation, such as Open MPI [99] or MPICH [94].

PBGL [57] exposes a set of templates for distributed data structures, such as adjacency lists, and also a library of existing algorithms, such as single-source shortest paths and weakly-connected components, which the programmer can invoke directly. However, the configuration of these templates requires programmers to code very low-level details, such as how to distribute the vertices and adjacency lists, and how to traverse them during the execution. In addition, programmers need to write manual code to describe which processors send each other messages during execution.

Combinatorial BLAS (CBLAS) [28] is another MPI-based library that exposes a set of linear algebra primitives, such as primitives that multiply a sparse matrix with a vector, or multiply two sparse matrices. Many graph algorithms can be implemented using linear algebra primitives. However, programmers need expertise in the linear algebra abstraction of graphs, as opposed to the classic graph abstraction consisting of nodes, edges, and values on nodes and edges, which our primitives are designed for. KDT [86] is a library of graph algorithms built on top of Combinatorial BLAS, which can be used to perform graph analysis tasks, but it does not have an API to develop custom algorithms.
Chapter 9

Summary and Future Work

This thesis studied distributed shared-nothing systems for processing very large-scale data, with a focus on two areas: (i) theoretical foundations for understanding the costs of distribution within the context of the MapReduce system; and (ii) processing large-scale graphs within the context of Pregel-like systems. Chapter 1 gave an overview of the MapReduce and Pregel systems, and motivated the problems we studied in this thesis. In this chapter, we first summarize our main contributions on theoretical foundations and distributed graph processing systems in Sections 9.1 and 9.2 respectively. Then, in Section 9.3 we discuss general directions for future research arising from the research contributions of this thesis.

9.1 Summary of Contributions on Theoretical Foundations

The first part of this thesis was motivated by two questions: (1) How costly is it to parallelize different problems in distributed data-processing systems? (2) How should we measure the “goodness” of different algorithms for solving a problem? We provided answers to these questions within the context of the MapReduce system, first considering the limited setting when computations are constrained to a single round of MapReduce. We then extended some of our results to multiround
MapReduce. Chapters 2-4 covered our contributions in this area:

- **A Theoretical Framework for MapReduce.** In Chapter 2 we described a theoretical framework for MapReduce to characterize the costs of parallelizing different problems and to measure how “good” different algorithms are. We observed that there is an inherent tradeoff between parallelism, or reducer size, and communication cost when solving problems. The tradeoff curve for a problem represents the minimum communication required to solve the problem for different parallelism levels, or intuitively the cost of parallelizing the problem. We presented a proof template for deriving the tradeoff curves of problems. We quantified the performance of algorithms by measuring their reducer sizes and communication costs, and we introduced the notion of Pareto optimality in our setting. We then used our framework to analyze the costs of parallelizing three problems: fuzzy joins with Hamming distance 1, dense matrix multiplication, and multiway equijoins. We derived the tradeoff curves of these problems and presented existing or new Pareto optimal algorithms.

- **Fuzzy Joins with Hamming and Edit Distance \( d > 1 \).** In Chapter 3 we studied the fuzzy joins problem for Hamming and edit distances greater than 1. We could not derive a tradeoff curve for fuzzy joins with large Hamming and edit distances, so it remains an open problem. We described our Generalized Anchor Points (GAP) algorithm, which improves and generalizes the Anchor Points algorithm from reference [3]. We showed that GAP is flexible in its parallelism and performs better than existing algorithms. GAP relies on constructing covering codes for Hamming and edit distances for different radiiuses. The main technical contribution of this chapter was the construction of the first covering codes for the edit-distance problem.

- **Multiround MapReduce.** Chapter 4 studied multiround algorithms for MapReduce. We showed that we can often beat the one-round tradeoff curves by executing multiple rounds of MapReduce. We gave examples from the problems of dense matrix multiplication and multiway equijoins. The main contribution of this chapter was the GYM algorithm for solving multiway...
equijoins. GYM takes as input a query $Q$ and a GHD of $Q$. On a GHD of width $w$ and depth $d$, GYM executes $O(d)$ rounds of computation with a communication cost that increases as a function of $w$. We showed methods for constructing GHDs of different depths and widths for queries, thereby exposing a spectrum of tradeoffs one can make between the number of rounds and communication.

9.2 Summary of Contributions on Distributed Graph Processing Systems

The second part of this thesis studied Pregel-like systems for processing large-scale graphs. Our contributions were motivated by two problems with the original Pregel system: (1) algorithms running on Pregel can be inefficient; and (2) programming and debugging `vertex.compute()` functions can be very challenging. Chapters 5-8 covered our contributions in this area:

- **GPS: A Graph Processing System.** Chapter 5 began by describing our implementation of an open-source version of Pregel called GPS, which we built from scratch. We described three additional features of GPS that did not exist in Pregel or other open-source Pregel-like systems: (1) an extension to Pregel’s original API called `master.compute()`, which simplifies implementing algorithms composed of one or more vertex-centric computations combined with sequential computations; (2) a dynamic repartitioning feature to reduce the communication costs of some algorithms; and (3) the LALP optimization, which partitions adjacency lists of high-degree vertices across compute nodes, reducing the communication cost of some algorithms on graphs with skewed-degree distributions. We presented experiments demonstrating how different ways of partitioning, and possibly repartitioning, graphs across compute nodes affects the performance of algorithms. Not surprisingly, partitioning schemes that maintain a good workload balance across compute nodes can yield significant run-time benefits.
• **Optimizing Graph Algorithms for Pregel-like Systems.** In Chapter 6 we studied the problem of implementing algorithms efficiently in Pregel-like systems. We showed that some algorithms incur high communication or computation costs, or converge very slowly, due to structural properties of the input graphs such as large diameters or skew in component sizes. We presented a set of algorithmic optimizations, some of which are applicable across multiple algorithms, to address some of the commonly appearing inefficiencies across algorithms.

• **Debugging Graph Algorithms for Pregel-like Systems.** Chapter 7 tackled the problem of debugging programs written for Pregel-like systems. We observed that programmers typically go through a three-step manual debugging cycle based on putting print statements into their compute() functions, examining the output these statements, then rechecking their compute() functions to detect bugs. We described the design and implementation of a new replay-style debugger called Graft we built for the Apache Giraph system. Graft’s approach was motivated by the three steps above, which we called capture, visualize, and reproduce, respectively. The three corresponding debugging steps using Graft are: (1) programmatically describe a set of vertices to capture information about; (2) replay the values and messages of these vertices superstep by superstep in a graph-specific GUI; and (3) perform line-by-line debugging inside a local debugger to see the exact lines of the compute() function that ran for a specific captured vertex and superstep. We demonstrated Graft’s debugging capabilities through three detailed debugging scenarios.

• **Higher-level Primitives for Distributed Graph Algorithms.** In Chapter 8 we proposed a set of high-level graph primitives called HelP, as an alternative to programming algorithms with low-level vertex-centric functions. The HelP primitives capture several commonly appearing operations in large-scale graph computations. We described the HelP primitives, our implementation
of these primitives on top of the GraphX system \cite{130}, and the implementation of many graph algorithms using the HelP primitives. We found that implementing algorithms using the HelP primitives is more intuitive and faster than using the APIs of existing distributed graph processing systems.

9.3 Future Directions

We describe a few open research areas arising from the work in this thesis. We divide the presentation along the broad two areas of this thesis, and provide a few suggested future directions in each one.

9.3.1 Theoretical Foundations

- **Tradeoff Curves for Multiround Computations.** The tradeoff curves we derived in this thesis captured the cost of parallelizing computations in a single round of MapReduce. The most interesting and natural future direction arising from the first part of this thesis is to derive tradeoff curves for multiround computations. In Chapter \[4\] we demonstrated that we can beat the tradeoff curve for one-round computations by designing multiround algorithms. However, it is an open problem whether the multiround algorithms we described are optimal, or even close to what is theoretically possible. Deriving tradeoff curves for multiround computations would be a deep contribution to understanding the computational power of synchronization between machines, which is primarily what defines rounds of computation.

- **Distributed Streaming Computations.** Our framework and algorithms focused on MapReduce, which performs batch, offline computations. However, many applications process data that arrives continuously as streams, or in many small batches or updates. An interesting future direction is developing frameworks and distributed algorithms that are suitable for a streaming or incremental environment. An important application of distributed streaming algorithms is in distributed materialized views in the relational setting. This
feature is not yet supported in existing widely-used distributed relational data processing systems such as Hive [121] or SparkSQL [14].

9.3.2 Distributed Graph Processing Systems

- **Memory-Constrained Graph Processing.** Current graph-processing systems, including the Pregel-like systems we focused on, assume there is sufficient RAM at each machine, which may not always be true in practice. Storage layers of distributed graph-processing systems could be made more robust to working under limited memory. A solution utilizing disk needs to handle three main operations carefully: (1) computation on the vertices; (2) a mix of memory and disk I/O for accessing the graph, vertex values, and messages; and (3) network I/O for sending and receiving messages across machines. One primary challenge to making the system efficient is to enable as much concurrency as possible among the different types and instances of operations.

- **Network Optimizations.** Existing distributed graph processing systems, including GPS, tend to have straightforward implementations of their networking modules for sending and receiving messages between machines: batching of messages is typically the only optimization in the current implementations of existing systems. An interesting future direction is to explore other network optimizations, for example: When a machine has multiple batches ready to send, does the order in which the messages are sent make a difference? When one machine has more workload than others, could we speed it up with a network layer optimization? Should the messaging be aware of the rack allocations of the machines? Answering these questions could yield simple yet effective optimizations.

- **Evolving Graphs.** Pregel-like systems perform batch, offline processing of graphs, in which the graph topology is assumed to be fixed. However, many real-world graphs evolve over time, with vertices and edges entering and leaving the graph. Pregel-like systems pose obstacles for applications that want to
perform computations on multiple snapshots of the same graph, say to detect the parts of the graph that are changing most frequently. An interesting research direction is to incorporate explicit support for performing computations on evolving graphs.
Bibliography


[99] Open MPI.


[101] Phoebus.


