Mining of Massive Datasets

Jure Leskovec Stanford Univ. Anand Rajaraman Milliway Labs

Jeffrey D. Ullman Stanford Univ.

Copyright c 2010, 2011, 2012, 2013, 2014 Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman

ii

Preface

This book evolved from material developed over several years by Anand Raja- raman and Jeff Ullman for a one-quarter course at Stanford. The course CS345A, titled “Web Mining,” was designed as an advanced graduate course, although it has become accessible and interesting to advanced undergraduates. When Jure Leskovec joined the Stanford faculty, we reorganized the material considerably. He introduced a new course CS224W on network analysis and added material to CS345A, which was renumbered CS246. The three authors also introduced a large-scale data-mining project course, CS341. The book now contains material taught in all three courses.

What the Book Is About

At the highest level of description, this book is about data mining. However, it focuses on data mining of very large amounts of data, that is, data so large it does not fit in main memory. Because of the emphasis on size, many of our examples are about the Web or data derived from the Web. Further, the book takes an algorithmic point of view: data mining is about applying algorithms to data, rather than using data to “train” a machine-learning engine of some sort. The principal topics covered are:

1. Distributed file systems and map-reduce as a tool for creating parallel

algorithms that succeed on very large amounts of data.

2. Similarity search, including the key techniques of minhashing and locality-

sensitive hashing.

3. Data-stream processing and specialized algorithms for dealing with data

that arrives so fast it must be processed immediately or lost.

4. The technology of search engines, including Google’s PageRank, link-spam

detection, and the hubs-and-authorities approach.

5. Frequent-itemset mining, including association rules, market-baskets, the

A-Priori Algorithm and its improvements.

6. Algorithms for clustering very large, high-dimensional datasets.

iii

iv PREFACE

7. Two key problems for Web applications: managing advertising and rec-

ommendation systems.

8. Algorithms for analyzing and mining the structure of very large graphs,

especially social-network graphs.

9. Techniques for obtaining the important properties of a large dataset by dimensionality reduction, including singular-value decomposition and la- tent semantic indexing.

10. Machine-learning algorithms that can be applied to very large data, such

as perceptrons, support-vector machines, and gradient descent.

Prerequisites

To appreciate fully the material in this book, we recommend the following prerequisites:

1. An introduction to database systems, covering SQL and related program-

ming systems.

2. A sophomore-level course in data structures, algorithms, and discrete

math.

3. A sophomore-level course in software systems, software engineering, and

programming languages.

Exercises

The book contains extensive exercises, with some for almost every section. We indicate harder exercises or parts of exercises with an exclamation point. The hardest exercises have a double exclamation point.

Support on the Web

Go to http://www.mmds.orgfor slides, homework assignments, project require- ments, and exams from courses related to this book.

Gradiance Automated Homework

There are automated exercises based on this book, using the Gradiance root- question technology, available at www.gradiance.com/services. Students may enter a public class by creating an account at that site and entering the class with code 1EDD8A1D. Instructors may use the site by making an account there

PREFACE v

and then emailing support at gradiance dot com with their login name, the name of their school, and a request to use the MMDS materials.

Acknowledgements

Cover art is by Scott Ullman.

We would like to thank Foto Afrati, Arun Marathe, and Rok Sosic for critical readings of a draft of this manuscript.

Errors were also reported by Rajiv Abraham, Ruslan Aduk, Apoorv Agar- wal, Aris Anagnostopoulos, Yokila Arora, Stefanie Anna Baby, Atilla Soner Balkir, Arnaud Belletoile, Robin Bennett, Susan Biancani, Richard Boyd, Amitabh Chaudhary, Leland Chen, Hua Feng, Marcus Gemeinder, Anastasios Gounar- is, Clark Grubb, Shrey Gupta, Waleed Hameid, Saman Haratizadeh, Julien Hoachuck, Przemyslaw Horban, Hsiu-Hsuan Huang, Jeff Hwang, Rafi Kamal, Lachlan Kang, Ed Knorr, Haewoon Kwak, Ellis Lau, Greg Lee, David Z. Liu, Ethan Lozano, Yunan Luo, Michael Mahoney, Sergio Matos, Justin Meyer, Bryant Moscon, Brad Penoff, John Phillips, Philips Kokoh Prasetyo, Qi Ge, Harizo Rajaona, Timon Ruban, Rich Seiter, Hitesh Shetty, Angad Singh, Sandeep Sripada, Dennis Sidharta, Krzysztof Stencel, Mark Storus, Roshan Sumbaly, Zack Taylor, Tim Triche Jr., Wang Bin, Weng Zhen-Bin, Robert West, Steven Euijong Whang, Oscar Wu, Xie Ke, Christopher T.-R. Yeh, Nicolas Zhao, and Zhou Jingbo, The remaining errors are ours, of course.

J. L. A. R. J. D. U. Palo Alto, CA March, 2014

vi PREFACE

Contents

1 Data Mining 1 1.1 What is Data Mining? . . . . . . . . . . . . . . . . . . . . . . . . 1 1.1.1 Statistical Modeling . . . . . . . . . . . . . . . . . . . . . 1 1.1.2 Machine Learning . . . . . . . . . . . . . . . . . . . . . . 2 1.1.3 Computational Approaches to Modeling . . . . . . . . . . 2 1.1.4 Summarization . . . . . . . . . . . . . . . . . . . . . . . . 3 1.1.5 Feature Extraction . . . . . . . . . . . . . . . . . . . . . . 4 1.2 Statistical Limits on Data Mining . . . . . . . . . . . . . . . . . . 4 1.2.1 Total Information Awareness . . . . . . . . . . . . . . . . 5 1.2.2 Bonferroni’s Principle . . . . . . . . . . . . . . . . . . . . 5 1.2.3 An Example of Bonferroni’s Principle . . . . . . . . . . . 6 1.2.4 Exercises for Section 1.2 . . . . . . . . . . . . . . . . . . . 7 1.3 Things Useful to Know . . . . . . . . . . . . . . . . . . . . . . . . 7 1.3.1 Importance of Words in Documents . . . . . . . . . . . . 8 1.3.2 Hash Functions . . . . . . . . . . . . . . . . . . . . . . . . 9 1.3.3 Indexes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10 1.3.4 Secondary Storage . . . . . . . . . . . . . . . . . . . . . . 11 1.3.5 The Base of Natural Logarithms . . . . . . . . . . . . . . 12 1.3.6 Power Laws . . . . . . . . . . . . . . . . . . . . . . . . . . 13 1.3.7 Exercises for Section 1.3 . . . . . . . . . . . . . . . . . . . 15 1.4 Outline of the Book . . . . . . . . . . . . . . . . . . . . . . . . . 15 1.5 Summary of Chapter 1 . . . . . . . . . . . . . . . . . . . . . . . . 17 1.6 References for Chapter 1 . . . . . . . . . . . . . . . . . . . . . . . 18

2 MapReduce and the New Software Stack 21 2.1 Distributed File Systems . . . . . . . . . . . . . . . . . . . . . . . 22 2.1.1 Physical Organization of Compute Nodes . . . . . . . . . 22 2.1.2 Large-Scale File-System Organization . . . . . . . . . . . 23 2.2 MapReduce . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24 2.2.1 The Map Tasks . . . . . . . . . . . . . . . . . . . . . . . . 25 2.2.2 Grouping by Key . . . . . . . . . . . . . . . . . . . . . . . 26 2.2.3 The Reduce Tasks . . . . . . . . . . . . . . . . . . . . . . 27 2.2.4 Combiners. . . . . . . . . . . . . . . . . . . . . . . . . . . 27

vii

viii CONTENTS

2.2.5 Details of MapReduce Execution . . . . . . . . . . . . . . 28 2.2.6 Coping With Node Failures . . . . . . . . . . . . . . . . . 29 2.2.7 Exercises for Section 2.2 . . . . . . . . . . . . . . . . . . . 30 2.3 Algorithms Using MapReduce . . . . . . . . . . . . . . . . . . . . 30 2.3.1 Matrix-Vector Multiplication by MapReduce . . . . . . . 31 2.3.2 If the Vector v Cannot Fit in Main Memory . . . . . . . . 31 2.3.3 Relational-Algebra Operations . . . . . . . . . . . . . . . 32 2.3.4 Computing Selections by MapReduce . . . . . . . . . . . 35 2.3.5 Computing Projections by MapReduce . . . . . . . . . . . 36 2.3.6 Union, Intersection, and Difference by MapReduce . . . . 36 2.3.7 Computing Natural Join by MapReduce . . . . . . . . . . 37 2.3.8 Grouping and Aggregation by MapReduce . . . . . . . . . 37 2.3.9 Matrix Multiplication . . . . . . . . . . . . . . . . . . . . 38 2.3.10 Matrix Multiplication with One MapReduce Step . . . . . 39 2.3.11 Exercises for Section 2.3 . . . . . . . . . . . . . . . . . . . 40 2.4 Extensions to MapReduce . . . . . . . . . . . . . . . . . . . . . . 41 2.4.1 Workflow Systems . . . . . . . . . . . . . . . . . . . . . . 41 2.4.2 Recursive Extensions to MapReduce . . . . . . . . . . . . 42 2.4.3 Pregel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45 2.4.4 Exercises for Section 2.4 . . . . . . . . . . . . . . . . . . . 46 2.5 The Communication Cost Model . . . . . . . . . . . . . . . . . . 46 2.5.1 Communication-Cost for Task Networks . . . . . . . . . . 47 2.5.2 Wall-Clock Time . . . . . . . . . . . . . . . . . . . . . . . 49 2.5.3 Multiway Joins . . . . . . . . . . . . . . . . . . . . . . . . 49 2.5.4 Exercises for Section 2.5 . . . . . . . . . . . . . . . . . . . 52 2.6 Complexity Theory for MapReduce . . . . . . . . . . . . . . . . . 54 2.6.1 Reducer Size and Replication Rate . . . . . . . . . . . . . 54 2.6.2 An Example: Similarity Joins . . . . . . . . . . . . . . . . 55 2.6.3 A Graph Model for MapReduce Problems . . . . . . . . . 57 2.6.4 Mapping Schemas . . . . . . . . . . . . . . . . . . . . . . 58 2.6.5 When Not All Inputs Are Present . . . . . . . . . . . . . 60 2.6.6 Lower Bounds on Replication Rate . . . . . . . . . . . . . 61 2.6.7 Case Study: Matrix Multiplication . . . . . . . . . . . . . 62 2.6.8 Exercises for Section 2.6 . . . . . . . . . . . . . . . . . . . 66 2.7 Summary of Chapter 2 . . . . . . . . . . . . . . . . . . . . . . . . 67 2.8 References for Chapter 2 . . . . . . . . . . . . . . . . . . . . . . . 69

3 Finding Similar Items 73 3.1 Applications of Near-Neighbor Search . . . . . . . . . . . . . . . 73 3.1.1 Jaccard Similarity of Sets . . . . . . . . . . . . . . . . . . 74 3.1.2 Similarity of Documents . . . . . . . . . . . . . . . . . . . 74 3.1.3 Collaborative Filtering as a Similar-Sets Problem . . . . . 75 3.1.4 Exercises for Section 3.1 . . . . . . . . . . . . . . . . . . . 77 3.2 Shingling of Documents . . . . . . . . . . . . . . . . . . . . . . . 77 3.2.1 k-Shingles . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

CONTENTS ix

3.2.2 Choosing the Shingle Size . . . . . . . . . . . . . . . . . . 78 3.2.3 Hashing Shingles . . . . . . . . . . . . . . . . . . . . . . . 79 3.2.4 Shingles Built from Words . . . . . . . . . . . . . . . . . . 79 3.2.5 Exercises for Section 3.2 . . . . . . . . . . . . . . . . . . . 80 3.3 Similarity-Preserving Summaries of Sets . . . . . . . . . . . . . . 80 3.3.1 Matrix Representation of Sets . . . . . . . . . . . . . . . . 81 3.3.2 Minhashing . . . . . . . . . . . . . . . . . . . . . . . . . . 81 3.3.3 Minhashing and Jaccard Similarity . . . . . . . . . . . . . 82 3.3.4 Minhash Signatures . . . . . . . . . . . . . . . . . . . . . 83 3.3.5 Computing Minhash Signatures . . . . . . . . . . . . . . . 83 3.3.6 Exercises for Section 3.3 . . . . . . . . . . . . . . . . . . . 86 3.4 Locality-Sensitive Hashing for Documents . . . . . . . . . . . . . 87 3.4.1 LSH for Minhash Signatures . . . . . . . . . . . . . . . . 88 3.4.2 Analysis of the Banding Technique . . . . . . . . . . . . . 89 3.4.3 Combining the Techniques . . . . . . . . . . . . . . . . . . 91 3.4.4 Exercises for Section 3.4 . . . . . . . . . . . . . . . . . . . 91 3.5 Distance Measures . . . . . . . . . . . . . . . . . . . . . . . . . . 92 3.5.1 Definition of a Distance Measure . . . . . . . . . . . . . . 92 3.5.2 Euclidean Distances . . . . . . . . . . . . . . . . . . . . . 93 3.5.3 Jaccard Distance . . . . . . . . . . . . . . . . . . . . . . . 94 3.5.4 Cosine Distance. . . . . . . . . . . . . . . . . . . . . . . . 95 3.5.5 Edit Distance . . . . . . . . . . . . . . . . . . . . . . . . . 95 3.5.6 Hamming Distance . . . . . . . . . . . . . . . . . . . . . . 96 3.5.7 Exercises for Section 3.5 . . . . . . . . . . . . . . . . . . . 97 3.6 The Theory of Locality-Sensitive Functions . . . . . . . . . . . . 99 3.6.1 Locality-Sensitive Functions . . . . . . . . . . . . . . . . . 99 3.6.2 Locality-Sensitive Families for Jaccard Distance . . . . . . 100 3.6.3 Amplifying a Locality-Sensitive Family . . . . . . . . . . . 101 3.6.4 Exercises for Section 3.6 . . . . . . . . . . . . . . . . . . . 103 3.7 LSH Families for Other Distance Measures . . . . . . . . . . . . . 104 3.7.1 LSH Families for Hamming Distance . . . . . . . . . . . . 104 3.7.2 Random Hyperplanes and the Cosine Distance . . . . . . 105 3.7.3 Sketches . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106 3.7.4 LSH Families for Euclidean Distance . . . . . . . . . . . . 107 3.7.5 More LSH Families for Euclidean Spaces . . . . . . . . . . 108 3.7.6 Exercises for Section 3.7 . . . . . . . . . . . . . . . . . . . 109 3.8 Applications of Locality-Sensitive Hashing . . . . . . . . . . . . . 110 3.8.1 Entity Resolution . . . . . . . . . . . . . . . . . . . . . . . 110 3.8.2 An Entity-Resolution Example . . . . . . . . . . . . . . . 111 3.8.3 Validating Record Matches . . . . . . . . . . . . . . . . . 112 3.8.4 Matching Fingerprints . . . . . . . . . . . . . . . . . . . . 113 3.8.5 A LSH Family for Fingerprint Matching . . . . . . . . . . 114 3.8.6 Similar News Articles . . . . . . . . . . . . . . . . . . . . 115 3.8.7 Exercises for Section 3.8 . . . . . . . . . . . . . . . . . . . 117 3.9 Methods for High Degrees of Similarity . . . . . . . . . . . . . . 118

x CONTENTS

3.9.1 Finding Identical Items . . . . . . . . . . . . . . . . . . . 118 3.9.2 Representing Sets as Strings . . . . . . . . . . . . . . . . . 118 3.9.3 Length-Based Filtering . . . . . . . . . . . . . . . . . . . . 119 3.9.4 Prefix Indexing . . . . . . . . . . . . . . . . . . . . . . . . 119 3.9.5 Using Position Information . . . . . . . . . . . . . . . . . 121 3.9.6 Using Position and Length in Indexes . . . . . . . . . . . 122 3.9.7 Exercises for Section 3.9 . . . . . . . . . . . . . . . . . . . 125 3.10 Summary of Chapter 3 . . . . . . . . . . . . . . . . . . . . . . . . 126 3.11 References for Chapter 3 . . . . . . . . . . . . . . . . . . . . . . . 128

4 Mining Data Streams 131 4.1 The Stream Data Model . . . . . . . . . . . . . . . . . . . . . . . 131 4.1.1 A Data-Stream-Management System . . . . . . . . . . . . 132 4.1.2 Examples of Stream Sources . . . . . . . . . . . . . . . . . 133 4.1.3 Stream Queries . . . . . . . . . . . . . . . . . . . . . . . . 134 4.1.4 Issues in Stream Processing . . . . . . . . . . . . . . . . . 135 4.2 Sampling Data in a Stream . . . . . . . . . . . . . . . . . . . . . 136 4.2.1 A Motivating Example . . . . . . . . . . . . . . . . . . . . 136 4.2.2 Obtaining a Representative Sample . . . . . . . . . . . . . 137 4.2.3 The General Sampling Problem . . . . . . . . . . . . . . . 137 4.2.4 Varying the Sample Size . . . . . . . . . . . . . . . . . . . 138 4.2.5 Exercises for Section 4.2 . . . . . . . . . . . . . . . . . . . 138 4.3 Filtering Streams . . . . . . . . . . . . . . . . . . . . . . . . . . . 139 4.3.1 A Motivating Example . . . . . . . . . . . . . . . . . . . . 139 4.3.2 The Bloom Filter . . . . . . . . . . . . . . . . . . . . . . . 140 4.3.3 Analysis of Bloom Filtering . . . . . . . . . . . . . . . . . 140 4.3.4 Exercises for Section 4.3 . . . . . . . . . . . . . . . . . . . 141 4.4 Counting Distinct Elements in a Stream . . . . . . . . . . . . . . 142 4.4.1 The Count-Distinct Problem . . . . . . . . . . . . . . . . 142 4.4.2 The Flajolet-Martin Algorithm . . . . . . . . . . . . . . . 143 4.4.3 Combining Estimates . . . . . . . . . . . . . . . . . . . . 144 4.4.4 Space Requirements . . . . . . . . . . . . . . . . . . . . . 144 4.4.5 Exercises for Section 4.4 . . . . . . . . . . . . . . . . . . . 145 4.5 Estimating Moments . . . . . . . . . . . . . . . . . . . . . . . . . 145 4.5.1 Definition of Moments . . . . . . . . . . . . . . . . . . . . 145 4.5.2 The Alon-Matias-Szegedy Algorithm for Second

Moments . . . . . . . . . . . . . . . . . . . . . . . . . . . 146 4.5.3 Why the Alon-Matias-Szegedy Algorithm Works . . . . . 147 4.5.4 Higher-Order Moments . . . . . . . . . . . . . . . . . . . 148 4.5.5 Dealing With Infinite Streams . . . . . . . . . . . . . . . . 148 4.5.6 Exercises for Section 4.5 . . . . . . . . . . . . . . . . . . . 149 4.6 Counting Ones in a Window . . . . . . . . . . . . . . . . . . . . . 150 4.6.1 The Cost of Exact Counts . . . . . . . . . . . . . . . . . . 151 4.6.2 The Datar-Gionis-Indyk-Motwani Algorithm . . . . . . . 151 4.6.3 Storage Requirements for the DGIM Algorithm . . . . . . 153

CONTENTS xi

4.6.4 Query Answering in the DGIM Algorithm . . . . . . . . . 153 4.6.5 Maintaining the DGIM Conditions . . . . . . . . . . . . . 154 4.6.6 Reducing the Error . . . . . . . . . . . . . . . . . . . . . . 155 4.6.7 Extensions to the Counting of Ones . . . . . . . . . . . . 156 4.6.8 Exercises for Section 4.6 . . . . . . . . . . . . . . . . . . . 157 4.7 Decaying Windows . . . . . . . . . . . . . . . . . . . . . . . . . . 157 4.7.1 The Problem of Most-Common Elements . . . . . . . . . 157 4.7.2 Definition of the Decaying Window . . . . . . . . . . . . . 158 4.7.3 Finding the Most Popular Elements . . . . . . . . . . . . 159 4.8 Summary of Chapter 4 . . . . . . . . . . . . . . . . . . . . . . . . 160 4.9 References for Chapter 4 . . . . . . . . . . . . . . . . . . . . . . . 161

5 Link Analysis 163 5.1 PageRank . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163 5.1.1 Early Search Engines and Term Spam . . . . . . . . . . . 164 5.1.2 Definition of PageRank . . . . . . . . . . . . . . . . . . . 165 5.1.3 Structure of the Web . . . . . . . . . . . . . . . . . . . . . 169 5.1.4 Avoiding Dead Ends . . . . . . . . . . . . . . . . . . . . . 170 5.1.5 Spider Traps and Taxation . . . . . . . . . . . . . . . . . 173 5.1.6 Using PageRank in a Search Engine . . . . . . . . . . . . 175 5.1.7 Exercises for Section 5.1 . . . . . . . . . . . . . . . . . . . 175 5.2 Efficient Computation of PageRank . . . . . . . . . . . . . . . . . 177 5.2.1 Representing Transition Matrices . . . . . . . . . . . . . . 178 5.2.2 PageRank Iteration Using MapReduce . . . . . . . . . . . 179 5.2.3 Use of Combiners to Consolidate the Result Vector . . . . 179 5.2.4 Representing Blocks of the Transition Matrix . . . . . . . 180 5.2.5 Other Efficient Approaches to PageRank Iteration . . . . 181 5.2.6 Exercises for Section 5.2 . . . . . . . . . . . . . . . . . . . 183 5.3 Topic-Sensitive PageRank . . . . . . . . . . . . . . . . . . . . . . 183 5.3.1 Motivation for Topic-Sensitive Page Rank . . . . . . . . . 183 5.3.2 Biased Random Walks . . . . . . . . . . . . . . . . . . . . 184 5.3.3 Using Topic-Sensitive PageRank . . . . . . . . . . . . . . 185 5.3.4 Inferring Topics from Words . . . . . . . . . . . . . . . . . 186 5.3.5 Exercises for Section 5.3 . . . . . . . . . . . . . . . . . . . 187 5.4 Link Spam . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 187 5.4.1 Architecture of a Spam Farm . . . . . . . . . . . . . . . . 187 5.4.2 Analysis of a Spam Farm . . . . . . . . . . . . . . . . . . 189 5.4.3 Combating Link Spam . . . . . . . . . . . . . . . . . . . . 190 5.4.4 TrustRank . . . . . . . . . . . . . . . . . . . . . . . . . . 190 5.4.5 Spam Mass . . . . . . . . . . . . . . . . . . . . . . . . . . 191 5.4.6 Exercises for Section 5.4 . . . . . . . . . . . . . . . . . . . 191 5.5 Hubs and Authorities . . . . . . . . . . . . . . . . . . . . . . . . 192 5.5.1 The Intuition Behind HITS . . . . . . . . . . . . . . . . . 192 5.5.2 Formalizing Hubbiness and Authority . . . . . . . . . . . 193 5.5.3 Exercises for Section 5.5 . . . . . . . . . . . . . . . . . . . 196

xii CONTENTS

5.6 Summary of Chapter 5 . . . . . . . . . . . . . . . . . . . . . . . . 196 5.7 References for Chapter 5 . . . . . . . . . . . . . . . . . . . . . . . 200

6 Frequent Itemsets 201 6.1 The Market-Basket Model . . . . . . . . . . . . . . . . . . . . . . 202 6.1.1 Definition of Frequent Itemsets . . . . . . . . . . . . . . . 202 6.1.2 Applications of Frequent Itemsets . . . . . . . . . . . . . 204 6.1.3 Association Rules . . . . . . . . . . . . . . . . . . . . . . . 205 6.1.4 Finding Association Rules with High Confidence . . . . . 207 6.1.5 Exercises for Section 6.1 . . . . . . . . . . . . . . . . . . . 207 6.2 Market Baskets and the A-Priori Algorithm . . . . . . . . . . . . 209 6.2.1 Representation of Market-Basket Data . . . . . . . . . . . 209 6.2.2 Use of Main Memory for Itemset Counting . . . . . . . . 210 6.2.3 Monotonicity of Itemsets . . . . . . . . . . . . . . . . . . 212 6.2.4 Tyranny of Counting Pairs . . . . . . . . . . . . . . . . . 213 6.2.5 The A-Priori Algorithm . . . . . . . . . . . . . . . . . . . 213 6.2.6 A-Priori for All Frequent Itemsets . . . . . . . . . . . . . 214 6.2.7 Exercises for Section 6.2 . . . . . . . . . . . . . . . . . . . 217 6.3 Handling Larger Datasets in Main Memory . . . . . . . . . . . . 218 6.3.1 The Algorithm of Park, Chen, and Yu . . . . . . . . . . . 218 6.3.2 The Multistage Algorithm . . . . . . . . . . . . . . . . . . 220 6.3.3 The Multihash Algorithm . . . . . . . . . . . . . . . . . . 222 6.3.4 Exercises for Section 6.3 . . . . . . . . . . . . . . . . . . . 224 6.4 Limited-Pass Algorithms . . . . . . . . . . . . . . . . . . . . . . . 226 6.4.1 The Simple, Randomized Algorithm . . . . . . . . . . . . 226 6.4.2 Avoiding Errors in Sampling Algorithms . . . . . . . . . . 227 6.4.3 The Algorithm of Savasere, Omiecinski, and

Navathe . . . . . . . . . . . . . . . . . . . . . . . . . . . . 228 6.4.4 The SON Algorithm and MapReduce . . . . . . . . . . . 229 6.4.5 Toivonen’s Algorithm . . . . . . . . . . . . . . . . . . . . 230 6.4.6 Why Toivonen’s Algorithm Works . . . . . . . . . . . . . 231 6.4.7 Exercises for Section 6.4 . . . . . . . . . . . . . . . . . . . 232 6.5 Counting Frequent Items in a Stream . . . . . . . . . . . . . . . . 232 6.5.1 Sampling Methods for Streams . . . . . . . . . . . . . . . 233 6.5.2 Frequent Itemsets in Decaying Windows . . . . . . . . . . 234 6.5.3 Hybrid Methods . . . . . . . . . . . . . . . . . . . . . . . 235 6.5.4 Exercises for Section 6.5 . . . . . . . . . . . . . . . . . . . 235 6.6 Summary of Chapter 6 . . . . . . . . . . . . . . . . . . . . . . . . 236 6.7 References for Chapter 6 . . . . . . . . . . . . . . . . . . . . . . . 238

7 Clustering 241 7.1 Introduction to Clustering Techniques . . . . . . . . . . . . . . . 241 7.1.1 Points, Spaces, and Distances . . . . . . . . . . . . . . . . 241 7.1.2 Clustering Strategies . . . . . . . . . . . . . . . . . . . . . 243 7.1.3 The Curse of Dimensionality . . . . . . . . . . . . . . . . 244

CONTENTS xiii

7.1.4 Exercises for Section 7.1 . . . . . . . . . . . . . . . . . . . 245 7.2 Hierarchical Clustering . . . . . . . . . . . . . . . . . . . . . . . . 245 7.2.1 Hierarchical Clustering in a Euclidean Space . . . . . . . 246 7.2.2 Efficiency of Hierarchical Clustering . . . . . . . . . . . . 248 7.2.3 Alternative Rules for Controlling Hierarchical

Clustering . . . . . . . . . . . . . . . . . . . . . . . . . . . 249 7.2.4 Hierarchical Clustering in Non-Euclidean Spaces . . . . . 252 7.2.5 Exercises for Section 7.2 . . . . . . . . . . . . . . . . . . . 253 7.3 K-means Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . 254 7.3.1 K-Means Basics . . . . . . . . . . . . . . . . . . . . . . . . 255 7.3.2 Initializing Clusters for K-Means . . . . . . . . . . . . . . 255 7.3.3 Picking the Right Value of k . . . . . . . . . . . . . . . . 256 7.3.4 The Algorithm of Bradley, Fayyad, and Reina . . . . . . . 257 7.3.5 Processing Data in the BFR Algorithm . . . . . . . . . . 259 7.3.6 Exercises for Section 7.3 . . . . . . . . . . . . . . . . . . . 262 7.4 The CURE Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 262 7.4.1 Initialization in CURE . . . . . . . . . . . . . . . . . . . . 263 7.4.2 Completion of the CURE Algorithm . . . . . . . . . . . . 264 7.4.3 Exercises for Section 7.4 . . . . . . . . . . . . . . . . . . . 265 7.5 Clustering in Non-Euclidean Spaces . . . . . . . . . . . . . . . . 266 7.5.1 Representing Clusters in the GRGPF Algorithm . . . . . 266 7.5.2 Initializing the Cluster Tree . . . . . . . . . . . . . . . . . 267 7.5.3 Adding Points in the GRGPF Algorithm . . . . . . . . . 268 7.5.4 Splitting and Merging Clusters . . . . . . . . . . . . . . . 269 7.5.5 Exercises for Section 7.5 . . . . . . . . . . . . . . . . . . . 270 7.6 Clustering for Streams and Parallelism . . . . . . . . . . . . . . . 270 7.6.1 The Stream-Computing Model . . . . . . . . . . . . . . . 271 7.6.2 A Stream-Clustering Algorithm . . . . . . . . . . . . . . . 271 7.6.3 Initializing Buckets . . . . . . . . . . . . . . . . . . . . . . 272 7.6.4 Merging Buckets . . . . . . . . . . . . . . . . . . . . . . . 272 7.6.5 Answering Queries . . . . . . . . . . . . . . . . . . . . . . 275 7.6.6 Clustering in a Parallel Environment . . . . . . . . . . . . 275 7.6.7 Exercises for Section 7.6 . . . . . . . . . . . . . . . . . . . 276 7.7 Summary of Chapter 7 . . . . . . . . . . . . . . . . . . . . . . . . 276 7.8 References for Chapter 7 . . . . . . . . . . . . . . . . . . . . . . . 280

8 Advertising on the Web 281 8.1 Issues in On-Line Advertising . . . . . . . . . . . . . . . . . . . . 281 8.1.1 Advertising Opportunities . . . . . . . . . . . . . . . . . . 281 8.1.2 Direct Placement of Ads . . . . . . . . . . . . . . . . . . . 282 8.1.3 Issues for Display Ads . . . . . . . . . . . . . . . . . . . . 283 8.2 On-Line Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . 284 8.2.1 On-Line and Off-Line Algorithms . . . . . . . . . . . . . . 284 8.2.2 Greedy Algorithms . . . . . . . . . . . . . . . . . . . . . . 285 8.2.3 The Competitive Ratio . . . . . . . . . . . . . . . . . . . 286

xiv CONTENTS

8.2.4 Exercises for Section 8.2 . . . . . . . . . . . . . . . . . . . 286 8.3 The Matching Problem . . . . . . . . . . . . . . . . . . . . . . . . 287 8.3.1 Matches and Perfect Matches . . . . . . . . . . . . . . . . 287 8.3.2 The Greedy Algorithm for Maximal Matching . . . . . . . 288 8.3.3 Competitive Ratio for Greedy Matching . . . . . . . . . . 289 8.3.4 Exercises for Section 8.3 . . . . . . . . . . . . . . . . . . . 290 8.4 The Adwords Problem . . . . . . . . . . . . . . . . . . . . . . . . 290 8.4.1 History of Search Advertising . . . . . . . . . . . . . . . . 291 8.4.2 Definition of the Adwords Problem . . . . . . . . . . . . . 291 8.4.3 The Greedy Approach to the Adwords Problem . . . . . . 292 8.4.4 The Balance Algorithm . . . . . . . . . . . . . . . . . . . 293 8.4.5 A Lower Bound on Competitive Ratio for Balance . . . . 294 8.4.6 The Balance Algorithm with Many Bidders . . . . . . . . 296 8.4.7 The Generalized Balance Algorithm . . . . . . . . . . . . 297 8.4.8 Final Observations About the Adwords Problem . . . . . 298 8.4.9 Exercises for Section 8.4 . . . . . . . . . . . . . . . . . . . 299 8.5 Adwords Implementation . . . . . . . . . . . . . . . . . . . . . . 299 8.5.1 Matching Bids and Search Queries . . . . . . . . . . . . . 300 8.5.2 More Complex Matching Problems . . . . . . . . . . . . . 300 8.5.3 A Matching Algorithm for Documents and Bids . . . . . . 301 8.6 Summary of Chapter 8 . . . . . . . . . . . . . . . . . . . . . . . . 303 8.7 References for Chapter 8 . . . . . . . . . . . . . . . . . . . . . . . 305

9 Recommendation Systems 307 9.1 A Model for Recommendation Systems . . . . . . . . . . . . . . . 307 9.1.1 The Utility Matrix . . . . . . . . . . . . . . . . . . . . . . 308 9.1.2 The Long Tail . . . . . . . . . . . . . . . . . . . . . . . . 309 9.1.3 Applications of Recommendation Systems . . . . . . . . . 309 9.1.4 Populating the Utility Matrix . . . . . . . . . . . . . . . . 311 9.2 Content-Based Recommendations . . . . . . . . . . . . . . . . . . 312 9.2.1 Item Profiles . . . . . . . . . . . . . . . . . . . . . . . . . 312 9.2.2 Discovering Features of Documents . . . . . . . . . . . . . 313 9.2.3 Obtaining Item Features From Tags . . . . . . . . . . . . 314 9.2.4 Representing Item Profiles . . . . . . . . . . . . . . . . . . 315 9.2.5 User Profiles . . . . . . . . . . . . . . . . . . . . . . . . . 316 9.2.6 Recommending Items to Users Based on Content . . . . . 317 9.2.7 Classification Algorithms . . . . . . . . . . . . . . . . . . 318 9.2.8 Exercises for Section 9.2 . . . . . . . . . . . . . . . . . . . 320 9.3 Collaborative Filtering . . . . . . . . . . . . . . . . . . . . . . . . 321 9.3.1 Measuring Similarity . . . . . . . . . . . . . . . . . . . . . 322 9.3.2 The Duality of Similarity . . . . . . . . . . . . . . . . . . 324 9.3.3 Clustering Users and Items . . . . . . . . . . . . . . . . . 325 9.3.4 Exercises for Section 9.3 . . . . . . . . . . . . . . . . . . . 327 9.4 Dimensionality Reduction . . . . . . . . . . . . . . . . . . . . . . 328 9.4.1 UV-Decomposition . . . . . . . . . . . . . . . . . . . . . . 328

CONTENTS xv

9.4.2 Root-Mean-Square Error . . . . . . . . . . . . . . . . . . 329 9.4.3 Incremental Computation of a UV-Decomposition . . . . 330 9.4.4 Optimizing an Arbitrary Element . . . . . . . . . . . . . . 332 9.4.5 Building a Complete UV-Decomposition Algorithm . . . . 334 9.4.6 Exercises for Section 9.4 . . . . . . . . . . . . . . . . . . . 336 9.5 The Netflix Challenge . . . . . . . . . . . . . . . . . . . . . . . . 337 9.6 Summary of Chapter 9 . . . . . . . . . . . . . . . . . . . . . . . . 338 9.7 References for Chapter 9 . . . . . . . . . . . . . . . . . . . . . . . 340

10 Mining Social-Network Graphs 343 10.1 Social Networks as Graphs . . . . . . . . . . . . . . . . . . . . . . 343 10.1.1 What is a Social Network? . . . . . . . . . . . . . . . . . 344 10.1.2 Social Networks as Graphs . . . . . . . . . . . . . . . . . 344 10.1.3 Varieties of Social Networks . . . . . . . . . . . . . . . . . 346 10.1.4 Graphs With Several Node Types . . . . . . . . . . . . . 347 10.1.5 Exercises for Section 10.1 . . . . . . . . . . . . . . . . . . 348 10.2 Clustering of Social-Network Graphs . . . . . . . . . . . . . . . . 349 10.2.1 Distance Measures for Social-Network Graphs . . . . . . . 349 10.2.2 Applying Standard Clustering Methods . . . . . . . . . . 349 10.2.3 Betweenness. . . . . . . . . . . . . . . . . . . . . . . . . . 351 10.2.4 The Girvan-Newman Algorithm . . . . . . . . . . . . . . . 351 10.2.5 Using Betweenness to Find Communities . . . . . . . . . 354 10.2.6 Exercises for Section 10.2 . . . . . . . . . . . . . . . . . . 356 10.3 Direct Discovery of Communities . . . . . . . . . . . . . . . . . . 357 10.3.1 Finding Cliques . . . . . . . . . . . . . . . . . . . . . . . . 357 10.3.2 Complete Bipartite Graphs . . . . . . . . . . . . . . . . . 357 10.3.3 Finding Complete Bipartite Subgraphs . . . . . . . . . . . 358 10.3.4 Why Complete Bipartite Graphs Must Exist . . . . . . . 359 10.3.5 Exercises for Section 10.3 . . . . . . . . . . . . . . . . . . 361 10.4 Partitioning of Graphs . . . . . . . . . . . . . . . . . . . . . . . . 361 10.4.1 What Makes a Good Partition? . . . . . . . . . . . . . . . 362 10.4.2 Normalized Cuts . . . . . . . . . . . . . . . . . . . . . . . 362 10.4.3 Some Matrices That Describe Graphs . . . . . . . . . . . 363 10.4.4 Eigenvalues of the Laplacian Matrix . . . . . . . . . . . . 364 10.4.5 Alternative Partitioning Methods . . . . . . . . . . . . . . 367 10.4.6 Exercises for Section 10.4 . . . . . . . . . . . . . . . . . . 368 10.5 Finding Overlapping Communities . . . . . . . . . . . . . . . . . 369 10.5.1 The Nature of Communities . . . . . . . . . . . . . . . . . 369 10.5.2 Maximum-Likelihood Estimation . . . . . . . . . . . . . . 369 10.5.3 The Affiliation-Graph Model . . . . . . . . . . . . . . . . 371 10.5.4 Avoiding the Use of Discrete Membership Changes . . . . 374 10.5.5 Exercises for Section 10.5 . . . . . . . . . . . . . . . . . . 375 10.6 Simrank . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 376 10.6.1 Random Walkers on a Social Graph . . . . . . . . . . . . 376 10.6.2 Random Walks with Restart . . . . . . . . . . . . . . . . 377

xvi CONTENTS

10.6.3 Exercises for Section 10.6 . . . . . . . . . . . . . . . . . . 380 10.7 Counting Triangles . . . . . . . . . . . . . . . . . . . . . . . . . . 380 10.7.1 Why Count Triangles? . . . . . . . . . . . . . . . . . . . . 380 10.7.2 An Algorithm for Finding Triangles . . . . . . . . . . . . 381 10.7.3 Optimality of the Triangle-Finding Algorithm . . . . . . . 382 10.7.4 Finding Triangles Using MapReduce . . . . . . . . . . . . 383 10.7.5 Using Fewer Reduce Tasks . . . . . . . . . . . . . . . . . . 384 10.7.6 Exercises for Section 10.7 . . . . . . . . . . . . . . . . . . 385 10.8 Neighborhood Properties of Graphs . . . . . . . . . . . . . . . . . 386 10.8.1 Directed Graphs and Neighborhoods . . . . . . . . . . . . 386 10.8.2 The Diameter of a Graph . . . . . . . . . . . . . . . . . . 388 10.8.3 Transitive Closure and Reachability . . . . . . . . . . . . 389 10.8.4 Transitive Closure Via MapReduce . . . . . . . . . . . . . 390 10.8.5 Smart Transitive Closure . . . . . . . . . . . . . . . . . . 392 10.8.6 Transitive Closure by Graph Reduction . . . . . . . . . . 393 10.8.7 Approximating the Sizes of Neighborhoods . . . . . . . . 395 10.8.8 Exercises for Section 10.8 . . . . . . . . . . . . . . . . . . 397 10.9 Summary of Chapter 10 . . . . . . . . . . . . . . . . . . . . . . . 398 10.10References for Chapter 10 . . . . . . . . . . . . . . . . . . . . . . 402

11 Dimensionality Reduction 405 11.1 Eigenvalues and Eigenvectors of Symmetric Matrices . . . . . . . 406 11.1.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . 406 11.1.2 Computing Eigenvalues and Eigenvectors . . . . . . . . . 407 11.1.3 Finding Eigenpairs by Power Iteration . . . . . . . . . . . 408 11.1.4 The Matrix of Eigenvectors . . . . . . . . . . . . . . . . . 411 11.1.5 Exercises for Section 11.1 . . . . . . . . . . . . . . . . . . 411 11.2 Principal-Component Analysis . . . . . . . . . . . . . . . . . . . 412 11.2.1 An Illustrative Example . . . . . . . . . . . . . . . . . . . 413 11.2.2 Using Eigenvectors for Dimensionality Reduction . . . . . 416 11.2.3 The Matrix of Distances . . . . . . . . . . . . . . . . . . . 417 11.2.4 Exercises for Section 11.2 . . . . . . . . . . . . . . . . . . 418 11.3 Singular-Value Decomposition . . . . . . . . . . . . . . . . . . . . 418 11.3.1 Definition of SVD . . . . . . . . . . . . . . . . . . . . . . 418 11.3.2 Interpretation of SVD . . . . . . . . . . . . . . . . . . . . 420 11.3.3 Dimensionality Reduction Using SVD . . . . . . . . . . . 422 11.3.4 Why Zeroing Low Singular Values Works . . . . . . . . . 423 11.3.5 Querying Using Concepts . . . . . . . . . . . . . . . . . . 425 11.3.6 Computing the SVD of a Matrix . . . . . . . . . . . . . . 426 11.3.7 Exercises for Section 11.3 . . . . . . . . . . . . . . . . . . 427 11.4 CUR Decomposition . . . . . . . . . . . . . . . . . . . . . . . . . 428 11.4.1 Definition of CUR . . . . . . . . . . . . . . . . . . . . . . 429 11.4.2 Choosing Rows and Columns Properly . . . . . . . . . . . 430 11.4.3 Constructing the Middle Matrix . . . . . . . . . . . . . . 431 11.4.4 The Complete CUR Decomposition . . . . . . . . . . . . 432

CONTENTS xvii

11.4.5 Eliminating Duplicate Rows and Columns . . . . . . . . . 433 11.4.6 Exercises for Section 11.4 . . . . . . . . . . . . . . . . . . 434 11.5 Summary of Chapter 11 . . . . . . . . . . . . . . . . . . . . . . . 434 11.6 References for Chapter 11 . . . . . . . . . . . . . . . . . . . . . . 436

12 Large-Scale Machine Learning 439 12.1 The Machine-Learning Model . . . . . . . . . . . . . . . . . . . . 440 12.1.1 Training Sets . . . . . . . . . . . . . . . . . . . . . . . . . 440 12.1.2 Some Illustrative Examples . . . . . . . . . . . . . . . . . 440 12.1.3 Approaches to Machine Learning . . . . . . . . . . . . . . 443 12.1.4 Machine-Learning Architecture . . . . . . . . . . . . . . . 444 12.1.5 Exercises for Section 12.1 . . . . . . . . . . . . . . . . . . 447 12.2 Perceptrons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 447 12.2.1 Training a Perceptron with Zero Threshold . . . . . . . . 447 12.2.2 Convergence of Perceptrons . . . . . . . . . . . . . . . . . 451 12.2.3 The Winnow Algorithm . . . . . . . . . . . . . . . . . . . 451 12.2.4 Allowing the Threshold to Vary . . . . . . . . . . . . . . . 453 12.2.5 Multiclass Perceptrons . . . . . . . . . . . . . . . . . . . . 455 12.2.6 Transforming the Training Set . . . . . . . . . . . . . . . 456 12.2.7 Problems With Perceptrons . . . . . . . . . . . . . . . . . 457 12.2.8 Parallel Implementation of Perceptrons . . . . . . . . . . 458 12.2.9 Exercises for Section 12.2 . . . . . . . . . . . . . . . . . . 459 12.3 Support-Vector Machines . . . . . . . . . . . . . . . . . . . . . . 461 12.3.1 The Mechanics of an SVM . . . . . . . . . . . . . . . . . . 461 12.3.2 Normalizing the Hyperplane . . . . . . . . . . . . . . . . . 462 12.3.3 Finding Optimal Approximate Separators . . . . . . . . . 464 12.3.4 SVM Solutions by Gradient Descent . . . . . . . . . . . . 467 12.3.5 Stochastic Gradient Descent . . . . . . . . . . . . . . . . . 470 12.3.6 Parallel Implementation of SVM . . . . . . . . . . . . . . 471 12.3.7 Exercises for Section 12.3 . . . . . . . . . . . . . . . . . . 472 12.4 Learning from Nearest Neighbors . . . . . . . . . . . . . . . . . . 472 12.4.1 The Framework for Nearest-Neighbor Calculations . . . . 473 12.4.2 Learning with One Nearest Neighbor . . . . . . . . . . . . 473 12.4.3 Learning One-Dimensional Functions . . . . . . . . . . . . 474 12.4.4 Kernel Regression . . . . . . . . . . . . . . . . . . . . . . 477 12.4.5 Dealing with High-Dimensional Euclidean Data . . . . . . 477 12.4.6 Dealing with Non-Euclidean Distances . . . . . . . . . . . 479 12.4.7 Exercises for Section 12.4 . . . . . . . . . . . . . . . . . . 479 12.5 Comparison of Learning Methods . . . . . . . . . . . . . . . . . . 480 12.6 Summary of Chapter 12 . . . . . . . . . . . . . . . . . . . . . . . 481 12.7 References for Chapter 12 . . . . . . . . . . . . . . . . . . . . . . 483

xviii CONTENTS

Chapter 1

Data Mining

In this intoductory chapter we begin with the essence of data mining and a dis- cussion of how data mining is treated by the various disciplines that contribute to this field. We cover “Bonferroni’s Principle,” which is really a warning about overusing the ability to mine data. This chapter is also the place where we summarize a few useful ideas that are not data mining but are useful in un- derstanding some important data-mining concepts. These include the TF.IDF measure of word importance, behavior of hash functions and indexes, and iden- tities involving e, the base of natural logarithms. Finally, we give an outline of the topics covered in the balance of the book.

1.1 What is Data Mining?

The most commonly accepted definition of “data mining” is the discovery of “models” for data. A “model,” however, can be one of several things. We mention below the most important directions in modeling.

1.1.1 Statistical Modeling

Statisticians were the first to use the term “data mining.” Originally, “data mining” or “data dredging” was a derogatory term referring to attempts to extract information that was not supported by the data. Section 1.2 illustrates the sort of errors one can make by trying to extract what really isn’t in the data. Today, “data mining” has taken on a positive meaning. Now, statisticians view data mining as the construction of a statistical model, that is, an underlying distribution from which the visible data is drawn.

Example 1.1: Suppose our data is a set of numbers. This data is much simpler than data that would be data-mined, but it will serve as an example. A statistician might decide that the data comes from a Gaussian distribution and use a formula to compute the most likely parameters of this Gaussian. The mean

1

2 CHAPTER 1. DATA MINING

and standard deviation of this Gaussian distribution completely characterize the distribution and would become the model of the data. ✷

1.1.2 Machine Learning

There are some who regard data mining as synonymous with machine learning. There is no question that some data mining appropriately uses algorithms from machine learning. Machine-learning practitioners use the data as a training set, to train an algorithm of one of the many types used by machine-learning prac- titioners, such as Bayes nets, support-vector machines, decision trees, hidden Markov models, and many others.

There are situations where using data in this way makes sense. The typical case where machine learning is a good approach is when we have little idea of what we are looking for in the data. For example, it is rather unclear what it is about movies that makes certain movie-goers like or dislike it. Thus, in answering the “Netflix challenge” to devise an algorithm that predicts the ratings of movies by users, based on a sample of their responses, machine- learning algorithms have proved quite successful. We shall discuss a simple form of this type of algorithm in Section 9.4.

On the other hand, machine learning has not proved successful in situations where we can describe the goals of the mining more directly. An interesting case in point is the attempt by WhizBang! Labs1 to use machine learning to locate people’s resumes on the Web. It was not able to do better than algorithms designed by hand to look for some of the obvious words and phrases that appear in the typical resume. Since everyone who has looked at or written a resume has a pretty good idea of what resumes contain, there was no mystery about what makes a Web page a resume. Thus, there was no advantage to machine-learning over the direct design of an algorithm to discover resumes.

1.1.3 Computational Approaches to Modeling

More recently, computer scientists have looked at data mining as an algorithmic problem. In this case, the model of the data is simply the answer to a complex query about it. For instance, given the set of numbers of Example 1.1, we might compute their average and standard deviation. Note that these values might not be the parameters of the Gaussian that best fits the data, although they will almost certainly be very close if the size of the data is large.

There are many different approaches to modeling data. We have already mentioned the possibility of constructing a statistical process whereby the data could have been generated. Most other approaches to modeling can be described as either

1. Summarizing the data succinctly and approximately, or

1This startup attempted to use machine learning to mine large-scale data, and hired many of the top machine-learning people to do so. Unfortunately, it was not able to survive.

1.1. WHAT IS DATA MINING? 3

2. Extracting the most prominent features of the data and ignoring the rest.

We shall explore these two approaches in the following sections.

1.1.4 Summarization

One of the most interesting forms of summarization is the PageRank idea, which made Google successful and which we shall cover in Chapter 5. In this form of Web mining, the entire complex structure of the Web is summarized by a single number for each page. This number, the “PageRank” of the page, is (oversimplifying somewhat) the probability that a random walker on the graph would be at that page at any given time. The remarkable property this ranking has is that it reflects very well the “importance” of the page – the degree to which typical searchers would like that page returned as an answer to their search query.

Another important form of summary – clustering – will be covered in Chap- ter 7. Here, data is viewed as points in a multidimensional space. Points that are “close” in this space are assigned to the same cluster. The clusters themselves are summarized, perhaps by giving the centroid of the cluster and the average distance from the centroid of points in the cluster. These cluster summaries become the summary of the entire data set.

Example 1.2: A famous instance of clustering to solve a problem took place long ago in London, and it was done entirely without computers.2 The physician John Snow, dealing with a Cholera outbreak plotted the cases on a map of the city. A small illustration suggesting the process is shown in Fig. 1.1.

Figure 1.1: Plotting cholera cases on a map of London

2See http://en.wikipedia.org/wiki/1854

Broad

Street

Street

cholera

cholera

cholera

outbreak.

outbreak.

outbreak.

outbreak.

4 CHAPTER 1. DATA MINING

The cases clustered around some of the intersections of roads. These inter- sections were the locations of wells that had become contaminated; people who lived nearest these wells got sick, while people who lived nearer to wells that had not been contaminated did not get sick. Without the ability to cluster the data, the cause of Cholera would not have been discovered. ✷

1.1.5 Feature Extraction

The typical feature-based model looks for the most extreme examples of a phe- nomenon and represents the data by these examples. If you are familiar with Bayes nets, a branch of machine learning and a topic we do not cover in this book, you know how a complex relationship between objects is represented by finding the strongest statistical dependencies among these objects and using only those in representing all statistical connections. Some of the important kinds of feature extraction from large-scale data that we shall study are:

1. Frequent Itemsets. This model makes sense for data that consists of “bas- kets” of small sets of items, as in the market-basket problem that we shall discuss in Chapter 6. We look for small sets of items that appear together in many baskets, and these “frequent itemsets” are the characterization of the data that we seek. The original application of this sort of mining was true market baskets: the sets of items, such as hamburger and ketchup, that people tend to buy together when checking out at the cash register of a store or super market.

2. Similar Items. Often, your data looks like a collection of sets, and the objective is to find pairs of sets that have a relatively large fraction of their elements in common. An example is treating customers at an on- line store like Amazon as the set of items they have bought. In order for Amazon to recommend something else they might like, Amazon can look for “similar” customers and recommend something many of these customers have bought. This process is called “collaborative filtering.” If customers were single-minded, that is, they bought only one kind of thing, then clustering customers might work. However, since customers tend to have interests in many different things, it is more useful to find, for each customer, a small number of other customers who are similar in their tastes, and represent the data by these connections. We discuss similarity in Chapter 3.

1.2 Statistical Limits on Data Mining

A common sort of data-mining problem involves discovering unusual events hidden within massive amounts of data. This section is a discussion of the problem, including “Bonferroni’s Principle,” a warning against overzealous use of data mining.

1.2. STATISTICAL LIMITS ON DATA MINING 5

1.2.1 Total Information Awareness

Following the terrorist attack of Sept. 11, 2001, it was noticed that there were four people enrolled in different flight schools, learning how to pilot commercial aircraft, although they were not affiliated with any airline. It was conjectured that the information needed to predict and foil the attack was available in data, but that there was then no way to examine the data and detect suspi- cious events. The response was a program called TIA, or Total Information Awareness, which was intended to mine all the data it could find, including credit-card receipts, hotel records, travel data, and many other kinds of infor- mation in order to track terrorist activity. TIA naturally caused great concern among privacy advocates, and the project was eventually killed by Congress. It is not the purpose of this book to discuss the difficult issue of the privacy- security tradeoff. However, the prospect of TIA or a system like it does raise many technical questions about its feasibility.

The concern raised by many is that if you look at so much data, and you try to find within it activities that look like terrorist behavior, are you not going to find many innocent activities – or even illicit activities that are not terrorism – that will result in visits from the police and maybe worse than just a visit? The answer is that it all depends on how narrowly you define the activities that you look for. Statisticians have seen this problem in many guises and have a theory, which we introduce in the next section.

1.2.2 Bonferroni’s Principle

Suppose you have a certain amount of data, and you look for events of a cer- tain type within that data. You can expect events of this type to occur, even if the data is completely random, and the number of occurrences of these events will grow as the size of the data grows. These occurrences are “bogus,” in the sense that they have no cause other than that random data will always have some number of unusual features that look significant but aren’t. A theorem of statistics, known as the Bonferroni correction gives a statistically sound way to avoid most of these bogus positive responses to a search through the data. Without going into the statistical details, we offer an informal version, Bon- ferroni’s principle, that helps us avoid treating random occurrences as if they were real. Calculate the expected number of occurrences of the events you are looking for, on the assumption that data is random. If this number is signifi- cantly larger than the number of real instances you hope to find, then you must expect almost anything you find to be bogus, i.e., a statistical artifact rather than evidence of what you are looking for. This observation is the informal statement of Bonferroni’s principle.

In a situation like searching for terrorists, where we expect that there are few terrorists operating at any one time, Bonferroni’s principle says that we may only detect terrorists by looking for events that are so rare that they are unlikely to occur in random data. We shall give an extended example in the

6 CHAPTER 1. DATA MINING

next section.

1.2.3 An Example of Bonferroni’s Principle

Suppose there are believed to be some “evil-doers” out there, and we want to detect them. Suppose further that we have reason to believe that periodi- cally, evil-doers gather at a hotel to plot their evil. Let us make the following assumptions about the size of the problem:

1. There are one billion people who might be evil-doers.

2. Everyone goes to a hotel one day in 100.

3. A hotel holds 100 people. Hence, there are 100,000 hotels – enough to

hold the 1% of a billion people who visit a hotel on any given day.

4. We shall examine hotel records for 1000 days.

To find evil-doers in this data, we shall look for people who, on two different days, were both at the same hotel. Suppose, however, that there really are no evil-doers. That is, everyone behaves at random, deciding with probability 0.01 to visit a hotel on any given day, and if so, choosing one of the 105 hotels at random. Would we find any pairs of people who appear to be evil-doers?

We can do a simple approximate calculation as follows. The probability of any two people both deciding to visit a hotel on any given day is .0001. The chance that they will visit the same hotel is this probability divided by 105, the number of hotels. Thus, the chance that they will visit the same hotel on one given day is 10−9. The chance that they will visit the same hotel on two different given days is the square of this number, 10−18. Note that the hotels can be different on the two days.

Now, we must consider how many events will indicate evil-doing. An “event” in this sense is a pair of people and a pair of days, such that the two people were that for at the large same n, hotel (n2on each of the two days. To simplify the arithmetic, note follows. ) Thus, the number is about of pairs n2/2. of We people shall is use (102

this 9approximation in what ) of pairs of days is (1000

2

) = 5 × 105. = 5 × 1017. The expected number of events The number that look like evil-doing is the product of the number of pairs of people, the number of pairs of days, and the probability that any one pair of people and pair of days is an instance of the behavior we are looking for. That number is

5 × 1017 × 5 × 105 × 10−18 = 250,000

That is, there will be a quarter of a million pairs of people who look like evil- doers, even though they are not.

Now, suppose there really are 10 pairs of evil-doers out there. The police will need to investigate a quarter of a million other pairs in order to find the real evil-doers. In addition to the intrusion on the lives of half a million innocent

1.3. THINGS USEFUL TO KNOW 7

people, the work involved is sufficiently great that this approach to finding evil-doers is probably not feasible.

1.2.4 Exercises for Section 1.2

Exercise 1.2.1: Using the information from Section 1.2.3, what would be the number of suspected pairs if the following changes were made to the data (and all other numbers remained as they were in that section)?

(a) The number of days of observation was raised to 2000.

(b) The number of people observed was raised to 2 billion (and there were

therefore 200,000 hotels).

(c) We only reported a pair as suspect if they were at the same hotel at the

same time on three different days.

! Exercise 1.2.2: Suppose we have information about the supermarket pur- chases of 100 million people. Each person goes to the supermarket 100 times in a year and buys 10 of the 1000 items that the supermarket sells. We believe that a pair of terrorists will buy exactly the same set of 10 items (perhaps the ingredients for a bomb?) at some time during the year. If we search for pairs of people who have bought the same set of items, would we expect that any such people found were truly terrorists?3

1.3 Things Useful to Know

In this section, we offer brief introductions to subjects that you may or may not have seen in your study of other courses. Each will be useful in the study of data mining. They include:

1. The TF.IDF measure of word importance.

2. Hash functions and their use.

3. Secondary storage (disk) and its effect on running time of algorithms.

4. The base e of natural logarithms and identities involving that constant.

5. Power laws.

3That is, assume our hypothesis that terrorists will surely buy a set of 10 items in common at some time during the year. We don’t want to address the matter of whether or not terrorists would necessarily do so.

8 CHAPTER 1. DATA MINING

1.3.1 Importance of Words in Documents

In several applications of data mining, we shall be faced with the problem of categorizing documents (sequences of words) by their topic. Typically, topics are identified by finding the special words that characterize documents about that topic. For instance, articles about baseball would tend to have many occurrences of words like “ball,” “bat,” “pitch,”, “run,” and so on. Once we have classified documents to determine they are about baseball, it is not hard to notice that words such as these appear unusually frequently. However, until we have made the classification, it is not possible to identify these words as characteristic.

Thus, classification often starts by looking at documents, and finding the significant words in those documents. Our first guess might be that the words appearing most frequently in a document are the most significant. However, that intuition is exactly opposite of the truth. The most frequent words will most surely be the common words such as “the” or “and,” which help build ideas but do not carry any significance themselves. In fact, the several hundred most common words in English (called stop words) are often removed from documents before any attempt to classify them.

In fact, the indicators of the topic are relatively rare words. However, not all rare words are equally useful as indicators. There are certain words, for example “notwithstanding” or “albeit,” that appear rarely in a collection of documents, yet do not tell us anything useful. On the other hand, a word like “chukker” is probably equally rare, but tips us off that the document is about the sport of polo. The difference between rare words that tell us something and those that do not has to do with the concentration of the useful words in just a few documents. That is, the presence of a word like “albeit” in a document does not make it terribly more likely that it will appear multiple times. However, if an article mentions “chukker” once, it is likely to tell us what happened in the “first chukker,” then the “second chukker,” and so on. That is, the word is likely to be repeated if it appears at all.

The formal measure of how concentrated into relatively few documents are the occurrences of a given word is called TF.IDF (Term Frequency times In- verse Document Frequency). It is normally computed as follows. Suppose we have a collection of N documents. Define fij to be the frequency (number of occurrences) of term (word) i in document j. Then, define the term frequency TF ij to be:

TF ij = maxfij

k fkj

That is, the term frequency of term i in document j is fij normalized by dividing it by the maximum number of occurrences of any term (perhaps excluding stop words) in the same document. Thus, the most frequent term in document j gets a TF of 1, and other terms get fractions as their term frequency for this document.

The IDF for a term is defined as follows. Suppose term i appears in ni

1.3. THINGS USEFUL TO KNOW 9

of the N documents in the collection. Then IDF i = log2(N/ni). The TF.IDF score for term i in document j is then defined to be TF ij × IDF i. The terms with the highest TF.IDF score are often the terms that best characterize the topic of the document.

Example 1.3: Suppose our repository consists of 220 = 1,048,576 documents. Suppose word w appears in 210 = 1024 of these documents. Then IDF w = log2(220/210) = log 2(210) = 10. Consider a document j in which w appears 20 times, and that is the maximum number of times in which any word appears (perhaps after eliminating stop words). Then TF wj = 1, and the TF.IDF score for w in document j is 10.

Suppose that in document k, word w appears once, while the maximum number of occurrences of any word in this document is 20. Then TF wk = 1/20, and the TF.IDF score for w in document k is 1/2. ✷

1.3.2 Hash Functions

The reader has probably heard of hash tables, and perhaps used them in Java classes or similar packages. The hash functions that make hash tables feasible are also essential components in a number of data-mining algorithms, where the hash table takes an unfamiliar form. We shall review the basics here.

First, a hash function h takes a hash-key value as an argument and produces a bucket number as a result. The bucket number is an integer, normally in the range 0 to B − 1, where B is the number of buckets. Hash-keys can be of any type. There is an intuitive property of hash functions that they “randomize” hash-keys. To be precise, if hash-keys are drawn randomly from a reasonable population of possible hash-keys, then h will send approximately equal numbers of hash-keys to each of the B buckets. It would be impossible to do so if, for example, the population of possible hash-keys were smaller than B. Such a population would not be “reasonable.” However, there can be more subtle rea- sons why a hash function fails to achieve an approximately uniform distribution into buckets.

Example 1.4: Suppose hash-keys are positive integers. A common and simple hash function is to pick h(x) = x mod B, that is, the remainder when x is divided by B. That choice works fine if our population of hash-keys is all positive integers. 1/Bth of the integers will be assigned to each of the buckets. However, suppose our population is the even integers, and B = 10. Then only buckets 0, 2, 4, 6, and 8 can be the value of h(x), and the hash function is distinctly nonrandom in its behavior. On the other hand, if we picked B = 11, then we would find that 1/11th of the even integers get sent to each of the 11 buckets, so the hash function would work very well. ✷

The generalization of Example 1.4 is that when hash-keys are integers, chos- ing B so it has any common factor with all (or even most of) the possible hash- keys will result in nonrandom distribution into buckets. Thus, it is normally

10 CHAPTER 1. DATA MINING

preferred that we choose B to be a prime. That choice reduces the chance of nonrandom behavior, although we still have to consider the possibility that all hash-keys have B as a factor. Of course there are many other types of hash functions not based on modular arithmetic. We shall not try to summarize the options here, but some sources of information will be mentioned in the bibliographic notes.

What if hash-keys are not integers? In a sense, all data types have values that are composed of bits, and sequences of bits can always be interpreted as in- tegers. However, there are some simple rules that enable us to convert common types to integers. For example, if hash-keys are strings, convert each character to its ASCII or Unicode equivalent, which can be interpreted as a small inte- ger. Sum the integers before dividing by B. As long as B is smaller than the typical sum of character codes for the population of strings, the distribution into buckets will be relatively uniform. If B is larger, then we can partition the characters of a string into groups of several characters each. Treat the concate- nation of the codes for the characters of a group as a single integer. Sum the integers associated with all the groups of a string, and divide by B as before. For instance, if B is around a billion, or 230, then grouping characters four at a time will give us 32-bit integers. The sum of several of these will distribute fairly evenly into a billion buckets.

For more complex data types, we can extend the idea used for converting strings to integers, recursively.

• For a type that is a record, each of whose components has its own type, recursively convert the value of each component to an integer, using the algorithm appropriate for the type of that component. Sum the integers for the components, and convert the integer sum to buckets by dividing by B.

• For a type that is an array, set, or bag of elements of some one type, convert the values of the elements’ type to integers, sum the integers, and divide by B.

1.3.3 Indexes

An index is a data structure that makes it efficient to retrieve objects given the value of one or more elements of those objects. The most common situation is one where the objects are records, and the index is on one of the fields of that record. Given a value v for that field, the index lets us retrieve all the records with value v in that field. For example, we could have a file of (name, address, phone) triples, and an index on the phone field. Given a phone number, the index allows us to find quickly the record or records with that phone number.

There are many ways to implement indexes, and we shall not attempt to survey the matter here. The bibliographic notes give suggestions for further reading. However, a hash table is one simple way to build an index. The field

1.3. THINGS USEFUL TO KNOW 11

or fields on which the index is based form the hash-key for a hash function. Records have the hash function applied to value of the hash-key, and the record itself is placed in the bucket whose number is determined by the hash function. The bucket could be a list of records in main-memory, or a disk block, for example.

Then, given a hash-key value, we can hash it, find the bucket, and need to search only that bucket to find the records with that value for the hash-key. If we choose the number of buckets B to be comparable to the number of records in the file, then there will be relatively few records in any bucket, and the search of a bucket takes little time.

0

*h* (800−555−1212)

17Records with *h*(phone) = 17

*B*−1 Array of

bucket headers

Sally Jones Maple St 800−555−1212

**.... ..**Figure 1.2: A hash table used as an index; phone numbers are hashed to buckets, and the entire record is placed in the bucket whose number is the hash value of the phone

Example 1.5: Figure 1.2 suggests what a main-memory index of records with name, address, and phone fields might look like. Here, the index is on the phone field, and buckets are linked lists. We show the phone 800-555-1212 hashed to bucket number 17. There is an array of bucket headers, whose ith element is the head of a linked list for the bucket numbered i. We show expanded one of the elements of the linked list. It contains a record with name, address, and phone fields. This record is in fact one with the phone number 800-555-1212. Other records in that bucket may or may not have this phone number. We only know that whatever phone number they have is a phone that hashes to 17. ✷

1.3.4 Secondary Storage

It is important, when dealing with large-scale data, that we have a good un- derstanding of the difference in time taken to perform computations when the data is initially on disk, as opposed to the time needed if the data is initially in

12 CHAPTER 1. DATA MINING

main memory. The physical characteristics of disks is another subject on which we could say much, but shall say only a little and leave the interested reader to follow the bibliographic notes.

Disks are organized into blocks, which are the minimum units that the oper- ating system uses to move data between main memory and disk. For example, the Windows operating system uses blocks of 64K bytes (i.e., 216 = 65,536 bytes to be exact). It takes approximately ten milliseconds to access (move the disk head to the track of the block and wait for the block to rotate under the head) and read a disk block. That delay is at least five orders of magnitude (a factor of 105) slower than the time taken to read a word from main memory, so if all we want to do is access a few bytes, there is an overwhelming benefit to having data in main memory. In fact, if we want to do something simple to every byte of a disk block, e.g., treat the block as a bucket of a hash table and search for a particular value of the hash-key among all the records in that bucket, then the time taken to move the block from disk to main memory will be far larger than the time taken to do the computation.

By organizing our data so that related data is on a single cylinder (the collection of blocks reachable at a fixed radius from the center of the disk, and therefore accessible without moving the disk head), we can read all the blocks on the cylinder into main memory in considerably less than 10 milliseconds per block. You can assume that a disk cannot transfer data to main memory at more than a hundred million bytes per second, no matter how that data is organized. That is not a problem when your dataset is a megabyte. But a dataset of a hundred gigabytes or a terabyte presents problems just accessing it, let alone doing anything useful with it.

1.3.5 The Base of Natural Logarithms

The constant e = 2.7182818··· has a number of useful special properties. In particular, e expression for is the limit x = 1,2,3,4 of are (1 + approximately x1

)x as x goes to infinity. 2,2.25,2.37,2.44, The values of this so you should find it easy to believe that the limit of this series is around 2.72.

Some algebra lets us obtain approximations to many seemingly complex expressions. Consider (1 +a)b, where a is small. We can rewrite the expression as (1+a)(1/a)(ab). Then substitute a = 1/x and which is ((1 + x1 )x)ab

1/a = x, so we have (1+ x1

)x(ab),

Since to a is assumed small, the limiting value of x is e. We large, so the subexpression can thus approximate (1 + (1 a)b + as x1

)ex abwill .

be close

Similar identities hold when a is negative. That is, the limit as x goes to infinity holds of even (1 − when x1

)a x is is a 1/e. small It negative follows that the number. approximation (1 + a)b Put another way, (1 − = eab a)b is approximately e−ab when a is small and b is large.

1.3. THINGS USEFUL TO KNOW 13

Some That is, x is large, other ex = the above ∑useful ∞i=0 xseries iapproximations /i!, or ex converges follow from the Taylor expansion of ex. =1+ x + x2/2 + x3/6 + x4/24 + ···. When slowly, although it does converge because n! grows faster than xn for any constant x. However, when x is small, either positive or negative, the series converges rapidly, and only a few terms are necessary to get a good approximation.

Example 1.6: Let x = 1/2. Then

e1/2 =1+ 1

384 + ···

or approximately e1/2 = 1.64844.

Let x = −1. Then

e−1 = 1 − 1 + 1

5040 + ···

or approximately e−1 = 0.36786. ✷

1.3.6 Power Laws

There are many phenomena that relate two variables by a power law, that is, a linear relationship between the logarithms of the variables. Figure 1.3 suggests such a relationship. If x is the horizontal axis and y is the vertical axis, then the relationship is log10 y = 6 − 2 log10 x.

10,000,000

1,000,000

100,000

10,000

1000

100

10

11 10 100 1000 10,000

Figure 1.3: A power law with a slope of −2

2 − 1

6 + 1

2 + 1

24 − 1

8 + 1

48 + 1

120 + 1

720 − 1

14 CHAPTER 1. DATA MINING

Example 1.7: We might examine book sales at Amazon.com, and let x rep- resent the rank of books by sales. Then y is the number of sales of the xth best-selling book over some period. The implication of the graph of Fig. 1.3 would be that the best-selling book sold 1,000,000 copies, the 10th best-selling book sold 10,000 copies, the 100th best-selling book sold 100 copies, and so on for all ranks between these numbers and beyond. The implication that above rank 1000 the sales are a fraction of a book is too extreme, and we would in fact expect the line to flatten out for ranks much higher than 1000. ✷

The general form of a power law relating x and y is logy = b+alogx. If we raise the base of the logarithm (which doesn’t actually matter), say e, to the values on both sides of this equation, we get y = ebea log x = ebxa. Since eb is just “some constant,” let us replace it by constant c. Thus, a power law can be written as y = cxa for some constants a and c.

Example 1.8: In Fig. 1.3 we see that when x = 1, y = 106, and when x = 1000, y = 1. Making the first substitution, we see 106 = c. The second substitution gives us 1 = c(1000)a. Since we now know c = 106, the second equation gives us 1 = 106(1000)a, from which we see a = −2. That is, the law expressed by Fig. 1.3 is y = 106x−2, or y = 106/x2. ✷

We shall meet in this book many ways that power laws govern phenomena. Here are some examples:

1. Node Degrees in the Web Graph: Order all pages by the number of in- links to that page. Let x be the position of a page in this ordering, and let y be the number of in-links to the xth page. Then y as a function of x looks very much like Fig. 1.3. The exponent a is slightly larger than the −2 shown there; it has been found closer to 2.1.

The Matthew Effect

Often, the existence of power laws with values of the exponent higher than 1 are explained by the Matthew effect. In the biblical Book of Matthew, there is a verse about “the rich get richer.” Many phenomena exhibit this behavior, where getting a high value of some property causes that very property to increase. For example, if a Web page has many links in, then people are more likely to find the page and may choose to link to it from one of their pages as well. As another example, if a book is selling well on Amazon, then it is likely to be advertised when customers go to the Amazon site. Some of these people will choose to buy the book as well, thus increasing the sales of this book.

1.4. OUTLINE OF THE BOOK 15

2. Sales of Products: Order products, say books at Amazon.com, by their sales over the past year. Let y be the number of sales of the xth most pop- ular book. Again, the function y(x) will look something like Fig. 1.3. we shall discuss the consequences of this distribution of sales in Section 9.1.2, where we take up the matter of the “long tail.”

3. Sizes of Web Sites: Count the number of pages at Web sites, and order sites by the number of their pages. Let y be the number of pages at the xth site. Again, the function y(x) follows a power law.

4. Zipf’s Law: This power law originally referred to the frequency of words in a collection of documents. If you order words by frequency, and let y be the number of times the xth word in the order appears, then you get a power law, although with a much shallower slope than that of Fig. 1.3. Zipf’s observation was that y = cx−1/2. Interestingly, a number of other kinds of data follow this particular power law. For example, if we order states in the US by population and let y be the population of the xth most populous state, then x and y obey Zipf’s law approximately.

1.3.7 Exercises for Section 1.3

Exercise 1.3.1: Suppose there is a repository of ten million documents. What (to the nearest integer) is the IDF for a word that appears in (a) 40 documents (b) 10,000 documents?

Exercise 1.3.2: Suppose there is a repository of ten million documents, and word w appears in 320 of them. In a particular document d, the maximum number of occurrences of a word is 15. Approximately what is the TF.IDF score for w if that word appears (a) once (b) five times?

! Exercise 1.3.3: Suppose hash-keys are drawn from the population of all non- negative integers that are multiples of some constant c, and hash function h(x) is x mod 15. For what values of c will h be a suitable hash function, i.e., a large random choice of hash-keys will be divided roughly equally into buckets?

Exercise 1.3.4: In terms of e, give approximations to

(a) (1.01)500 (b) (1.05)1000 (c) (0.9)40

Exercise 1.3.5: Use the Taylor expansion of ex to compute, to three decimal places: (a) e1/10 (b) e−1/10 (c) e2.

1.4 Outline of the Book

This section gives brief summaries of the remaining chapters of the book.

Chapter 2 is not about data mining per se. Rather, it introduces us to the MapReduce methodology for exploiting parallelism in computing clouds (racks

16 CHAPTER 1. DATA MINING

of interconnected processors). There is reason to believe that cloud computing, and MapReduce in particular, will become the normal way to compute when analysis of very large amounts of data is involved. A pervasive issue in later chapters will be the exploitation of the MapReduce methodology to implement the algorithms we cover.

Chapter 3 is about finding similar items. Our starting point is that items can be represented by sets of elements, and similar sets are those that have a large fraction of their elements in common. The key techniques of minhashing and locality-sensitive hashing are explained. These techniques have numerous applications and often give surprisingly efficient solutions to problems that ap- pear impossible for massive data sets.

In Chapter 4, we consider data in the form of a stream. The difference between a stream and a database is that the data in a stream is lost if you do not do something about it immediately. Important examples of streams are the streams of search queries at a search engine or clicks at a popular Web site. In this chapter, we see several of the surprising applications of hashing that make management of stream data feasible.

Chapter 5 is devoted to a single application: the computation of PageRank. This computation is the idea that made Google stand out from other search engines, and it is still an essential part of how search engines know what pages the user is likely to want to see. Extensions of PageRank are also essential in the fight against spam (euphemistically called “search engine optimization”), and we shall examine the latest extensions of the idea for the purpose of combating spam.Then, Chapter 6 introduces the market-basket model of data, and its canon- ical problems of association rules and finding frequent itemsets. In the market- basket model, data consists of a large collection of baskets, each of which con- tains a small set of items. We give a sequence of algorithms capable of finding all frequent pairs of items, that is pairs of items that appear together in many baskets. Another sequence of algorithms are useful for finding most of the frequent itemsets larger than pairs, with high efficiency.

Chapter 7 examines the problem of clustering. We assume a set of items with a distance measure defining how close or far one item is from another. The goal is to examine a large amount of data and partition it into subsets (clusters), each cluster consisting of items that are all close to one another, yet far from items in the other clusters.

Chapter 8 is devoted to on-line advertising and the computational problems it engenders. We introduce the notion of an on-line algorithm – one where a good response must be given immediately, rather than waiting until we have seen the entire dataset. The idea of competitive ratio is another important concept covered in this chapter; it is the ratio of the guaranteed performance of an on-line algorithm compared with the performance of the optimal algorithm that is allowed to see all the data before making any decisions. These ideas are used to give good algorithms that match bids by advertisers for the right to display their ad in response to a query against the search queries arriving at a

1.5. SUMMARY OF CHAPTER 1 17

search engine.

Chapter 9 is devoted to recommendation systems. Many Web applications involve advising users on what they might like. The Netflix challenge is one example, where it is desired to predict what movies a user would like, or Ama- zon’s problem of pitching a product to a customer based on information about what they might be interested in buying. There are two basic approaches to recommendation. We can characterize items by features, e.g., the stars of a movie, and recommend items with the same features as those the user is known to like. Or, we can look at other users with preferences similar to that of the user in question, and see what they liked (a technique known as collaborative filtering).

In Chapter 10, we study social networks and algorithms for their analysis. The canonical example of a social network is the graph of Facebook friends, where the nodes are people, and edges connect two people if they are friends. Directed graphs, such as followers on Twitter, can also be viewed as social networks. A common example of a problem to be addressed is identifying “communities,” that is, small sets of nodes with an unusually large number of edges among them. Other questions about social networks are general questions about graphs, such as computing the transitive closure or diameter of a graph, but are made more difficult by the size of typical networks.

Chapter 11 looks at dimensionality reduction. We are given a very large matrix, typically sparse. Think of the matrix as representing a relationship between two kinds of entities, e.g., ratings of movies by viewers. Intuitively, there are a small number of concepts, many fewer concepts than there are movies or viewers, that explain why certain viewers like certain movies. We offer several algorithms that simplify matrices by decomposing them into a product of matrices that are much smaller in one of the two dimensions. One matrix relates entities of one kind to the small number of concepts and another relates the concepts to the other kind of entity. If done correctly, the product of the smaller matrices will be very close to the original matrix.

Finally, Chapter 12 discusses algorithms for machine learning from very large datasets. Techniques covered include perceptrons, support-vector ma- chines, finding models by gradient descent, nearest-neighbor models, and deci- sion trees.

1.5 Summary of Chapter 1

♦ Data Mining: This term refers to the process of extracting useful models of data. Sometimes, a model can be a summary of the data, or it can be the set of most extreme features of the data.

♦ Bonferroni’s Principle: If we are willing to view as an interesting fea- ture of data something of which many instances can be expected to exist in random data, then we cannot rely on such features being significant.

18 CHAPTER 1. DATA MINING

This observation limits our ability to mine data for features that are not sufficiently rare in practice.

♦ TF.IDF: The measure called TF.IDF lets us identify words in a collection of documents that are useful for determining the topic of each document. A word has high TF.IDF score in a document if it appears in relatively few documents, but appears in this one, and when it appears in a document it tends to appear many times.

♦ Hash Functions: A hash function maps hash-keys of some data type to integer bucket numbers. A good hash function distributes the possible hash-key values approximately evenly among buckets. Any data type can be the domain of a hash function.

♦ Indexes: An index is a data structure that allows us to store and retrieve data records efficiently, given the value in one or more of the fields of the record. Hashing is one way to build an index.

♦ Storage on Disk: When data must be stored on disk (secondary memory), it takes very much more time to access a desired data item than if the same data were stored in main memory. When data is large, it is important that algorithms strive to keep needed data in main memory.

♦ Power Laws: Many phenomena obey a law that can be expressed as y = cxa for some power a, often around −2. Such phenomena include the sales of the xth most popular book, or the number of in-links to the xth most popular page.

1.6 References for Chapter 1

[7] is a clear introduction to the basics of data mining. [2] covers data mining principally from the point of view of machine learning and statistics.

For construction of hash functions and hash tables, see [4]. Details of the TF.IDF measure and other matters regarding document processing can be found in [5]. See [3] for more on managing indexes, hash tables, and data on disk.

Power laws pertaining to the Web were explored by [1]. The Matthew effect was first observed in [6].

1. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Weiner, “Graph structure in the web,” Com- puter Networks 33:1–6, pp. 309–320, 2000.

2. M.M. Gaber, Scientific Data Mining and Knowledge Discovery — Prin-

ciples and Foundations, Springer, New York, 2010.

1.6. REFERENCES FOR CHAPTER 1 19

3. H. Garcia-Molina, J.D. Ullman, and J. Widom, Database Systems: The Complete Book Second Edition, Prentice-Hall, Upper Saddle River, NJ, 2009.

4. D.E. Knuth, The Art of Computer Programming Vol. 3 (Sorting and Searching), Second Edition, Addison-Wesley, Upper Saddle River, NJ, 1998.

5. C.P. Manning, P. Raghavan, and H. Schütze, Introduction to Information

Retrieval, Cambridge Univ. Press, 2008.

6. R.K. Merton, “The Matthew effect in science,” Science 159:3810, pp. 56–

63, Jan. 5, 1968.

7. P.-N. Tan, M. Steinbach, and V. Kumar, Introduction to Data Mining,

Addison-Wesley, Upper Saddle River, NJ, 2005.

20 CHAPTER 1. DATA MINING

Chapter 2

MapReduce and the New Software Stack

Modern data-mining applications, often called “big-data” analysis, require us to manage immense amounts of data quickly. In many of these applications, the data is extremely regular, and there is ample opportunity to exploit parallelism. Important examples are:

1. The ranking of Web pages by importance, which involves an iterated

matrix-vector multiplication where the dimension is many billions.

2. Searches in “friends” networks at social-networking sites, which involve graphs with hundreds of millions of nodes and many billions of edges.

To deal with applications such as these, a new software stack has evolved. These programming systems are designed to get their parallelism not from a “super- computer,” but from “computing clusters” – large collections of commodity hardware, including conventional processors (“compute nodes”) connected by Ethernet cables or inexpensive switches. The software stack begins with a new form of file system, called a “distributed file system,” which features much larger units than the disk blocks in a conventional operating system. Distributed file systems also provide replication of data or redundancy to protect against the frequent media failures that occur when data is distributed over thousands of low-cost compute nodes.

On top of these file systems, many different higher-level programming sys- tems have been developed. Central to the new software stack is a programming system called MapReduce. Implementations of MapReduce enable many of the most common calculations on large-scale data to be performed on computing clusters efficiently and in a way that is tolerant of hardware failures during the computation.

MapReduce systems are evolving and extending rapidly. Today, it is com- mon for MapReduce programs to be created from still higher-level programming

21

22 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

systems, often an implementation of SQL. Further, MapReduce turns out to be a useful, but simple, case of more general and powerful ideas. We include in this chapter a discussion of generalizations of MapReduce, first to systems that support acyclic workflows and then to systems that implement recursive algorithms.

Our last topic for this chapter is the design of good MapReduce algorithms, a subject that often differs significantly from the matter of designing good parallel algorithms to be run on a supercomputer. When designing MapReduce algorithms, we often find that the greatest cost is in the communication. We thus investigate communication cost and what it tells us about the most efficient MapReduce algorithms. For several common applications of MapReduce we are able to give families of algorithms that optimally trade the communication cost against the degree of parallelism.

2.1 Distributed File Systems

Most computing is done on a single processor, with its main memory, cache, and local disk (a compute node). In the past, applications that called for parallel processing, such as large scientific calculations, were done on special-purpose parallel computers with many processors and specialized hardware. However, the prevalence of large-scale Web services has caused more and more computing to be done on installations with thousands of compute nodes operating more or less independently. In these installations, the compute nodes are commodity hardware, which greatly reduces the cost compared with special-purpose parallel machines.

These new computing facilities have given rise to a new generation of pro- gramming systems. These systems take advantage of the power of parallelism and at the same time avoid the reliability problems that arise when the comput- ing hardware consists of thousands of independent components, any of which could fail at any time. In this section, we discuss both the characteristics of these computing installations and the specialized file systems that have been developed to take advantage of them.

2.1.1 Physical Organization of Compute Nodes

The new parallel-computing architecture, sometimes called cluster computing, is organized as follows. Compute nodes are stored on racks, perhaps 8–64 on a rack. The nodes on a single rack are connected by a network, typically gigabit Ethernet. There can be many racks of compute nodes, and racks are connected by another level of network or a switch. The bandwidth of inter-rack communication is somewhat greater than the intrarack Ethernet, but given the number of pairs of nodes that might need to communicate between racks, this bandwidth may be essential. Figure 2.1 suggests the architecture of a large- scale computing system. However, there may be many more racks and many more compute nodes per rack.

2.1. DISTRIBUTED FILE SYSTEMS 23

Switch

Racks of compute nodes

Figure 2.1: Compute nodes are organized into racks, and racks are intercon- nected by a switch

It is a fact of life that components fail, and the more components, such as compute nodes and interconnection networks, a system has, the more frequently something in the system will not be working at any given time. For systems such as Fig. 2.1, the principal failure modes are the loss of a single node (e.g., the disk at that node crashes) and the loss of an entire rack (e.g., the network connecting its nodes to each other and to the outside world fails).

Some important calculations take minutes or even hours on thousands of compute nodes. If we had to abort and restart the computation every time one component failed, then the computation might never complete successfully. The solution to this problem takes two forms:

1. Files must be stored redundantly. If we did not duplicate the file at several compute nodes, then if one node failed, all its files would be unavailable until the node is replaced. If we did not back up the files at all, and the disk crashes, the files would be lost forever. We discuss file management in Section 2.1.2.

2. Computations must be divided into tasks, such that if any one task fails to execute to completion, it can be restarted without affecting other tasks. This strategy is followed by the MapReduce programming system that we introduce in Section 2.2.

2.1.2 Large-Scale File-System Organization

To exploit cluster computing, files must look and behave somewhat differently from the conventional file systems found on single computers. This new file system, often called a distributed file system or DFS (although this term has had other meanings in the past), is typically used as follows.

24 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

• Files can be enormous, possibly a terabyte in size. If you have only small files, there is no point using a DFS for them.

• Files are rarely updated. Rather, they are read as data for some calcula- tion, and possibly additional data is appended to files from time to time. For example, an airline reservation system would not be suitable for a DFS, even if the data were very large, because the data is changed so frequently.

Files are divided into chunks, which are typically 64 megabytes in size. Chunks are replicated, perhaps three times, at three different compute nodes. Moreover, the nodes holding copies of one chunk should be located on different racks, so we don’t lose all copies due to a rack failure. Normally, both the chunk size and the degree of replication can be decided by the user.

To find the chunks of a file, there is another small file called the master node or name node for that file. The master node is itself replicated, and a directory for the file system as a whole knows where to find its copies. The directory itself can be replicated, and all participants using the DFS know where the directory copies are.

2.2 MapReduce

MapReduce is a style of computing that has been implemented in several sys- tems, including Google’s internal implementation (simply called MapReduce) and the popular open-source implementation Hadoop which can be obtained, along with the HDFS file system from the Apache Foundation. You can use an implementation of MapReduce to manage many large-scale computations in a way that is tolerant of hardware faults. All you need to write are two functions, called Map and Reduce, while the system manages the parallel exe- cution, coordination of tasks that execute Map or Reduce, and also deals with

DFS Implementations

There are several distributed file systems of the type we have described that are used in practice. Among these:

1. The Google File System (GFS), the original of the class.

2. Hadoop Distributed File System (HDFS), an open-source DFS used with Hadoop, an implementation of MapReduce (see Section 2.2) and distributed by the Apache Software Foundation.

3. CloudStore, an open-source DFS originally developed by Kosmix.

2.2. MAPREDUCE 25

the possibility that one of these tasks will fail to execute. In brief, a MapReduce computation executes as follows:

1. Some number of Map tasks each are given one or more chunks from a distributed file system. These Map tasks turn the chunk into a sequence of key-value pairs. The way key-value pairs are produced from the input data is determined by the code written by the user for the Map function.

2. The key-value pairs from each Map task are collected by a master con- troller and sorted by key. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Re- duce task.

3. The Reduce tasks work on one key at a time, and combine all the val- ues associated with that key in some way. The manner of combination of values is determined by the code written by the user for the Reduce function.

Figure 2.2 suggests this computation.

Keys with all

Key−value

pairs

their values (k, [v, w,...]) Input

(k,v) chunks

Combined

output

Map

Group tasks

by keys

Reduce

tasks

Figure 2.2: Schematic of a MapReduce computation

2.2.1 The Map Tasks

We view input files for a Map task as consisting of elements, which can be any type: a tuple or a document, for example. A chunk is a collection of elements, and no element is stored across two chunks. Technically, all inputs

26 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

to Map tasks and outputs from Reduce tasks are of the key-value-pair form, but normally the keys of input elements are not relevant and we shall tend to ignore them. Insisting on this form for inputs and outputs is motivated by the desire to allow composition of several MapReduce processes.

The Map function takes an input element as its argument and produces zero or more key-value pairs. The types of keys and values are each arbitrary. Further, keys are not “keys” in the usual sense; they do not have to be unique. Rather a Map task can produce several key-value pairs with the same key, even from the same element.

Example 2.1: We shall illustrate a MapReduce computation with what has become the standard example application: counting the number of occurrences for each word in a collection of documents. In this example, the input file is a repository of documents, and each document is an element. The Map function for this example uses keys that are of type String (the words) and values that are integers. The Map task reads a document and breaks it into its sequence of words w1,w2,...,wn. It then emits a sequence of key-value pairs where the value is always 1. That is, the output of the Map task for this document is the sequence of key-value pairs:(w1,1), (w2,1),...,(wn,1)

Note that a single Map task will typically process many documents – all the documents in one or more chunks. Thus, its output will be more than the sequence for the one document suggested above. Note also that if a word w appears m times among all the documents assigned to that process, then there will be m key-value pairs (w,1) among its output. An option, which we discuss in Section 2.2.4, is to combine these m pairs into a single pair (w, m), but we can only do that because, as we shall see, the Reduce tasks apply an associative and commutative operation, addition, to the values. ✷

2.2.2 Grouping by Key

As soon as the Map tasks have all completed successfully, the key-value pairs are grouped by key, and the values associated with each key are formed into a list of values. The grouping is performed by the system, regardless of what the Map and Reduce tasks do. The master controller process knows how many Reduce tasks there will be, say r such tasks. The user typically tells the MapReduce system what r should be. Then the master controller picks a hash function that applies to keys and produces a bucket number from 0 to r − 1. Each key that is output by a Map task is hashed and its key-value pair is put in one of r local files. Each file is destined for one of the Reduce tasks.1

1Optionally, users can specify their own hash function or other method for assigning keys to Reduce tasks. However, whatever algorithm is used, each key is assigned to one and only one Reduce task.

2.2. MAPREDUCE 27

To perform the grouping by key and distribution to the Reduce tasks, the master controller merges the files from each Map task that are destined for a particular Reduce task and feeds the merged file to that process as a se- quence of key-list-of-value pairs. That is, for each key k, the input to the Reduce task that handles key k is a pair of the form (k,[v1,v2,...,vn]), where (k, v1), (k, v2),...,(k, vn) are all the key-value pairs with key k coming from all the Map tasks.

2.2.3 The Reduce Tasks

The Reduce function’s argument is a pair consisting of a key and its list of associated values. The output of the Reduce function is a sequence of zero or more key-value pairs. These key-value pairs can be of a type different from those sent from Map tasks to Reduce tasks, but often they are the same type. We shall refer to the application of the Reduce function to a single key and its associated list of values as a reducer.

A Reduce task receives one or more keys and their associated value lists. That is, a Reduce task executes one or more reducers. The outputs from all the Reduce tasks are merged into a single file. Reducers may be partitioned among a smaller number of Reduce tasks is by hashing the keys and associating each Reduce task with one of the buckets of the hash function.

Example 2.2: Let us continue with the word-count example of Example 2.1. The Reduce function simply adds up all the values. The output of a reducer consists of the word and the sum. Thus, the output of all the Reduce tasks is a sequence of (w, m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents. ✷

2.2.4 Combiners

Sometimes, a Reduce function is associative and commutative. That is, the values to be combined can be combined in any order, with the same result. The addition performed in Example 2.2 is an example of an associative and commutative operation. It doesn’t matter how we group a list of numbers v1,v2,...,vn; the sum will be the same.

When the Reduce function is associative and commutative, we can push some of what the reducers do to the Map tasks. For example, instead of the Map tasks in Example 2.1 producing many pairs (w,1), (w,1),..., we could apply the Reduce function within the Map task, before the output of the Map tasks is subject to grouping and aggregation. These key-value pairs would thus be replaced by one pair with key w and value equal to the sum of all the 1’s in all those pairs. That is, the pairs with key w generated by a single Map task would be replaced by a pair (w, m), where m is the number of times that w appears among the documents handled by this Map task. Note that it is still necessary to do grouping and aggregation and to pass the result to the Reduce

28 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

tasks, since there will typically be one key-value pair with key w coming from each of the Map tasks.

2.2.5 Details of MapReduce Execution

Let us now consider in more detail how a program using MapReduce is executed. Figure 2.3 offers an outline of how processes, tasks, and files interact. Taking advantage of a library provided by a MapReduce system such as Hadoop, the user program forks a Master controller process and some number of Worker processes at different compute nodes. Normally, a Worker handles either Map tasks (a Map worker) or Reduce tasks (a Reduce worker), but not both.

The Master has many responsibilities. One is to create some number of Map tasks and some number of Reduce tasks, these numbers being selected by the user program. These tasks will be assigned to Worker processes by the Master. It is reasonable to create one Map task for every chunk of the input file(s), but we may wish to create fewer Reduce tasks. The reason for limiting the number of Reduce tasks is that it is necessary for each Map task to create an intermediate file for each Reduce task, and if there are too many Reduce tasks the number of intermediate files explodes.

The Master keeps track of the status of each Map and Reduce task (idle,

Reducers, Reduce Tasks, Compute Nodes, and Skew

If we want maximum parallelism, then we could use one Reduce task to execute each reducer, i.e., a single key and its associated value list. Further, we could execute each Reduce task at a different compute node, so they would all execute in parallel. This plan is not usually the best. One problem is that there is overhead associated with each task we create, so we might want to keep the number of Reduce tasks lower than the number of different keys. Moreover, often there are far more keys than there are compute nodes available, so we would get no benefit from a huge number of Reduce tasks.

Second, there is often significant variation in the lengths of the value lists for different keys, so different reducers take different amounts of time. If we make each reducer a separate Reduce task, then the tasks themselves will exhibit skew – a significant difference in the amount of time each takes. We can reduce the impact of skew by using fewer Reduce tasks than there are reducers. If keys are sent randomly to Reduce tasks, we can expect that there will be some averaging of the total time required by the different Reduce tasks. We can further reduce the skew by using more Reduce tasks than there are compute nodes. In that way, long Reduce tasks might occupy a compute node fully, while several shorter Reduce tasks might run sequentially at a single compute node.

2.2. MAPREDUCE 29

User Program Master

Worker

Worker

fork fork fork

assign Map

assign

Reduce

Worker

Worker

Input

Worker Data

Output

Intermediate

File

Files

Figure 2.3: Overview of the execution of a MapReduce program

executing at a particular Worker, or completed). A Worker process reports to the Master when it finishes a task, and a new task is scheduled by the Master for that Worker process.

Each Map task is assigned one or more chunks of the input file(s) and executes on it the code written by the user. The Map task creates a file for each Reduce task on the local disk of the Worker that executes the Map task. The Master is informed of the location and sizes of each of these files, and the Reduce task for which each is destined. When a Reduce task is assigned by the Master to a Worker process, that task is given all the files that form its input. The Reduce task executes code written by the user and writes its output to a file that is part of the surrounding distributed file system.

2.2.6 Coping With Node Failures

The worst thing that can happen is that the compute node at which the Master is executing fails. In this case, the entire MapReduce job must be restarted. But only this one node can bring the entire process down; other failures will be managed by the Master, and the MapReduce job will complete eventually.

Suppose the compute node at which a Map worker resides fails. This fail- ure will be detected by the Master, because it periodically pings the Worker processes. All the Map tasks that were assigned to this Worker will have to be redone, even if they had completed. The reason for redoing completed Map

30 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

tasks is that their output destined for the Reduce tasks resides at that compute node, and is now unavailable to the Reduce tasks. The Master sets the status of each of these Map tasks to idle and will schedule them on a Worker when one becomes available. The Master must also inform each Reduce task that the location of its input from that Map task has changed.

Dealing with a failure at the node of a Reduce worker is simpler. The Master simply sets the status of its currently executing Reduce tasks to idle. These will be rescheduled on another reduce worker later.

2.2.7 Exercises for Section 2.2

Exercise 2.2.1: Suppose we execute the word-count MapReduce program de- scribed in this section on a large repository such as a copy of the Web. We shall use 100 Map tasks and some number of Reduce tasks.

(a) Suppose we do not use a combiner at the Map tasks. Do you expect there to be significant skew in the times taken by the various reducers to process their value list? Why or why not?

(b) If we combine the reducers into a small number of Reduce tasks, say 10 tasks, at random, do you expect the skew to be significant? What if we instead combine the reducers into 10,000 Reduce tasks?

! (c) Suppose we do use a combiner at the 100 Map tasks. Do you expect skew

to be significant? Why or why not?

2.3 Algorithms Using MapReduce

MapReduce is not a solution to every problem, not even every problem that profitably can use many compute nodes operating in parallel. As we mentioned in Section 2.1.2, the entire distributed-file-system milieu makes sense only when files are very large and are rarely updated in place. Thus, we would not expect to use either a DFS or an implementation of MapReduce for managing on- line retail sales, even though a large on-line retailer such as Amazon.com uses thousands of compute nodes when processing requests over the Web. The reason is that the principal operations on Amazon data involve responding to searches for products, recording sales, and so on, processes that involve relatively little calculation and that change the database.2 On the other hand, Amazon might use MapReduce to perform certain analytic queries on large amounts of data, such as finding for each user those users whose buying patterns were most similar.

The original purpose for which the Google implementation of MapReduce was created was to execute very large matrix-vector multiplications as are

2Remember that even looking at a product you don’t buy causes Amazon to remember that you looked at it.

2.3. ALGORITHMS USING MAPREDUCE 31

needed in the calculation of PageRank (See Chapter 5). We shall see that matrix-vector and matrix-matrix calculations fit nicely into the MapReduce style of computing. Another important class of operations that can use MapRe- duce effectively are the relational-algebra operations. We shall examine the MapReduce execution of these operations as well.

2.3.1 Matrix-Vector Multiplication by MapReduce

Suppose we have an n×n matrix M, whose element in row i and column j will be denoted mij. Suppose we also have a vector v of length n, whose jth element is vj. Then the matrix-vector product is the vector x of length n, whose ith element xi is given by

xi =

∑nj=1

mijvj

If n = 100, we do not want to use a DFS or MapReduce for this calculation. But this sort of calculation is at the heart of the ranking of Web pages that goes on at search engines, and there, n is in the tens of billions.3 Let us first assume that n is large, but not so large that vector v cannot fit in main memory and thus be available to every Map task.

The matrix M and the vector v each will be stored in a file of the DFS. We assume that the row-column coordinates of each matrix element will be discov- erable, either from its position in the file, or because it is stored with explicit coordinates, as a triple (i, j, mij). We also assume the position of element vj in the vector v will be discoverable in the analogous way. The Map Function: The Map function is written to apply to one element of M. However, if v is not already read into main memory at the compute node executing a Map task, then v is first read, in its entirety, and subsequently will be available to all applications of the Map function performed at this Map task. Each Map task will operate on a chunk of the matrix M. From each matrix element mij it produces the key-value pair (i, mijvj). Thus, all terms of the sum that make up the component xi of the matrix-vector product will get the same key, i. The Reduce Function: The Reduce function simply sums all the values as- sociated with a given key i. The result will be a pair (i, xi).

2.3.2 If the Vector v Cannot Fit in Main Memory

However, it is possible that the vector v is so large that it will not fit in its entirety in main memory. It is not required that v fit in main memory at a compute node, but if it does not then there will be a very large number of

3The matrix is sparse, with on the average of 10 to 15 nonzero elements per row, since the matrix represents the links in the Web, with mij nonzero if and only if there is a link from page j to page i. Note that there is no way we could store a dense matrix whose side was 1010, since it would have 1020 elements.

32 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

disk accesses as we move pieces of the vector into main memory to multiply components by elements of the matrix. Thus, as an alternative, we can divide the matrix into vertical stripes of equal width and divide the vector into an equal number of horizontal stripes, of the same height. Our goal is to use enough stripes so that the portion of the vector in one stripe can fit conveniently into main memory at a compute node. Figure 2.4 suggests what the partition looks like if the matrix and vector are each divided into five stripes.

*M* Matrix Vector **v**

Figure 2.4: Division of a matrix and vector into five stripes

The ith stripe of the matrix multiplies only components from the ith stripe of the vector. Thus, we can divide the matrix into one file for each stripe, and do the same for the vector. Each Map task is assigned a chunk from one of the stripes of the matrix and gets the entire corresponding stripe of the vector. The Map and Reduce tasks can then act exactly as was described above for the case where Map tasks get the entire vector.

We shall take up matrix-vector multiplication using MapReduce again in Section 5.2. There, because of the particular application (PageRank calcula- tion), we have an additional constraint that the result vector should be part- itioned in the same way as the input vector, so the output may become the input for another iteration of the matrix-vector multiplication. We shall see there that the best strategy involves partitioning the matrix M into square blocks, rather than stripes.

2.3.3 Relational-Algebra Operations

There are a number of operations on large-scale data that are used in database queries. Many traditional database applications involve retrieval of small am- ounts of data, even though the database itself may be large. For example, a query may ask for the bank balance of one particular account. Such queries are not useful applications of MapReduce.

However, there are many operations on data that can be described easily in terms of the common database-query primitives, even if the queries themselves

2.3. ALGORITHMS USING MAPREDUCE 33

are not executed within a database management system. Thus, a good starting point for exploring applications of MapReduce is by considering the standard operations on relations. We assume you are familiar with database systems, the query language SQL, and the relational model, but to review, a relation is a table with column headers called attributes. Rows of the relation are called tuples. The set of attributes of a relation is called its schema. We often write an expression like R(A1,A2,...,An) to say that the relation name is R and its attributes are A1, A2,...,An.

From

···

Figure 2.5: Relation Links consists of the set of pairs of URL’s, such that the first has one or more links to the second

Example 2.3: In Fig. 2.5 we see part of the relation Links that describes the structure of the Web. There are two attributes, From and To. A row, or tuple, of the relation is a pair of URL’s, such that there is at least one link from the first URL to the second. For instance, the first row of Fig. 2.5 is the pair (url1, url2) that says the Web page url1 has a link to page url2. While we have shown only four tuples, the real relation of the Web, or the portion of it that would be stored by a typical search engine, has billions of tuples. ✷

A relation, however large, can be stored as a file in a distributed file system. The elements of this file are the tuples of the relation.

There are several standard operations on relations, often referred to as re- lational algebra, that are used to implement queries. The queries themselves usually are written in SQL. The relational-algebra operations we shall discuss are:1. Selection: Apply a condition C to each tuple in the relation and produce as output only those tuples that satisfy C. The result of this selection is denoted σC(R).

2. Projection: For some subset S of the attributes of the relation, produce from each tuple only the components for the attributes in S. The result of this projection is denoted πS(R).

3. Union, Intersection, and Difference: These well-known set operations apply to the sets of tuples in two relations that have the same schema. There are also bag (multiset) versions of the operations in SQL, with

url2 url1 url1

url3 url2

url3 url2

url4

···

To

34 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

somewhat unintuitive definitions, but we shall not go into the bag versions of these operations here.

4. Natural Join: Given two relations, compare each pair of tuples, one from each relation. If the tuples agree on all the attributes that are common to the two schemas, then produce a tuple that has components for each of the attributes in either schema and agrees with the two tuples on each attribute. If the tuples disagree on one or more shared attributes, then produce nothing from this pair of tuples. The natural join of relations R and S is denoted R ⊲⊳ S. While we shall discuss executing only the nat- ural join with MapReduce, all equijoins (joins where the tuple-agreement condition involves equality of attributes from the two relations that do not necessarily have the same name) can be executed in the same manner. We shall give an illustration in Example 2.4.

5. Grouping and Aggregation:4 Given a relation R, partition its tuples according to their values in one set of attributes G, called the grouping attributes. Then, for each group, aggregate the values in certain other at- tributes. The normally permitted aggregations are SUM, COUNT, AVG, MIN, and MAX, with the obvious meanings. Note that MIN and MAX require that the aggregated attributes have a type that can be compared, e.g., numbers or strings, while SUM and AVG require that the type allow arithmetic operations. We denote a grouping-and-aggregation operation on a relation R by γX(R), where X is a list of elements that are either

(a) A grouping attribute, or (b) An expression θ(A), where θ is one of the five aggregation opera- tions such as SUM, and A is an attribute not among the grouping attributes.

The result of this operation is one tuple for each group. That tuple has a component for each of the grouping attributes, with the value common to tuples of that group. It also has a component for each aggregation, with the aggregated value for that group. We shall see an illustration in Example 2.5.

Example 2.4: Let us try to find the paths of length two in the Web, using the relation Links of Fig. 2.5. That is, we want to find the triples of URL’s (u, v, w) such that there is a link from u to v and a link from v to w. We essentially want to take the natural join of Links with itself, but we first need to imagine that it is two relations, with different schemas, so we can describe the desired connection as a natural join. Thus, imagine that there are two copies of Links, namely L1(U1,U2) and L2(U2,U3). Now, if we compute L1 ⊲⊳ L2,

4Some descriptions of relational algebra do not include these operations, and indeed they were not part of the original definition of this algebra. However, these operations are so important in SQL, that modern treatments of relational algebra include them.

2.3. ALGORITHMS USING MAPREDUCE 35

we shall have exactly what we want. That is, for each tuple t1 of L1 (i.e., each tuple of Links) and each tuple t2 of L2 (another tuple of Links, possibly even the same tuple), see if their U2 components are the same. Note that these components are the second component of t1 and the first component of t2. If these two components agree, then produce a tuple for the result, with schema (U1,U2,U3). This tuple consists of the first component of t1, the second component of t1 (which must equal the first component of t2), and the second component of t2.

We may not want the entire path of length two, but only want the pairs (u, w) of URL’s such that there is at least one path from u to w of length two. If so, we can project out the middle components by computing πU1,U3(L1 ⊲⊳ L2). ✷Example 2.5: Imagine that a social-networking site has a relation

Friends(User, Friend)

This relation has tuples that are pairs (a, b) such that b is a friend of a. The site might want to develop statistics about the number of friends members have. Their first step would be to compute a count of the number of friends of each user. This operation can be done by grouping and aggregation, specifically

γUser,COUNT(Friend)(Friends)

This operation groups all the tuples by the value in their first component, so there is one group for each user. Then, for each group the count of the number of friends of that user is made. The result will be one tuple for each group, and a typical tuple would look like (Sally,300), if user “Sally” has 300 friends. ✷

2.3.4 Computing Selections by MapReduce

Selections really do not need the full power of MapReduce. They can be done most conveniently in the map portion alone, although they could also be done in the reduce portion alone. Here is a MapReduce implementation of selection σC(R). The Map Function: For each tuple t in R, test if it satisfies C. If so, produce the key-value pair (t, t). That is, both the key and value are t.

The Reduce Function: The Reduce function is the identity. It simply passes each key-value pair to the output.

Note that the output is not exactly a relation, because it has key-value pairs. However, a relation can be obtained by using only the value components (or only the key components) of the output.

36 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

2.3.5 Computing Projections by MapReduce

Projection is performed similarly to selection, because projection may cause the same tuple to appear several times, the Reduce function must eliminate duplicates. We may compute πS(R) as follows. The Map Function: For each tuple t in R, construct a tuple t′ by eliminating from t those components whose attributes are not in S. Output the key-value pair (t′,t′). The Reduce Function: For each key t′ produced by any of the Map tasks, there will be one or more key-value pairs (t′,t′). The Reduce function turns (t′,[t′,t′,...,t′]) into (t′,t′), so it produces exactly one pair (t′,t′) for this key t′.

Observe that the Reduce operation is duplicate elimination. This operation is associative and commutative, so a combiner associated with each Map task can eliminate whatever duplicates are produced locally. However, the Reduce tasks are still needed to eliminate two identical tuples coming from different Map tasks.

2.3.6 Union, Intersection, and Difference by MapReduce

First, consider the union of two relations. Suppose relations R and S have the same schema. Map tasks will be assigned chunks from either R or S; it doesn’t matter which. The Map tasks don’t really do anything except pass their input tuples as key-value pairs to the Reduce tasks. The latter need only eliminate duplicates as for projection. The Map Function: Turn each input tuple t into a key-value pair (t, t). The Reduce Function: Associated with each key t there will be either one or two values. Produce output (t, t) in either case.

To compute the intersection, we can use the same Map function. However, the Reduce function must produce a tuple only if both relations have the tuple. If the key t has a list of two values [t, t] associated with it, then the Reduce task for t should produce (t, t). However, if the value-list associated with key t is just [t], then one of R and S is missing t, so we don’t want to produce a tuple for the intersection. The Map Function: Turn each tuple t into a key-value pair (t, t). The Reduce Function: If key t has value list [t, t], then produce (t, t). Oth- erwise, produce nothing.

The Difference R − S requires a bit more thought. The only way a tuple t can appear in the output is if it is in R but not in S. The Map function can pass tuples from R and S through, but must inform the Reduce function whether the tuple came from R or S. We shall thus use the relation as the value associated with the key t. Here is a specification for the two functions.

2.3. ALGORITHMS USING MAPREDUCE 37

The Map Function: For a tuple t in R, produce key-value pair (t, R), and for a tuple t in S, produce key-value pair (t, S). Note that the intent is that the value is the name of R or S (or better, a single bit indicating whether the relation is R or S), not the entire relation. The Reduce Function: For each key t, if the associated value list is [R], then produce (t, t). Otherwise, produce nothing.

2.3.7 Computing Natural Join by MapReduce

The idea behind implementing natural join via MapReduce can be seen if we look at the specific case of joining R(A, B) with S(B,C). We must find tuples that agree on their B components, that is the second component from tuples of R and the first component of tuples of S. We shall use the B-value of tuples from either relation as the key. The value will be the other component and the name of the relation, so the Reduce function can know where each tuple came from. The Map Function: For each tuple (a, b) of R, produce the key-value pair (b,(R, a)). For each tuple (b, c) of S, produce the key-value pair (b,(S, c)). The Reduce Function: Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c). Construct all pairs consisting of one with first component R and the other with first component S, say (R, a) and (S, c). The output from this key and value list is a sequence of key-value pairs. The key is irrelevant. Each value is one of the triples (a, b, c) such that (R, a) and (S, c) are on the input list of values.

The same algorithm works if the relations have more than two attributes. You can think of A as representing all those attributes in the schema of R but not S. B represents the attributes in both schemas, and C represents attributes only in the schema of S. The key for a tuple of R or S is the list of values in all the attributes that are in the schemas of both R and S. The value for a tuple of R is the name R together with the values of all the attributes belonging to R but not to S, and the value for a tuple of S is the name S together with the values of the attributes belonging to S but not R.

The Reduce function looks at all the key-value pairs with a given key and combines those values from R with those values of S in all possible ways. From each pairing, the tuple produced has the values from R, the key values, and the values from S.

2.3.8 Grouping and Aggregation by MapReduce

As with the join, we shall discuss the minimal example of grouping and aggrega- tion, where there is one grouping attribute and one aggregation. Let R(A,B,C) be a relation to which we apply the operator γA,θ(B)(R). Map will perform the grouping, while Reduce does the aggregation. The Map Function: For each tuple (a, b, c) produce the key-value pair (a, b).

38 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

The Reduce Function: Each key a represents a group. Apply the aggregation operator θ to the list [b1,b2,...,bn] of B-values associated with key a. The output is the pair (a, x), where x is the result of applying θ to the list. For example, if θ is SUM, then x = b1 + b2 + ··· + bn, and if θ is MAX, then x is the largest of b1, b2,...,bn.

If there are several grouping attributes, then the key is the list of the values of a tuple for all these attributes. If there is more than one aggregation, then the Reduce function applies each of them to the list of values associated with a given key and produces a tuple consisting of the key, including components for all grouping attributes if there is more than one, followed by the results of each of the aggregations.

2.3.9 Matrix Multiplication

If M is a matrix with element mij in row i and column j, and N is a matrix with element njk in row j and column k, then the product P = MN is the matrix P with element pik in row i and column k, where

pik = ∑j

mijnjk

It is required that the number of columns of M equals the number of rows of N, so the sum over j makes sense.

We can think of a matrix as a relation with three attributes: the row number, the column number, and the value in that row and column. Thus, we could view matrix M as a relation M(I,J,V ), with tuples (i, j, mij), and we could view matrix N as a relation N(J, K, W), with tuples (j, k, njk). As large matrices are often sparse (mostly 0’s), and since we can omit the tuples for matrix elements that are 0, this relational representation is often a very good one for a large matrix. However, it is possible that i, j, and k are implicit in the position of a matrix element in the file that represents it, rather than written explicitly with the element itself. In that case, the Map function will have to be designed to construct the I, J, and K components of tuples from the position of the data. The product MN is almost a natural join followed by grouping and ag- gregation. That is, the natural join of M(I,J,V ) and N(J, K, W), having only attribute J in common, would produce tuples (i, j, k, v, w) from each tuple (i, j, v) in M and tuple (j, k, w) in N. This five-component tuple represents the pair of matrix elements (mij,njk). What we want instead is the product of these elements, that is, the four-component tuple (i, j, k, v × w), because that represents the product mijnjk. Once we have this relation as the result of one MapReduce operation, we can perform grouping and aggregation, with I and K as the grouping attributes and the sum of V × W as the aggregation. That is, we can implement matrix multiplication as the cascade of two MapReduce operations, as follows. First: The (j,(M,i,mMap ij)Function: ). For each matrix element mij, produce the key value pair Likewise, for each matrix element njk, produce the key value

2.3. ALGORITHMS USING MAPREDUCE 39

pair (j,(N,k,njk)). Note that M and N in the values are not the matrices themselves. Rather they are names of the matrices or (as we mentioned for the similar Map function used for natural join) better, a bit indicating whether the element comes from M or N. The Reduce Function: For each key j, examine its list of associated values. For each value that comes from M, say (M,i,mij), and each value that comes from N, say (N,k,njk), produce a key-value pair with key equal to (i, k) and value equal to the product of these elements, mijnjk.

Now, we perform a grouping and aggregation by another MapReduce operation. The Map Function: This function is just the identity. That is, for every input element with key (i, k) and value v, produce exactly this key-value pair. The Reduce Function: For each key (i, k), produce the sum of the list of values associated with this key. The result is a pair ((i, k),v), where v is the value of the element in row i and column k of the matrix P = MN.

2.3.10 Matrix Multiplication with One MapReduce Step

There often is more than one way to use MapReduce to solve a problem. You may wish to use only a single MapReduce pass to perform matrix multiplication P = MN. 5 It is possible to do so if we put more work into the two functions. Start by using the Map function to create the sets of matrix elements that are needed to compute each element of the answer P. Notice that an element of M or N contributes to many elements of the result, so one input element will be turned into many key-value pairs. The keys will be pairs (i, k), where i is a row of M and k is a column of N. Here is a synopsis of the Map and Reduce functions. The Map Function: For each element mij of M, produce all the key-value pairs ((i, k), (M,j,mij)) for k = 1,2,..., up to the number of columns of N. Similarly, for each element njk of N, produce all the key-value pairs ((i, k), (N,j,njk)) for i = 1,2,..., up to the number of rows of M. As be- fore, M and N are really bits to tell which of the two matrices a value comes from. The Reduce Function: Each key (i, k) will have an associated list with all the values (M,j,mij) and (N,j,njk), for all possible values of j. The Reduce function needs to connect the two values on the list that have the same value of j, for each j. An easy way to do this step is to sort by j the values that begin with M and sort by j the values that begin with N, in separate lists. The jth values on each list must have their third components, mij and njk extracted and multiplied. Then, these products are summed and the result is paired with (i, k) in the output of the Reduce function.

5However, we show in Section 2.6.7 that two passes of MapReduce are usually better than one for matrix multiplication.

40 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

You may notice that if a row of the matrix M or a column of the matrix N is so large that it will not fit in main memory, then the Reduce tasks will be forced to use an external sort to order the values associated with a given key (i, k). However, in that case, the matrices themselves are so large, perhaps 1020 elements, that it is unlikely we would attempt this calculation if the matrices were dense. If they are sparse, then we would expect many fewer values to be associated with any one key, and it would be feasible to do the sum of products in main memory.

2.3.11 Exercises for Section 2.3

Exercise 2.3.1: Design MapReduce algorithms to take a very large file of integers and produce as output:

(a) The largest integer.

(b) The average of all the integers.

(c) The same set of integers, but with each integer appearing only once.

(d) The count of the number of distinct integers in the input.

Exercise 2.3.2: Our formulation of matrix-vector multiplication assumed that the matrix M was square. Generalize the algorithm to the case where M is an r-by-c matrix for some number of rows r and columns c.

! Exercise 2.3.3: In the form of relational algebra implemented in SQL, rela- tions are not sets, but bags; that is, tuples are allowed to appear more than once. There are extended definitions of union, intersection, and difference for bags, which we shall define below. Write MapReduce algorithms for computing the following operations on bags R and S:

(a) Bag Union, defined to be the bag of tuples in which tuple t appears the

sum of the numbers of times it appears in R and S.

(b) Bag Intersection, defined to be the bag of tuples in which tuple t appears

the minimum of the numbers of times it appears in R and S.

(c) Bag Difference, defined to be the bag of tuples in which the number of times a tuple t appears is equal to the number of times it appears in R minus the number of times it appears in S. A tuple that appears more times in S than in R does not appear in the difference.

! Exercise 2.3.4: Selection can also be performed on bags. Give a MapReduce implementation that produces the proper number of copies of each tuple t that passes the selection condition. That is, produce key-value pairs from which the correct result of the selection can be obtained easily from the values.

2.4. EXTENSIONS TO MAPREDUCE 41

Exercise 2.3.5: The relational-algebra operation R(A, B) ⊲⊳ B<C S(C, D) produces all tuples (a, b, c, d) such that tuple (a, b) is in relation R, tuple (c, d) is in S, and b<c. Give a MapReduce implementation of this operation, assuming R and S are sets.

2.4 Extensions to MapReduce

MapReduce has proved so influential that it has spawned a number of extensions and modifications. These systems typically share a number of characteristics with MapReduce systems:

1. They are built on a distributed file system.

2. They manage very large numbers of tasks that are instantiations of a

small number of user-written functions.

3. They incorporate a method for dealing with most of the failures that occur during the execution of a large job, without having to restart that job from the beginning.

In this section, we shall mention some of the interesting directions being ex- plored. References to the details of the systems mentioned can be found in the bibliographic notes for this chapter.

2.4.1 Workflow Systems

Two experimental systems called Clustera from the University of Wisconsin and Hyracks from the University of California at Irvine extend MapReduce from the simple two-step workflow (the Map function feeds the Reduce function) to any collection of functions, with an acyclic graph representing workflow among the functions. That is, there is an acyclic flow graph whose arcs a → b represent the fact that function a’s output is input to function b. A suggestion of what a workflow might look like is in Fig. 2.6. There, five functions, f through j, pass data from left to right in specific ways, so the flow of data is acyclic and no task needs to provide data out before its input is available. For instance, function h takes its input from a preexisting file of the distributed file system. Each of h’s output elements is passed to at least one of the functions i and j.

In analogy to Map and Reduce functions, each function of a workflow can be executed by many tasks, each of which is assigned a portion of the input to the function. A master controller is responsible for dividing the work among the tasks that implement a function, usually by hashing the input elements to decide on the proper task to receive an element. Thus, like Map tasks, each task implementing a function f has an output file of data destined for each of the tasks that implement the successor function(s) of f. These files are delivered by the Master at the appropriate time – after the task has completed its work.

42 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

f ghi

j

Figure 2.6: An example of a workflow that is more complex than Map feeding Reduce

The functions of a workflow, and therefore the tasks, share with MapReduce tasks the important property that they only deliver output after they complete. As a result, if a task fails, it has not delivered output to any of its successors in the flow graph. A master controller can therefore restart the failed task at another compute node, without worrying that the output of the restarted task will duplicate output that previously was passed to some other task.

Many applications of workflow systems such as Clustera or Hyracks are cascades of MapReduce jobs. An example would be the join of three relations, where one MapReduce job joins the first two relations, and a second MapReduce job joins the third relation with the result of joining the first two relations. Both jobs would use an algorithm like that of Section 2.3.7.

There is an advantage to implementing such cascades as a single workflow. For example, the flow of data among tasks, and its replication, can be managed by the master controller, without need to store the temporary file that is out- put of one MapReduce job in the distributed file system. By locating tasks at compute nodes that have a copy of their input, we can avoid much of the com- munication that would be necessary if we stored the result of one MapReduce job and then initiated a second MapReduce job (although Hadoop and other MapReduce systems also try to locate Map tasks where a copy of their input is already present).

2.4.2 Recursive Extensions to MapReduce

Many large-scale computations are really recursions. An important example is PageRank, which is the subject of Chapter 5. That computation is, in sim- ple terms, the computation of the fixedpoint of a matrix-vector multiplication. It is computed under MapReduce systems by the iterated application of the matrix-vector multiplication algorithm described in Section 2.3.1, or by a more complex strategy that we shall introduce in Section 5.2. The iteration typi- cally continues for an unknown number of steps, each step being a MapReduce job, until the results of two consecutive iterations are sufficiently close that we believe convergence has occurred.

The reason recursions are normally implemented by iterated MapReduce

2.4. EXTENSIONS TO MAPREDUCE 43

jobs is that a true recursive task does not have the property necessary for independent restart of failed tasks. It is impossible for a collection of mutually recursive tasks, each of which has an output that is input to at least some of the other tasks, to produce output only at the end of the task. If they all followed that policy, no task would ever receive any input, and nothing could be accomplished. As a result, some mechanism other than simple restart of failed tasks must be implemented in a system that handles recursive workflows (flow graphs that are not acyclic). We shall start by studying an example of a recursion implemented as a workflow, and then discuss approaches to dealing with task failures.

Example 2.6: Suppose we have a directed graph whose arcs are represented by the relation E(X, Y ), meaning that there is an arc from node X to node Y . We wish to compute the paths relation P(X, Y ), meaning that there is a path of length 1 or more from node X to node Y . That is, P is the transitive closure of E. A simple recursive algorithm to do so is:

1. Start with P(X, Y ) = E(X, Y ).

2. While changes to the relation P occur, add to P all tuples in

πX,Y (P(X, Z) ⊲⊳ P(Z, Y ))

That is, find pairs of nodes X and Y such that for some node Z there is known to be a path from X to Z and also a path from Z to Y .

Figure 2.7 suggests how we could organize recursive tasks to perform this computation. There are two kinds of tasks: Join tasks and Dup-elim tasks. There are n Join tasks, for some n, and each corresponds to a bucket of a hash function h. A path tuple P(a, b), when it is discovered, becomes input to two Join tasks: those numbered h(a) and h(b). The job of the ith Join task, when it receives input tuple P(a, b), is to find certain other tuples seen previously (and stored locally by that task).

1. Store P(a, b) locally.

2. If h(a) = i then look for tuples P(x, a) and produce output tuple P(x, b).

3. If h(b) = i then look for tuples P(b, y) and produce output tuple P(a, y).

Note that in rare cases, we have h(a) = h(b), so both (2) and (3) are executed. But generally, only one of these needs to be executed for a given tuple.

There are also m Dup-elim tasks, and each corresponds to a bucket of a hash function g that takes two arguments. If P(c, d) is an output of some Join task, then it is sent to Dup-elim task j = g(c, d). On receiving this tuple, the jth Dup-elim task checks that it had not received it before, since its job is duplicate elimination. If previously received, the tuple is ignored. But if this tuple is new, it is stored locally and sent to two Join tasks, those numbered h(c) and h(d).

44 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

Join task0Join task1Join task i

Dup−elim task0

Dup−elim

To join task h(c) task1

**..... .**

P(c,d) if

Dup−elim

P(a,b) if

g(c,d) = j task

j

P(c,d) if never seen before h(a) = i or h(b) = i

**... ...**

To join task h(d)

Figure 2.7: Implementation of transitive closure by a collection of recursive tasksEvery Join task has m output files – one for each Dup-elim task – and every Dup-elim task has n output files – one for each Join task. These files may be distributed according to any of several strategies. Initially, the E(a, b) tuples representing the arcs of the graph are distributed to the Dup-elim tasks, with E(a, b) being sent as P(a, b) to Dup-elim task g(a, b). The Master can wait until each Join task has processed its entire input for a round. Then, all output files are distributed to the Dup-elim tasks, which create their own output. That output is distributed to the Join tasks and becomes their input for the next round. Alternatively, each task can wait until it has produced enough output to justify transmitting its output files to their destination, even if the task has not consumed all its input. ✷

In Example 2.6 it is not essential to have two kinds of tasks. Rather, Join tasks could eliminate duplicates as they are received, since they must store their previously received inputs anyway. However, this arrangement has an advantage when we must recover from a task failure. If each task stores all the output files it has ever created, and we place Join tasks on different racks from the Dup-elim tasks, then we can deal with any single compute node or

2.4. EXTENSIONS TO MAPREDUCE 45

single rack failure. That is, a Join task needing to be restarted can get all the previously generated inputs that it needs from the Dup-elim tasks, and vice versa.

In the particular case of computing transitive closure, it is not necessary to prevent a restarted task from generating outputs that the original task gener- ated previously. In the computation of the transitive closure, the rediscovery of a path does not influence the eventual answer. However, many computations cannot tolerate a situation where both the original and restarted versions of a task pass the same output to another task. For example, if the final step of the computation were an aggregation, say a count of the number of nodes reached by each node in the graph, then we would get the wrong answer if we counted a path twice. In such a case, the master controller can record what files each task generated and passed to other tasks. It can then restart a failed task and ignore those files when the restarted version produces them a second time.

2.4.3 Pregel

Another approach to managing failures when implementing recursive algorithms on a computing cluster is represented by the Pregel system. This system views its data as a graph. Each node of the graph corresponds roughly to a task (although in practice many nodes of a large graph would be bundled into a single task, as in the Join tasks of Example 2.6). Each graph node generates output messages that are destined for other nodes of the graph, and each graph node processes the inputs it receives from other nodes.

Example 2.7: Suppose our data is a collection of weighted arcs of a graph, and we want to find, for each node of the graph, the length of the shortest path to each of the other nodes. Initially, each graph node a stores the set of pairs (b, w) such that there is an arc from a to b of weight w. These facts are initially sent to all other nodes, as triples (a, b, w).6 When the node a receives a triple (c, d, w), it looks up its current distance to c; that is, it finds the pair (c, v) stored locally, if there is one. It also finds the pair (d, u) if there is one. If w + v<u, then the pair (d, u) is replaced by (d, w + v), and if there was no pair (d, u), then the pair (d, w + v) is stored at the node a. Also, the other nodes are sent the message (a, d, w + v) in either of these two cases. ✷

6This algorithm uses much too much communication, but it will serve as a simple example of the Pregel computation model.

Pregel and Giraph

Like MapReduce, Pregel was developed originally at Google. Also like MapReduce, there is an Apache, open-source equivalent, called Giraph.

46 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

Computations in Pregel are organized into supersteps. In one superstep, all the messages that were received by any of the nodes at the previous superstep (or initially, if it is the first superstep) are processed, and then all the messages generated by those nodes are sent to their destination.

In case of a compute-node failure, there is no attempt to restart the failed tasks at that compute node. Rather, Pregel checkpoints its entire computation after some of the supersteps. A checkpoint consists of making a copy of the entire state of each task, so it can be restarted from that point if necessary. If any compute node fails, the entire job is restarted from the most recent checkpoint.

Although this recovery strategy causes many tasks that have not failed to redo their work, it is satisfactory in many situations. Recall that the reason MapReduce systems support restart of only the failed tasks is that we want assurance that the expected time to complete the entire job in the face of fail- ures is not too much greater than the time to run the job with no failures. Any failure-management system will have that property as long as the time to recover from a failure is much less than the average time between failures. Thus, it is only necessary that Pregel checkpoints its computation after a num- ber of supersteps such that the probability of a failure during that number of supersteps is low.

2.4.4 Exercises for Section 2.4

! Exercise 2.4.1: Suppose a job consists of n tasks, each of which takes time t seconds. Thus, if there are no failures, the sum over all compute nodes of the time taken to execute tasks at that node is nt. Suppose also that the probability of a task failing is p per job per second, and when a task fails, the overhead of management of the restart is such that it adds 10t seconds to the total execution time of the job. What is the total expected execution time of the job?

! Exercise 2.4.2: Suppose a Pregel job has a probability p of a failure during any superstep. Suppose also that the execution time (summed over all compute nodes) of taking a checkpoint is c times the time it takes to execute a superstep. To minimize the expected execution time of the job, how many supersteps should elapse between checkpoints?

2.5 The Communication Cost Model

In this section we shall introduce a model for measuring the quality of algorithms implemented on a computing cluster of the type so far discussed in this chapter. We assume the computation is described by an acyclic workflow, as discussed in Section 2.4.1. For many applications, the bottleneck is moving data among tasks, such as transporting the outputs of Map tasks to their proper Reduce tasks. As an example, we explore the computation of multiway joins as single

2.5. THE COMMUNICATION COST MODEL 47

MapReduce jobs, and we see that in some situations, this approach is more efficient than the straightforward cascade of 2-way joins.

2.5.1 Communication-Cost for Task Networks

Imagine that an algorithm is implemented by an acyclic network of tasks. These tasks could be Map tasks feeding Reduce tasks, as in a standard MapReduce algorithm, or they could be several MapReduce jobs cascaded, or a more general workflow structure, such as a collection of tasks each of which implements the workflow of Fig. 2.6.7 The communication cost of a task is the size of the input to the task. This size can be measured in bytes. However, since we shall be using relational database operations as examples, we shall often use the number of tuples as a measure of size.

The communication cost of an algorithm is the sum of the communication cost of all the tasks implementing that algorithm. We shall focus on the commu- nication cost as the way to measure the efficiency of an algorithm. In particular, we do not consider the amount of time it takes each task to execute when es- timating the running time of an algorithm. While there are exceptions, where execution time of tasks dominates, these exceptions are rare in practice. We can explain and justify the importance of communication cost as follows.

• The algorithm executed by each task tends to be very simple, often linear in the size of its input.

• The typical interconnect speed for a computing cluster is one gigabit per second. That may seem like a lot, but it is slow compared with the speed at which a processor executes instructions. Moreover, in many cluster architectures, there is competition for the interconnect when several com- pute nodes need to communicate at the same time. As a result, the compute node can do a lot of work on a received input element in the time it takes to deliver that element.

• Even if a task executes at a compute node that has a copy of the chunk(s) on which the task operates, that chunk normally will be stored on disk, and the time taken to move the data into main memory may exceed the time needed to operate on the data once it is available in memory.

Assuming that communication cost is the dominant cost, we might still ask why we count only input size, and not output size. The answer to this question involves two points:

1. If the output of one task τ is input to another task, then the size of τ’s output will be accounted for when measuring the input size for the receiv- ing task. Thus, there is no reason to count the size of any output except for those tasks whose output forms the result of the entire algorithm.

7Recall that this figure represented functions, not tasks. As a network of tasks, there would be, for example, many tasks implementing function f, each of which feeds data to each of the tasks for function g and each of the tasks for function i.

48 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

2. But in practice, the algorithm output is rarely large compared with the input or the intermediate data produced by the algorithm. The reason is that massive outputs cannot be used unless they are summarized or aggregated in some way. For example, although we talked in Example 2.6 of computing the entire transitive closure of a graph, in practice we would want something much simpler, such as the count of the number of nodes reachable from each node, or the set of nodes reachable from a single node.

Example 2.8: Let us evaluate the communication cost for the join algorithm from Section 2.3.7. Suppose we are joining R(A, B) ⊲⊳ S(B,C), and the sizes of relations R and S are r and s, respectively. Each chunk of the files holding R and S is fed to one Map task, so the sum of the communication costs for all the Map tasks is r + s. Note that in a typical execution, the Map tasks will each be executed at a compute node holding a copy of the chunk to which it applies. Thus, no internode communication is needed for the Map tasks, but they still must read their data from disk. Since all the Map tasks do is make a simple transformation of each input tuple into a key-value pair, we expect that the computation cost will be small compared with the communication cost, regardless of whether the input is local to the task or must be transported to its compute node.

The sum of the outputs of the Map tasks is roughly as large as their in- puts. Each output key-value pair is sent to exactly one Reduce task, and it is unlikely that this Reduce task will execute at the same compute node. There- fore, communication from Map tasks to Reduce tasks is likely to be across the interconnect of the cluster, rather than memory-to-disk. This communication is O(r + s), so the communication cost of the join algorithm is O(r + s).

The Reduce tasks execute the reducer (application of the Reduce function to a key and its associated value list) for one or more values of attribute B. Each reducer takes the inputs it receives and divides them between tuples that came from R and those that came from S. Each tuple from R pairs with each tuple from S to produce one output. The output size for the join can be either larger or smaller than r + s, depending on how likely it is that a given R-tuple joins with a given S-tuple. For example, if there are many different B-values, we would expect the output to be small, while if there are few B-values, a large output is likely.

If the output is large, then the computation cost of generating all the outputs from a reducer could be much larger than O(r+s). However, we shall rely on our supposition that if the output of the join is large, then there is probably some aggregation being done to reduce the size of the output. It will be necessary to communicate the result of the join to another collection of tasks that perform this aggregation, and thus the communication cost will be at least proportional to the computation needed to produce the output of the join. ✷

2.5. THE COMMUNICATION COST MODEL 49

2.5.2 Wall-Clock Time

While communication cost often influences our choice of algorithm to use in a cluster-computing environment, we must also be aware of the importance of wall-clock time, the time it takes a parallel algorithm to finish. Using careless reasoning, one could minimize total communication cost by assigning all the work to one task, and thereby minimize total communication. However, the wall-clock time of such an algorithm would be quite high. The algorithms we suggest, or have suggested so far, have the property that the work is divided fairly among the tasks. Therefore, the wall-clock time would be approximately as small as it could be, given the number of compute nodes available.

2.5.3 Multiway Joins

To see how analyzing the communication cost can help us choose an algorithm in the cluster-computing environment, we shall examine carefully the case of a multiway join. There is a general theory in which we:

1. Select certain attributes of the relations involved in the natural join of three or more relations to have their values hashed, each to some number of buckets.

2. Select the number of buckets for each of these attributes, subject to the constraint that the product of the numbers of buckets for each attribute is k, the number of reducers that will be used.

3. Identify each of the k reducers with a vector of bucket numbers. These vectors have one component for each of the attributes selected at step (1).

4. Send tuples of each relation to all those reducers where it might find tuples to join with. That is, the given tuple t will have values for some of the attributes selected at step (1), so we can apply the hash function(s) to those values to determine certain components of the vector that identifies the reducers. Other components of the vector are unknown, so t must be sent to reducers for all vectors having any value in these unknown components.

Some examples of this general technique appear in the exercises.

Here, we shall look only at the join R(A, B) ⊲⊳ S(B,C) ⊲⊳ T(C, D) as an example. Suppose that the relations R, S, and T have sizes r, s, and t, respectively, and for simplicity, suppose p is the probability that

1. An R-tuple and and S-tuple agree on B, and also the probability that

2. An S-tuple and a T-tuple agree on C.

If we join R and S first, using the MapReduce algorithm of Section 2.3.7, then the communication cost is O(r + s), and the size of the intermediate join

50 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

R ⊲⊳ S is prs. When we join this result with T, the communication of this second MapReduce job is O(t + prs). Thus, the entire communication cost of the algorithm consisting of two 2-way joins is O(r + s + t + prs). If we instead join S and T first, and then join R with the result, we get another algorithm whose communication cost is O(r + s + t + pst).

A third way to take this join is to use a single MapReduce job that joins the three relations at once. Suppose that we plan to use k reducers for this job. Pick numbers b and c representing the number of buckets into which we shall hash B- and C-values, respectively. Let h be a hash function that sends B-values into b buckets, and let g be another hash function that sends C-values into c buckets. We require that bc = k; that is, each reducer corresponds to a pair of buckets, one for the B-value and one for the C-value. The reducer corresponding to bucket pair (i, j) is responsible for joining the tuples R(u, v), S(v, w), and T(w, x) whenever h(v) = i and g(w) = j.

As a result, the Map tasks that send tuples of R, S, and T to the reducers that need them must send R- and T-tuples to more than one reducer. For an S-tuple the it reducer S(v, for w), (we h(v),g(w)know only needs to go to the ). reducers B- and C-values, so we can send this tuple only to However, that consider an correspond to (R-tuple h(v),y)R(u, , v). We know for some y. But we don’t know y; the value of C could be anything as far as we know. Thus, we must send R(u, v) to c reducers, since y could be any of the c buckets for C-values. (z,g(w)) for Similarly, any z. There we must are send b such the reducers.

T-tuple T(w, x) to each of the reducers 0 1 2 3

0 12g(T.C) = 1

g(C) = h(S.B) = 2 and g(S.C) = 1

h(B) =

h(R.B) = 2 3

Figure 2.8: Sixteen reducers together perform a 3-way join

Example 2.9: Suppose that b = c = 4, so k = 16. The sixteen reducers can be thought of as arranged in a rectangle, as suggested by Fig. 2.8. There, we see a hypothetical S-tuple S(v, w) for which h(v) = 2 and g(w) = 1. This tuple is sent by its Map task only to the reducer for key (2,1). We also see

2.5. THE COMMUNICATION COST MODEL 51

an R-tuple R(u, v). Since h(v) = 2, this tuple is sent to all reducers (2,y), for y = 1,2,3,4. Finally, we see a T-tuple T(w, x). Since g(w) = 1, this tuple is sent to all reducers (z,1) for z = 1,2,3,4. Notice that these three tuples join, and they meet at exactly one reducer, the reducer for key (2,1). ✷

Now, suppose that the sizes of R, S, and T are different; recall we use r, s, and t, respectively, for those sizes. If we hash B-values to b buckets and C-values to c buckets, where bc = k, then the total communication cost for moving the tuples to the proper reducers is the sum of:

1. s to move each tuple S(v, w) once to the reducer (h(v),g(w)). 2. cr to move each tuple R(u, v) to the c reducers (h(v),y) for each of the c

possible values of y. 3. bt to move each tuple T(w, x) to the b reducers (z,g(w)) for each of the

b possible values of z.

There is also a cost r + s + t to make each tuple of each relation be input to one of the Map tasks. This cost is fixed, independent of b, c, and k.

We must select b and c, subject to the constraint bc = k, to minimize s + cr + bt. We shall use the technique of Lagrangean multipliers to find the place where the function s + cr + bt − λ(bc − k) has its derivatives with respect to b and c equal to 0. That is, we must solve the equations r − λb = 0 and t these Thus, and − λc b equations = the = r/λ 0. minimum Since = √

to r get = communication rt λb = and λ2bc. t = Since λc, cost we bc is may = obtained k, we multiply get when rt corresponding = λc 2k, = or t/λ λ = = sides √

√

rt/k. kt/r, of

krt. That is the communication cost for the Reduce tasks, to which we must add the cost s + r + t for the communication cost of the Map tasks. The total

Computation Cost of the 3-Way Join

Each of the reducers must join of parts of the three relations, and it is reasonable to ask whether this join can be taken in time that is linear in the size of the input to that Reduce task. While more complex joins might not be computable in linear time, the join of our running example can be executed at each Reduce process efficiently. First, create an index on R.B, to organize the R-tuples received. Likewise, create an index on T.C for the T-tuples. Then, consider each received S-tuple, S(v, w). Use the index on R.B to find all R-tuples with R.B = v and use the index on T.C to find all T-tuples with T.C = w.

kr/t. If we substitute these values into the formula s + cr + bt, we get s + 2√

52 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

communication neglect r + t, because cost is it thus will r be + 2s less + than t + 2√

2√

krt. In most circumstances, we can k).

Example 2.10: Let us see under what circumstances the 3-way join has lower communication cost than the cascade of two 2-way joins. To make matters simple, let us assume that R, S, and T are all the same relation R, which represents the “friends” relation in a social network like Facebook. There are roughly a billion subscribers on Facebook, with an average of 300 friends each, so relation R has r = 3 × 1011 tuples. Suppose we want to compute R ⊲⊳ R ⊲⊳ R, perhaps as part of a calculation to find the number of friends of friends of friends each subscriber has, or perhaps just the person with the largest number of 4r friends + 2r√

of friends of friends.8 The cost of the 3-way join of R with itself is kr2 is the cost of 1011the √

k. Reduce tasks. Since we assume r = 3×1011, this cost is 1,2×1012 +6×

Now consider the communication cost of joining R with itself, and then joining the result with R again. The Map and Reduce tasks for the first join each have a cost of 2r, so the first join only has communication cost 4r = 1.2×1012. But the size of R ⊲⊳ R is large. We cannot say exactly how large, since friends tend to fall into cliques, and therefore a person with 300 friends will have many fewer than the maximum possible number of friends of friends, which is 90,000. Let us estimate conservatively that the size of R ⊲⊳ R is not 300r, but only 30r, or 9 × 1012. The communication cost for the second join of (R ⊲⊳ R) ⊲⊳ R is thus 1.8 × 1013 + 6 × 1011. The total cost of the two joins is therefore 1.2 × 1012 + 1.8 × 1013 + 6 × 1011 = 1.98 × 1013.

We must ask whether the cost of the 3-way join, which is

1.2 × 1012 + 6 × 1011√

k

is √

k less < than 1.98 × 1013. That is so, provided 6 × 1011√

k < 1.86 × 1013, or 31. That is, the 3-way join will be preferable provided we use no more than 312 = 961 reducers. ✷

2.5.4 Exercises for Section 2.5

Exercise 2.5.1: What is the communication cost of each of the following algorithms, as a function of the size of the relations, matrices, or vectors to which they are applied?

(a) The matrix-vector multiplication algorithm of Section 2.3.2.

(b) The union algorithm of Section 2.3.6.

(c) The aggregation algorithm of Section 2.3.8.

8This person, or more generally, people with large extended circles of friends, are good people to use to start a marketing campaign by giving them free samples.

k; 3r represents the cost of the Map tasks, and r + 2√

krt, usually by a factor of O(√

(d) The matrix-multiplication algorithm of Section 2.3.10.

! Exercise 2.5.2: Suppose relations R, S, and T have sizes r, s, and t, respec- tively, and we want to take the 3-way join R(A, B) ⊲⊳ S(B,C) ⊲⊳ T(A, C), using k reducers. We shall hash values of attributes A, B, and C to a, b, and c buckets, respectively, where abc = k. Each reducer is associated with a vector of buckets, one for each of the three hash functions. Find, as a function of r, s, t, and k, the values of a, b, and c that minimize the communication cost of the algorithm.

! Exercise 2.5.3: Suppose we take a star join of a fact table F(A1,A2,...,Am) with dimension tables Di(Ai,Bi) for i = 1,2,...,m. Let there be k reducers, each associated with a vector of buckets, one for each of the key attributes A1,A2,...,Am. Suppose the number of buckets into which we hash Ai is ai.

2.5. THE COMMUNICATION COST MODEL 53

Star Joins

A common structure for data mining of commercial data is the star join. For example, a chain store like Walmart keeps a fact table whose tu- ples each represent a single sale. This relation looks like F(A1,A2,...), where each attribute Ai is a key representing one of the important com- ponents of the sale, such as the purchaser, the item purchased, the store branch, or the date. For each key attribute there is a dimension table giving information about the participant. For instance, the dimension ta- ble D(A1,B11,B12,...) might represent purchasers. A1 is the purchaser ID, the key for this relation. The B1i’s might give the purchaser’s name, address, phone, and so on. Typically, the fact table is much larger than the dimension tables. For instance, there might be a fact table of a billion tuples and ten dimension tables of a million tuples each.

Analysts mine this data by asking analytic queries that typically join the fact table with several of the dimension tables (a “star join”) and then aggregate the result into a useful form. For instance, an analyst might ask “give me a table of sales of pants, broken down by region and color, for each month of 2012.” Under the communication-cost model of this section, joining the fact table and dimension tables by a multiway join is almost certain to be more efficient than joining the relations in pairs. In fact, it may make sense to store the fact table over however many compute nodes are available, and replicate the dimension tables permanently in exactly the same way as we would replicate them should we take the join of the fact table and all the dimension tables. In this special case, only the key attributes (the A’s above) are hashed to buckets, and the number of buckets for each key attribute is proportional to the size of its dimension table.

54 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

Naturally, a1a2 ···am = k. Finally, suppose each dimension table Di has size di, and the size of the fact table is much larger than any of these sizes. Find the values of the ai’s that minimize the cost of taking the star join as one MapReduce operation.

2.6 Complexity Theory for MapReduce

Now, we shall explore the design of MapReduce algorithms in more detail. Sec- tion 2.5 introduced the idea that communication between the Map and Reduce tasks often accounts for the largest fraction of the time spent by these tasks. Here, we shall look at how the communication cost relates to other desiderata for MapReduce algorithms, in particular our desire to shrink the wall-clock time and to execute each reducer in main memory. Recall that a “reducer” is the execution of the Reduce function on a single key and its associated value list. The point of the exploration in this section is that for many problems there is a spectrum of MapReduce algorithms requiring different amounts of communica- tion. Moreover, the less communication an algorithm uses, the worse it may be in other respects, including wall-clock time and the amount of main memory it requires.

2.6.1 Reducer Size and Replication Rate

Let us now introduce the two parameters that characterize families of MapRe- duce algorithms. The first is the reducer size, which we denote by q. This parameter is the upper bound on the number of values that are allowed to ap- pear in the list associated with a single key. Reducer size can be selected with at least two goals in mind.

1. By making the reducer size small, we can force there to be many reducers, i.e., many different keys according to which the problem input is divided by the Map tasks. If we also create many Reduce tasks – even one for each reducer – then there will be a high degree of parallelism, and we can look forward to a low wall-clock time.

2. We can choose a reducer size sufficiently small that we are certain the computation associated with a single reducer can be executed entirely in the main memory of the compute node where its Reduce task is located. Regardless of the computation done by the reducers, the running time will be greatly reduced if we can avoid having to move data repeatedly between main memory and disk.

The second parameter is the replication rate, denoted r. We define r to be the number of key-value pairs produced by all the Map tasks on all the inputs, divided by the number of inputs. That is, the replication rate is the average communication from Map tasks to Reduce tasks (measured by counting key-value pairs) per input.

2.6. COMPLEXITY THEORY FOR MAPREDUCE 55

Example 2.11: Let us consider the one-pass matrix-multiplication algorithm of Section 2.3.10. Suppose that all the matrices involved are n × n matrices. Then the replication rate r is equal to n. That fact is easy to see, since for each element mij, there are n key-value pairs produced; these have all keys of the form (i, k), for 1 ≤ k ≤ n. Likewise, for each element of the other matrix, say njk, we produce n key-value pairs, each having one of the keys (i, k), for 1 ≤ i ≤ n. In this case, not only is n the average number of key-value pairs produced for an input element, but each input produces exactly this number of pairs.We also see that q, the required reducer size, is 2n. That is, for each key (i, k), there are n key-value pairs representing elements mij of the first matrix and another n key-value pairs derived from the elements njk of the second matrix. While this pair of values represents only one particular algorithm for one-pass matrix multiplication, we shall see that it is part of a spectrum of algorithms, and in fact represents an extreme point, where q is as small as can be, and r is at its maximum. More generally, there is a tradeoff between r and q, that can be expressed as qr ≥ 2n2. ✷

2.6.2 An Example: Similarity Joins

To see the tradeoff between r and q in a realistic situation, we shall examine a problem known as similarity join. In this problem, we are given a large set of elements X and a similarity measure s(x, y) that tells how similar two elements x and y of set X are. In Chapter 3 we shall learn about the most important notions of similarity and also learn some tricks that let us find similar pairs quickly. But here, we shall consider only the raw form of the problem, where we have to look at each pair of elements of X and determine their similarity by applying the function s. We assume that s is symmetric, so s(x, y) = s(y,x), but we assume nothing else about s. The output of the algorithm is those pairs whose similarity exceeds a given threshold t.

For example, let us suppose we have a collection of one million images, each of size one megabyte. Thus, the dataset has size one terabyte. We shall not try to describe the similarity function s, but it might, say, involve giving higher values when images have roughly the same distribution of colors or when images have corresponding regions with the same distribution of colors. The goal would be to discover pairs of images that show the same type of object or scene. This problem is extremely hard, but classifying by color distribution is generally of some help toward that goal.

Let us look at how we might do the computation using MapReduce to exploit the natural parallelism found in this problem. The input is key-value pairs (i, Pi), where i is an ID for the picture and Pi is the picture itself. We want to compare each pair of pictures, so let us use one key for each set of two ID’s {i, j}. There are approximately 5 × 1011 pairs of two ID’s. We want each key {i, j} to be associated with the two values Pi and Pj, so the input to the corresponding reducer will be ({i, j}, [Pi,Pj]). Then, the Reduce function can

56 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

simply apply the similarity function s to the two pictures on its value list, that is, compute s(Pi,Pj), and decide whether the similarity of the two pictures is above threshold. The pair would be output if so.

Alas, this algorithm will fail completely. The reducer size is small, since no list has more than two values, or a total of 2MB of input. Although we don’t know exactly how the similarity function s operates, we can reasonably expect that it will not require more than the available main memory. However, the replication rate is 999,999, since for each picture we generate that number of key-value pairs, one for each of the other pictures in the dataset. The total number of bytes communicated from Map tasks to Reduce tasks is 1,000,000 (for the pictures) times 999,999 (for the replication), times 1,000,000 (for the size of each picture). That’s 1018 bytes, or one exabyte. To communicate this amount of data over gigabit Ethernet would take 1010 seconds, or about 300 years.9

Fortunately, this algorithm is only the extreme point in a spectrum of possi- ble algorithms. We can characterize these algorithms by grouping pictures into g groups, each of 106/g pictures. The Map Function: Take an input element (i, Pi) and generate g − 1 key- value pairs. For each, the key is one of the sets {u, v}, where u is the group to which picture i belongs, and v is one of the other groups. The associated value is the pair (i, Pi). The Reduce Function: Consider the key {u, v}. The associated value list will have the 2 × 106/g elements (j, Pj), where j belongs to either group u or group v. The Reduce function takes each (i, Pi) and (j, Pj) on this list, where i and j belong to different groups, and applies the similarity function s(Pi,Pj). In addition, we need to compare the pictures that belong to the same group, but we don’t want to do the same comparison at each of the g − 1 reducers whose key contains a given group number. There are many ways to handle this problem, but one way is as follows. Compare the members of group u at the reducer {u, u + 1}, where the “+1” is taken in the end-around sense. That is, if u = g (i.e., u is the last group), then u+ 1 is group 1. Otherwise, u+ 1 is the group whose number is one greater than u.

We can compute the replication rate and reducer size as a function of the number of groups g. Each input element is turned into g − 1 key-value pairs. That is, the replication rate is g − 1, or approximately r = g, since we suppose that the number of groups is still fairly large. The reducer size is 2×106/g, since that is the number of values on the list for each reducer. Each value is about a megabyte, so the number of bytes needed to store the input is 2 × 1012/g.

Example 2.12: If g is 1000, then the input consumes about 2GB. That’s enough to hold everything in a typical main memory. Moreover, the total

9In a typical cluster, there are many switches connecting subsets of the compute nodes, so all the data does not need to go across a single gigabit switch. However, the total available communication is still small enough that it is not feasible to implement this algorithm for the scale of data we have hypothesized.

2.6. COMPLEXITY THEORY FOR MAPREDUCE 57

number of bytes communicated is now 106 × 999 × 106, or about 1015 bytes. While that is still a huge amount of data to communicate, it is 1000 times less than that of the obvious algorithm. Moreover, there are still about half a million reducers. Since we are unlikely to have available that many compute nodes, we can divide all the reducers into a smaller number of Reduce tasks and still keep all the compute nodes busy; i.e., we can get as much parallelism as our computing cluster offers us. ✷

The computation cost for algorithms in this family is independent of the number of groups g, as long as the input to each reducer fits in main memory. The reason is that the bulk of the computation is the application of function s to the pairs of pictures. No matter what value g has, s is applied to each pair once and only once. Thus, although the work of algorithms in the family may be divided among reducers in widely different ways, all members of the family do the same computation.

2.6.3 A Graph Model for MapReduce Problems

In this section, we begin the study of a technique that will enable us to prove lower bounds on the replication rate, as a function of reducer size for a number of problems. Our first step is to introduce a graph model of problems. For each problem solvable by a MapReduce algorithm there is:

1. A set of inputs.

2. A set of outputs.

3. A many-many relationship between the inputs and outputs, which de-

scribes which inputs are necessary to produce which outputs.

Example 2.13: Figure 2.9 shows the graph for the similarity-join problem discussed in Section 2.6.2, if there were four pictures rather than a million. The inputs are the pictures, and the outputs are the six possible pairs of pictures. Each output is related to the two inputs that are members of its pair. ✷

Example 2.14: Matrix multiplication presents a more complex graph. If we multiply n × n matrices M and N to get matrix P, then there are 2n2 inputs, mij and njk, and there are n2 outputs pik. Each output pik is related to 2n inputs: mi1,mi2,...,min and n1k,n2k,...,nnk. Moreover, each input is related to n outputs. For example, mij is related to pi1,pi2,...,pin. Figure 2.10 shows the input-output relationship for matrix multiplication for the simple case of 2 × 2 matrices, specifically

[ a b c d

][ e f g h

]

]

=

=

[ i j k l

[ i j k l

[ i j k l

[ i j k l

]

]

]

]

✷

58 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

*P* 1*P* 2*P* 3*P* 4

{ *P*1, *P* 2}

{ *P*1, *P* 3}

{ *P*1

, *P* 4

}

{ *P*2, }

{ *P*2

, }

{ , }

Figure 2.9: Input-output relationship for a similarity join

In the problems of Examples 2.13 and 2.14, the inputs and outputs were clearly all present. However, there are other problems where the inputs and/or outputs may not all be present in any instance of the problem. An example of such a problem is the natural join of R(A, B) and S(B,C) discussed in Section 2.3.7. We assume the attributes A, B, and C each have a finite domain, so there are only a finite number of possible inputs and outputs. The inputs are all possible R-tuples, those consisting of a value from the domain of A paired with a value from the domain of B, and all possible S-tuples – pairs from the domains of B and C. The outputs are all possible triples, with components from the domains of A, B, and C in that order. The output (a, b, c) is connected to two inputs, namely R(a, b) and S(b, c).

But in an instance of the join computation, only some of the possible inputs will be present, and therefore only some of the possible outputs will be produced. That fact does not influence the graph for the problem. We still need to know how every possible output relates to inputs, whether or not that output is produced in a given instance.

2.6.4 Mapping Schemas

Now that we see how to represent problems addressable by MapReduce as graphs, we can define the requirements for a MapReduce algorithm to solve a given problem. Each such algorithm must have a mapping schema, which expresses how outputs are produced by the various reducers used by the algo- rithm. That is, a mapping schema for a given problem with a given reducer size q is an assignment of inputs to one or more reducers, such that:

1. No reducer is assigned more than q inputs.

*P* 3*P* 4

*P*3 *P* 4

2.6. COMPLEXITY THEORY FOR MAPREDUCE 59

*abcdefgh*

*ijkl*

Figure 2.10: Input-output relationship for matrix multiplication

2. For every output of the problem, there is at least one reducer that is assigned all the inputs that are related to that output. We say this reducer covers the output.

It can be argued that the existence of a mapping schema for any reducer size is what distinguishes problems that can be solved by a single MapReduce job from those that cannot.

Example 2.15: Let us reconsider the “grouping” strategy we discussed in connection with the similarity join in Section 2.6.2. To generalize the problem, suppose the input is p inputs each. The number reducer will get the inputs pictures, of from outputs two which groups is we (p2)place – , in g or approximately that is 2p/g equal-sized groups of p/g p2/2 outputs. A inputs – so the reducer size we need is q = 2p/g. Each picture is sent to the reducers corresponding to the pairs consisting of its group and any of the g − 1 other groups. Thus, the replication rate is g − 1, or approximately g. If we replace g by the replication rate r in q = 2p/g, we conclude that r = 2p/q. That is, the replication rate is inversely proportional to the reducer size. That relationship is common; the smaller the reducer size, the larger the replication rate, and therefore the higher the communication. This family of algorithms is described by a family for each possible q. In the mapping schema for q approximately g2/2 reducers. Each reducer corresponds of mapping = 2p/g, to schemas, there are a pair of groups, (g2), one or

and an input P is assigned to all the reducers whose pair includes the group of P. Thus, no reducer is assigned more than 2p/g inputs; in fact each reducer is assigned exactly that number. Moreover, every output is covered by some reducer. Specifically, if the output is a pair from two different groups u and v, then this output is covered by the reducer for the pair of groups {u, v}. If the

60 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

output corresponds to inputs from only one group u, then the output is covered by several reducers – those corresponding to the set of groups {u, v} for any v = u. Note that the algorithm we described has only one of these reducers computing the output, but any of them could compute it. ✷

The fact that an output depends on a certain input means that when that input is processed at the Map task, there will be at least one key-value pair generated to be used when computing that output. The value might not be exactly the input (as was the case in Example 2.15), but it is derived from that input. What is important is that for every related input and output there is a unique key-value pair that must be communicated. Note that there is technically never a need for more than one key-value pair for a given input and output, because the input could be transmitted to the reducer as itself, and whatever transformations on the input were applied by the Map function could instead be applied by the Reduce function at the reducer for that output.

2.6.5 When Not All Inputs Are Present

Example 2.15 describes a problem where we know every possible input is pre- sent, because we can define the input set to be those pictures that actually exist in the dataset. However, as discussed at the end of Section 2.6.3, there are problems like computing the join, where the graph of inputs and outputs describes inputs that might exist, and outputs that are only made when at least one of the inputs exists in the dataset. In fact, for the join, both inputs related to an output must exist if we are to make that output.

An algorithm for a problem where outputs can be missing still needs a mapping schema. The justification is that all inputs, or any subset of them, might be present, so an algorithm without a mapping schema would not be able to produce every possible output if all the inputs related to that output happened to be present, and yet no reducer covered that output.

The only way the absence of some inputs makes a difference is that we may wish to rethink the desired value of the reducer size q when we select an algorithm from the family of possible algorithms. Especially, if the value of q we select is that number such that we can be sure the input will just fit in main memory, then we may wish to increase q to take into account that some fraction of the inputs are not really there.

Example 2.16: Suppose that we know we can execute the Reduce function in main memory on a key and its associated list of q values. However, we also know that only 5% of the possible inputs are really present in the data set. Then a mapping schema for reducer size q will really send about q/20 of the inputs that exist to each reducer. Put another way, we could use the algorithm for reducer size 20q and expect that an average of q inputs will actually appear on the list for each reducer. We can thus choose 20q as the reducer size, or since there will be some randomness in the number of inputs actually appearing at

2.6. COMPLEXITY THEORY FOR MAPREDUCE 61

each reducer, we might wish to pick a slightly smaller value of reducer size, such as 18q. ✷

2.6.6 Lower Bounds on Replication Rate

The family of similarity-join algorithms described in Example 2.15 lets us trade off communication against the reducer size, and through reducer size to trade communication against parallelism or against the ability to execute the Reduce function in main memory. How do we know we are getting the best possible tradeoff? We can only know we have the minimum possible communication if we can prove a matching lower bound. Using existence of a mapping schema as the starting point, we can often prove such a lower bound. Here is an outline of the technique.

1. Prove an upper bound on how many outputs a reducer with q inputs can cover. Call this bound g(q). This step can be difficult, but for examples like similarity join, it is actually quite simple.

2. Determine the total number of outputs produced by the problem.

3. Suppose Observe computed that that in step there ∑ki=1 (2).

are g(qi) k reducers, must and the ith reducer has qi < q inputs. be no less than the number of outputs

4. Manipulate the inequality from (3) to get Often, the trick used at this step is to replace a some lower factors bound of on qi ∑by ki=1 their qi.

upper bound q, but leave a single factor of qi in the term for i. 5. Since divide ∑the ki=1 lower qi is the bound total from communication (4) on this quantity from Map by tasks the number to Reduce of inputs. tasks,

The result is a lower bound on the replication rate.

Example 2.17: This sequence of steps may seem mysterious, but let us con- sider the similarity join as an example that we hope will make things clear. Recall that in Example 2.15 we gave an upper bound on the replication rate r of r ≤ 2p/q, where p was the number of inputs and q was the reducer size. We shall show a lower bound on r that is half that amount, which implies that, although improvements to the algorithm might be possible, any reduction in communication for a given reducer size will be by a factor of 2 at most. than total inequality For (of q2)step (, p2or )constructed , (1), or approximately approximately observe at that step q2if /2 (3) p2a /2 outputs. is reducer thus

outputs gets For that q step inputs, each (2), must it we cannot know be cover more there are a covered. The

∑ki=1

qi 2/2 ≥ p2/2

62 CHAPTER 2. MAPREDUCE AND THE NEW SOFTWARE STACK

or, multiplying both sides by 2,

∑ki=1

qi 2≥ p2 (2.1)

Now, we must do the manipulation of step (4). Following the hint, we note that there are two factors of qi in each term on the left of Equation (2.1), so we replace one factor by q and leave the other as qi. Since q ≥ qi, we can only increase the left side by doing so, and thus the inequality continues to hold:

q

∑ki=1

qi ≥ p2

qi ≥ p2

or, dividing by q:

∑ki=1

qi ≥ p2/q (2.2)

p, to The final step, which is step (5), is to divide both the number of inputs. As a result, the left side, which the replication rate, and the right side becomes p/q. sides That is (∑of is, ki=1 Equation we qi)/p have is proved 2.2 equal by

the lower bound on r:

r ≥ p/q

As claimed, this shows that the family of algorithms from Example 2.15 all have a replication rate that is at most twice the lowest possible replication rate. ✷

2.6.7 Case Study: Matrix Multiplication

In this section we shall apply the lower-bound technique to one-pass matrix- multiplication algorithms. We saw one such algorithm in Section 2.3.10, but that is only an extreme case of a family of possible algorithms. In particular, for that algorithm, a reducer corresponds to a single element of the output matrix. Just as we grouped inputs in the similarity-join problem to reduce the communication at the expense of a larger reducer size, we can group rows and columns of the two input matrices into bands. Each pair consisting of a band of rows of the first matrix and a band of columns of the second matrix is used by one reducer to produce a square of elements of the output matrix. An example is suggested by Fig. 2.11.

In more detail, suppose we want to compute MN = P, and all three matrices are n × n. Group the rows of M into g bands of n/g rows each, and group the columns of N into g bands of n/g columns each. This grouping is as suggested by Fig. 2.11. Keys correspond to two groups (bands), one from M and one from N. The Map Function: For each element of M, the Map function generates g key-value pairs. The value in each case is the element itself, together with its