

Wiley Precise Textbook Series

Big Data Analytics

**Radha Shankarmani
M. Vijayalakshmi**



Big Data Analytics

Big Data Analytics

Dr. Radha Shankarmani

Prof. & HOD, Dept. of Information Technology,
Sardar Patel Institute Of Technology,
Affiliated to Mumbai University,
Andheri-West, Mumbai

Dr. M. Vijayalakshmi

Professor, Department of Information Technology, VESIT
Vivekanand Education Society Institute of Technology,
Affiliated to Mumbai University

WILEY

Big Data Analytics

Copyright © 2016 by Wiley India Pvt. Ltd., 4435-36/7, Ansari Road, Daryaganj, New Delhi-110002.

Cover Image: © yienkeat/Shutterstock

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or scanning without the written permission of the publisher.

Limits of Liability: While the publisher and the author have used their best efforts in preparing this book, Wiley and the author make no representation or warranties with respect to the accuracy or completeness of the contents of this book, and specifically disclaim any implied warranties of merchantability or fitness for any particular purpose. There are no warranties which extend beyond the descriptions contained in this paragraph. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided herein and the opinions stated herein are not guaranteed or warranted to produce any particular results, and the advice and strategies contained herein may not be suitable for every individual. Neither Wiley India nor the author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Disclaimer: The contents of this book have been checked for accuracy. Since deviations cannot be precluded entirely, Wiley or its author cannot guarantee full agreement. As the book is intended for educational purpose, Wiley or its author shall not be responsible for any errors, omissions or damages arising out of the use of the information contained in the book. This publication is designed to provide accurate and authoritative information with regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services.

Trademarks: All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders. Wiley is not associated with any product or vendor mentioned in this book.

Other Wiley Editorial Offices:

John Wiley & Sons, Inc. 111 River Street, Hoboken, NJ 07030, USA

Wiley-VCH Verlag GmbH, Pappellaee 3, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 1 Fusionopolis Walk #07-01 Solaris, South Tower, Singapore 138628

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada, M9W 1L1

Edition: 2016

ISBN: 978-81-265-5865-0

ISBN: 978-81-265-8224-2 (ebk)

www.wileyindia.com

Printed at:

*Dedicated to my husband, Shankaramani,
and son Rohit who were kind enough to
understand my busy schedule and patiently waited
for a holiday together.*

—Radha Shankarmani

*Dedicated to my family who steadfastly supported me and
my crazy schedule during the months of writing this book –
my mother-in-law Mrs G. Sharada, my husband G. Murlidhar and
my elder son Goutam. A special dedication to Pranav my long
suffering younger son, whose computer I hijacked to speedup
my writing for the last two months.*

—M. Vijayalakshmi

Preface

Importance of Big Data Analytics

The recent explosion of digital data has made organizations to learn more about their businesses, and directly use that knowledge for improved decision making and performance. When shopping moved online, understanding of customers by business managers increased tremendously. E-business not only could track what customers bought, but also track the way they navigated through the site; how much they are influenced by price discounts; review of products bought by their family and friends. The whole look and feel of the contents and its organization in the site is valuable. This information was not available to business managers a decade ago, and the sale-predictions were restricted to the buying pattern of their customers by looking into their past records.

What is New in Analytics?

Analytics can be reactive or proactive. In traditional methods, reactive analytics are done through business intelligence tools and OLAP. For proactive analytics techniques like optimization, predictive modeling, text mining, forecasting and statistical analysis are used. But these tools and techniques cannot be used for Big Data Analytics.

In case of big data, volume, variety and velocity are the three main drivers that gave a new dimension to the way analytics had to be performed. For instance, in Walmart, data collected cross the Internet every hour from its customer transactions is in the range of petabytes. The speed in which data is created is more important than the volume. Real-time or nearly real-time analysis makes a company more agile to face the demand and competitions. For example, a lot of decisions have to be made in real time during the sale on Thanksgiving day. Big data takes the form of messages and images posted to social networks; readings from sensors; GPS signals from cell phones; and more. There is huge volume and variety of information available from social networks, namely Facebook and Twitter. Mobile devices provide streams of data related to people, activities and locations. All the above said data are unstructured and so cannot be stored in structured databases.

To process large amount of unstructured data, technologies and practices were developed. The first step is to fragment such data and store it in a cluster of commodity servers. Here computing is moved to data and not otherwise. The activities of the commodity servers are coordinated by an open-source

software framework called Hadoop. A NoSQL database is used to capture and store the reference data that are diverse in format and also change frequently.

With the advent of Big Data, applying existing traditional data mining algorithms to current real-world problems faces several tough challenges due to the inadequate scalability and other limitations of these algorithms. The biggest limitation is the inability of these existing algorithms to match the three Vs of the emerging big data. Not only the scale of data generated today is unprecedented, the produced data is often continuously generated in the form of high-dimensional streams which require decisions just in time. Further, these algorithms were not designed to be applicable to current areas like web based analytics, social network analysis, etc.

Thus, even though big data bears greater value (i.e., hidden knowledge and more valuable insights), it brings tremendous challenges to extract these hidden knowledge and insights from big data since the established process of knowledge discovering and data mining from conventional datasets was not designed to and will not work well with big data.

One solution to the problem is to improve existing techniques by applying massive parallel processing architectures and novel distributed storage systems which help faster storage and retrieval of data. But this is not sufficient for mining these new data forms. The true solution lies in designing newer and innovative mining techniques which can handle the three V's effectively.

Intent of the Book

The book focuses on storage and processing of Big Data in the first four chapters and discusses newer mining algorithms for analytics in the rest of the chapters.

The first four chapter focus on the business drivers for Big Data Analytics, Hadoop distributed file system, Hadoop framework and Hadoop eco-systems. The four main architectural patterns for storing Big Data and its variations are also discussed.

The latter chapters cover extensions and innovations to traditional data mining like clustering and frequent itemset mining. The book further looks at newer data forms like web-based data, social network data and discusses algorithms to effectively mine them. One popular practical application of Big Data Analytics, Recommendations systems, is also studied in great detail.

Journey of the Reader

The book takes through the theoretical and practical approach to teaching readers the various concepts of Big Data management and analytics. Readers are expected to be aware of traditional classification, clustering and frequent pattern mining based algorithms. The laboratory exercises are given at the end of relevant chapters will help them to perform laboratory exercises. The readers are expected to have knowledge of database management and data mining concepts. The review questions and/or exercises

given at the end of the chapters can be used to test the readers understanding of the content provided in the chapter. Further, a list of suggested programming assignments can be used by a mature reader to gain expertise in this field.

Organization of the Book

1. Chapter 1 contains introduction to Big Data, Big Data Characteristics, Types of Big Data, comparison of Traditional and Big Data Business Approach, Case Study of Big Data Solutions.
2. Chapter 2 contains introduction to Hadoop, Core Hadoop Components, Hadoop Ecosystem, Physical Architecture, and Hadoop Limitations.
3. Chapter 3 discusses about No SQL, NoSQL Business Drivers, Case Studies on NoSQL, No SQL Data Architecture Patterns, Variations of NoSQL Architectural Patterns, Using NoSQL to Manage Big Data, Understanding Types of Big Data Problems, Analyzing Big Data with a Shared-Nothing Architecture, Choosing Distribution Models, Master-Slave vs Peer-to-Peer, the way NoSQL System Handles Big Data Problems.
4. Chapter 4 covers MapReduce and Distributed File Systems; Map Reduce: The Map Tasks and The Reduce Tasks; MapReduce Execution, Coping with Node Failures, Algorithms Using MapReduce: Matrix-Vector Multiplication and Relational Algebra operations.
5. Chapter 5 introduces the concept of similarity between items in a large dataset which is the foundation for several big data mining algorithms like clustering and frequent itemset mining. Different measures are introduced so that the reader can apply the appropriate distance measure to the given application.
6. Chapter 6 introduces the concept of a data stream and the challenges it poses. The chapter looks at a generic model for a stream-based management system. Several Sampling and Filtering techniques which form the heart of any stream mining technique are discussed; among them the popularly used is Bloom filter. Several popular steam-based algorithms like Counting Distinct Elements in a Stream, Counting Ones in a Window, Query Processing in a Stream are discussed.
7. Chapter 7 introduces the concept of looking at the web in the form of a huge webgraph. This chapter discusses the ill effects of “Spam” and looks at Link analysis as a way to combat text bead “Spam”. The chapter discusses Google’s PageRank algorithm and its variants in detail. The alternate ranking algorithm HITS is also discussed. A brief overview of Link spam and techniques to overcome them are also provided.
8. Chapter 8 covers very comprehensively algorithms for Frequent Itemset Mining which is at the heart of any analytics effort. The chapter reviews basic concepts and discusses improvements to the popular A-priori algorithm to make it more efficient. Several newer big data frequent itemset mining algorithms like PCY, Multihash, Multistage algorithms are discussed. Sampling-based

algorithms are also dealt with. The chapter concludes with a brief overview of identifying frequent itemsets in a data stream.

9. Chapter 9 covers clustering which is another important data mining technique. Traditional clustering algorithms like partition-based and hierarchical are insufficient to handle the challenges posed by Big Data clustering. This chapter discusses two newer algorithms, BFR and CURE, which can cluster big data effectively. The chapter provides a brief overview of stream clustering.
10. Chapter 10 discusses Recommendation Systems, A Model for Recommendation Systems, Content-Based Recommendations and Collaborative Filtering.
11. Chapter 11 introduces the social network and enumerates different types of networks and their applications. The concept of representing a Social Network as a Graph is introduced. Algorithms for identifying communities in a social graph and counting triangles in a social graph are discussed. The chapter introduces the concept of SimRank to identify similar entities in a social network.
12. **Appendix:** This book also provides a rather comprehensive list of websites which contain open datasets that the reader can use to understand the concept and use in their research on Big Data Analytics.
13. Additionally each chapter provides several exercises based on the chapters and also several programming assignments that can be used to demonstrate the concepts discussed in the chapters.
14. References are given for detail reading of the concepts in most of the chapters.

Audience

This book can be used to teach a first course on Big Data Analytics in any senior undergraduate or graduate course in any field of Computer Science or Information Technology. Further it can also be used by practitioners and researchers as a single source of Big Data Information.

Acknowledgements

First and foremost, I would like to thank my mother for standing beside me throughout my career and writing this book. My sincere thanks to Principal, Dr. Prachi Gharpure, too. She has been my inspiration and motivation for continuing to improve my knowledge and move my career forward. My thanks to M.E. research students in writing installation procedures for laboratory exercises.

Radha Shankarmani

Several people deserve my gratitude for their help and guidance in making this book a reality. Foremost among them is Prof. Radha Shankaramani, my co-author who pushed and motivated me to start this venture. My sincere thanks to my principal Dr. J.M. Nair (VESIT) who has supported me full heartedly in this venture. My thanks to Amey Patankar and Raman Kandpal of Wiley India for mootting the idea of this book in the first place.

M. Vijayalakshmi

Together,

We would like to express our gratitude to the many people who inspired us and provided support.

Our sincere thanks to the Dean, Ad hoc Board of Studies, Information Technology, Dr. Bakal for introducing the course in under graduate program and providing us an opportunity to take this venture. Our sincere thanks to the publishers, Wiley India and the editorial team for their continuing support in publishing this book.

Radha Shankarmani

M. Vijayalakshmi

About the Authors



Dr. Radha Shankarmani is currently working as Professor and Head at Department of Information Technology, Sardar Patel Institute of Technology, Mumbai. Her areas of interest include Business Intelligence, Software Engineering, Software Testing, Databases, Data Warehousing and Mining, Computer Simulation and Modeling, Management Information System and SOA. Dr. Radha Shankarmani holds a PhD degree from JNTUH; Masters degree in Computer Science and Engineering from NIT, Trichy and Bachelors degree from PSG College of Technology in Electronics and Communication Engineering. She has more than 20 years of teaching experience and 4 years of industry experience where she has held designations such as Programmer, Software Engineer and Manager. She did her sabbaticals for two months in Infosys, Pune in 2005 and has published a number of papers in National, International conferences and International journal.



Dr. M. Vijayalakshmi is Professor of Information Technology at VES Institute of Technology Mumbai. Currently she is also the Vice Principal of the college. She has more than 25 years of teaching experience both at undergraduate and postgraduate engineering level. Dr. M. Vijayalakshmi holds a Master of Technology and Doctorate Degree in Computer Science and Engineering from the Indian Institute of Technology Mumbai, India. During her career at VESIT, she has served on syllabus board of Mumbai University for BE of Computer Science and Information Technology departments. She has made several contributions to conferences, national and international in the field of Data Mining, Big Data Analytics and has conducted several workshops on data mining related fields. Her areas of research include Databases, Data Mining, Business Intelligence and designing new algorithms for Big Data Analytics.

Contents

Preface	vii
Acknowledgements	xi

Chapter 1 Big Data Analytics 1

Learning Objectives	1
1.1 Introduction to Big Data	1
<i>1.1.1 So What is Big Data?</i>	1
1.2 Big Data Characteristics	2
<i>1.2.1 Volume of Data</i>	2
1.3 Types of Big Data	3
1.4 Traditional Versus Big Data Approach	4
<i>1.4.1 Traditional Data Warehouse Approach</i>	4
<i>1.4.2 Big Data Approach</i>	5
<i>1.4.3 Advantage of “Big Data” Analytics</i>	5
1.5 Technologies Available for Big Data	6
1.6 Case Study of Big Data Solutions	7
<i>1.6.1 Case Study 1</i>	7
<i>1.6.2 Case Study 2</i>	7
Summary	8
Exercises	8

Chapter 2 Hadoop 11

Learning Objectives	11
2.1 Introduction	11

2.2	What is Hadoop?	11
2.2.1	<i>Why Hadoop?</i>	12
2.2.2	<i>Hadoop Goals</i>	12
2.2.3	<i>Hadoop Assumptions</i>	13
2.3	Core Hadoop Components	13
2.3.1	<i>Hadoop Common Package</i>	14
2.3.2	<i>Hadoop Distributed File System (HDFS)</i>	14
2.3.3	<i>MapReduce</i>	16
2.3.4	<i>Yet Another Resource Negotiator (YARN)</i>	18
2.4	Hadoop Ecosystem	18
2.4.1	<i>HBase</i>	19
2.4.2	<i>Hive</i>	19
2.4.3	<i>HCatalog</i>	20
2.4.4	<i>Pig</i>	20
2.4.5	<i>Sqoop</i>	20
2.4.6	<i>Oozie</i>	20
2.4.7	<i>Mahout</i>	20
2.4.8	<i>ZooKeeper</i>	21
2.5	Physical Architecture	21
2.6	Hadoop Limitations	23
2.6.1	<i>Security Concerns</i>	23
2.6.2	<i>Vulnerable By Nature</i>	24
2.6.3	<i>Not Fit for Small Data</i>	24
2.6.4	<i>Potential Stability Issues</i>	24
2.6.5	<i>General Limitations</i>	24
	Summary	24
	Review Questions	25
	Laboratory Exercise	25

Chapter 3	What is NoSQL?	37
------------------	-----------------------	-----------

Learning Objectives	37	
3.1	What is NoSQL?	37
3.1.1	<i>Why NoSQL?</i>	38
3.1.2	<i>CAP Theorem</i>	38

3.2 NoSQL Business Drivers	38
3.2.1 <i>Volume</i>	39
3.2.2 <i>Velocity</i>	39
3.2.3 <i>Variability</i>	40
3.2.4 <i>Agility</i>	40
3.3 NoSQL Case Studies	42
3.3.1 <i>Amazon DynamoDB</i>	42
3.3.2 <i>Google's BigTable</i>	43
3.3.3 <i>MongoDB</i>	44
3.3.4 <i>Neo4j</i>	44
3.4 NoSQL Data Architectural Patterns	45
3.4.1 <i>Types of NoSQL Data Stores</i>	45
3.5 Variations of NoSQL Architectural Patterns	50
3.6 Using NoSQL to Manage Big Data	51
3.6.1 <i>What is a Big Data NoSQL Solution?</i>	51
3.6.2 <i>Understanding Types of Big Data Problems</i>	53
3.6.3 <i>Analyzing Big Data with a Shared Nothing Architecture</i>	54
3.6.4 <i>Choosing Distribution Models</i>	54
3.6.5 <i>Four Ways that NoSQL System Handles Big Data Problems</i>	55
Summary	58
Review Questions	58
Laboratory Exercise	59

Chapter 4 MapReduce 69

Learning Objectives	69
4.1 MapReduce and The New Software Stack	69
4.1.1 <i>Distributed File Systems</i>	70
4.1.2 <i>Physical Organization of Compute Nodes</i>	71
4.2 MapReduce	75
4.2.1 <i>The Map Tasks</i>	76
4.2.2 <i>Grouping by Key</i>	76
4.2.3 <i>The Reduce Tasks</i>	76
4.2.4 <i>Combiners</i>	76
4.2.5 <i>Details of MapReduce Execution</i>	78
4.2.6 <i>Coping with Node Failures</i>	80

4.3 Algorithms Using MapReduce	81
4.3.1 <i>Matrix-Vector Multiplication by MapReduce</i>	82
4.3.2 <i>MapReduce and Relational Operators</i>	83
4.3.3 <i>Computing Selections by MapReduce</i>	83
4.3.4 <i>Computing Projections by MapReduce</i>	84
4.3.5 <i>Union, Intersection and Difference by MapReduce</i>	85
4.3.6 <i>Computing Natural Join by MapReduce</i>	87
4.3.7 <i>Grouping and Aggregation by MapReduce</i>	88
4.3.8 <i>Matrix Multiplication of Large Matrices</i>	89
4.3.9 <i>MapReduce Job Structure</i>	90
Summary	91
Review Questions	92
Laboratory Exercise	92

Chapter 5 Finding Similar Items 105

Learning Objectives	105
5.1 Introduction	105
5.2 Nearest Neighbor Search	106
5.2.1 <i>The NN Search Problem Formulation</i>	107
5.2.2 <i>Jaccard Similarity of Sets</i>	107
5.3 Applications of Nearest Neighbor Search	109
5.4 Similarity of Documents	110
5.4.1 <i>Plagiarism Detection</i>	111
5.4.2 <i>Document Clustering</i>	112
5.4.3 <i>News Aggregators</i>	112
5.5 Collaborative Filtering as a Similar-Sets Problem	112
5.5.1 <i>Online Retail</i>	113
5.6 Recommendation Based on User Ratings	115
5.7 Distance Measures	116
5.7.1 <i>Definition of a Distance Metric</i>	117
5.7.2 <i>Euclidean Distances</i>	118
5.7.3 <i>Jaccard Distance</i>	120
5.7.4 <i>Cosine Distance</i>	120
5.7.5 <i>Edit Distance</i>	122
5.7.6 <i>Hamming Distance</i>	122

Summary	123
Exercises	124
Programming Assignments	125
References	125
Chapter 6 Mining Data Streams	127
Learning Objectives	127
6.1 Introduction	127
6.2 Data Stream Management Systems	128
6.2.1 <i>Data Stream Model</i>	128
6.3 Data Stream Mining	130
6.4 Examples of Data Stream Applications	131
6.4.1 <i>Sensor Networks</i>	131
6.4.2 <i>Network Traffic Analysis</i>	131
6.4.3 <i>Financial Applications</i>	132
6.4.4 <i>Transaction Log Analysis</i>	132
6.5 Stream Queries	132
6.6 Issues in Data Stream Query Processing	133
6.6.1 <i>Unbounded Memory Requirements</i>	133
6.6.2 <i>Approximate Query Answering</i>	134
6.6.3 <i>Sliding Windows</i>	134
6.6.4 <i>Batch Processing, Sampling and Synopses</i>	135
6.6.5 <i>Blocking Operators</i>	135
6.7 Sampling in Data Streams	136
6.7.1 <i>Reservoir Sampling</i>	136
6.7.2 <i>Biased Reservoir Sampling</i>	137
6.7.3 <i>Concise Sampling</i>	137
6.8 Filtering Streams	138
6.8.1 <i>An Example</i>	139
6.8.2 <i>The Bloom Filter</i>	140
6.8.3 <i>Analysis of the Bloom Filter</i>	141
6.9 Counting Distinct Elements in a Stream	143
6.9.1 <i>Count Distinct Problem</i>	143
6.9.2 <i>The Flajolet–Martin Algorithm</i>	143
6.9.3 <i>Variations to the FM Algorithm</i>	145
6.9.4 <i>Space Requirements</i>	146

6.10	Querying on Windows – Counting Ones in a Window	146
6.10.1	<i>Cost of Exact Counting</i>	147
6.10.2	<i>The Datar–Gionis–Indyk–Motwani Algorithm</i>	147
6.10.3	<i>Query Answering in DGIM Algorithm</i>	149
6.10.4	<i>Updating Windows in DGIM Algorithm</i>	151
6.11	Decaying Windows	152
6.11.1	<i>The Problem of Most-Common Elements</i>	152
6.11.2	<i>Describing a Decaying Window</i>	153
	Summary	155
	Exercises	156
	Programming Assignments	157
	References	158

Chapter 7	Link Analysis	159
------------------	----------------------	------------

	Learning Objectives	159
7.1	Introduction	159
7.2	History of Search Engines and Spam	160
7.3	PageRank	162
7.3.1	<i>PageRank Definition</i>	163
7.3.2	<i>PageRank Computation</i>	164
7.3.3	<i>Structure of the Web</i>	167
7.3.4	<i>Modified PageRank</i>	169
7.3.5	<i>Using PageRank in a Search Engine</i>	172
7.4	Efficient Computation of PageRank	173
7.4.1	<i>Efficient Representation of Transition Matrices</i>	173
7.4.2	<i>PageRank Implementation Using Map Reduce</i>	174
7.4.3	<i>Use of Combiners to Consolidate the Result Vector</i>	176
7.5	Topic-Sensitive PageRank	176
7.5.1	<i>Motivation for Topic-Sensitive PageRank</i>	177
7.5.2	<i>Implementing Topic-Sensitive PageRank</i>	178
7.5.3	<i>Using Topic-Sensitive PageRank in a Search Engine</i>	178
7.6	Link Spam	179
7.6.1	<i>Spam Farm</i>	180
7.6.2	<i>Link Spam Combating Techniques</i>	182

7.7 Hubs and Authorities	183
7.7.1 <i>Hyperlink-Induced Topic Search Concept</i>	184
7.7.2 <i>Hyperlink-Induced Topic Search Algorithm</i>	185
Summary	189
Exercises	191
Programming Assignments	192
References	192

Chapter 8 Frequent Itemset Mining	195
--	------------

Learning Objectives	195
8.1 Introduction	195
8.2 Market-Basket Model	196
8.2.1 <i>Frequent-Itemset Mining</i>	196
8.2.2 <i>Applications</i>	197
8.2.3 <i>Association Rule Mining</i>	199
8.3 Algorithm for Finding Frequent Itemsets	204
8.3.1 <i>Framework for Frequent-Itemset Mining</i>	204
8.3.2 <i>Itemset Counting using Main Memory</i>	206
8.3.3 <i>Approaches for Main Memory Counting</i>	208
8.3.4 <i>Monotonicity Property of Itemsets</i>	210
8.3.5 <i>The Apriori Algorithm</i>	211
8.4 Handling Larger Datasets in Main Memory	215
8.4.1 <i>Algorithm of Park–Chen–Yu</i>	216
8.4.2 <i>The Multistage Algorithm</i>	221
8.4.3 <i>The Multihash Algorithm</i>	223
8.5 Limited Pass Algorithms	224
8.5.1 <i>The Randomized Sampling Algorithm</i>	224
8.5.2 <i>The Algorithm of Savasere, Omiecinski and Navathe</i>	226
8.5.3 <i>The SON Algorithm and MapReduce</i>	228
8.5.4 <i>Toivonen’s Algorithm</i>	229
8.6 Counting Frequent Items in a Stream	231
8.6.1 <i>Sampling Methods for Streams</i>	232
8.6.2 <i>Frequent Itemsets in Decaying Windows</i>	233
Summary	234

Exercises	236
Programming Assignments	238
References	238

Chapter 9 Clustering Approaches **239**

Learning Objectives	239
9.1 Introduction	239
9.2 Overview of Clustering Techniques	240
9.2.1 <i>Basic Concepts</i>	240
9.2.2 <i>Clustering Applications</i>	242
9.2.3 <i>Clustering Strategies</i>	243
9.2.4 <i>Curse of Dimensionality</i>	244
9.3 Hierarchical Clustering	245
9.3.1 <i>Hierarchical Clustering in Euclidean Space</i>	245
9.3.2 <i>Hierarchical Clustering in Non-Euclidean Space</i>	248
9.4 Partitioning Methods	248
9.4.1 <i>K-Means Algorithm</i>	249
9.4.2 <i>K-Means For Large-Scale Data</i>	251
9.5 The CURE Algorithm	254
9.5.1 <i>Overview of the Algorithm</i>	255
9.5.2 <i>CURE Implementation</i>	255
9.6 Clustering Streams	258
9.6.1 <i>A Simple Streaming Model</i>	259
9.6.2 <i>A Stream Clustering Algorithm</i>	259
9.6.3 <i>Answering Queries</i>	261
Summary	261
Exercises	262
Programming Assignments	264
References	264

Chapter 10 Recommendation Systems **265**

Learning Objectives	265
10.1 Introduction	265
10.1.1 <i>What is the Use of Recommender System?</i>	265

10.1.2 <i>Where Do We See Recommendations?</i>	265
10.1.3 <i>What Does Recommender System Do?</i>	265
10.2 A Model for Recommendation Systems	266
10.3 Collaborative-Filtering System	266
10.3.1 <i>Nearest-Neighbor Technique</i>	266
10.3.2 <i>Collaborative Filtering Example</i>	267
10.3.3 <i>Methods for Improving Prediction Function</i>	268
10.4 Content-Based Recommendations	269
10.4.1 <i>Discovering Features of Documents</i>	269
10.4.2 <i>Standard Measure: Term Frequency–Inverse Document Frequency</i>	271
10.4.3 <i>Obtaining Item Features from Tags</i>	272
10.4.4 <i>User Profiles</i>	273
10.4.5 <i>Classification Algorithms</i>	274
Summary	275
Review Questions	276
Laboratory Exercise	276

Chapter 11 Mining Social Network Graphs **279**

Learning Objectives	279
11.1 Introduction	279
11.2 Applications of Social Network Mining	280
11.3 Social Networks as a Graph	280
11.3.1 <i>Social Networks</i>	281
11.3.2 <i>Social Network Graphs</i>	281
11.4 Types of Social Networks	284
11.5 Clustering of Social Graphs	285
11.5.1 <i>Applying Standard Clustering Techniques</i>	286
11.5.2 <i>Introducing “Betweenness” Measure for Graph Clustering</i>	288
11.5.3 <i>Girvan–Newman Algorithm</i>	289
11.6 Direct Discovery of Communities in a Social Graph	290
11.6.1 <i>Clique Percolation Method (CPM)</i>	291
11.6.2 <i>Other Community Detection Techniques</i>	293
11.7 SimRank	294
11.7.1 <i>SimRank Implementation</i>	294

11.8 Counting Triangles in a Social Graph	295
11.8.1 <i>Why Should We Count Triangles?</i>	296
11.8.2 <i>Triangle Counting Algorithms</i>	296
11.8.3 <i>Counting Triangles Using MapReduce</i>	297
Summary	298
Exercises	299
Programming Assignments	300
References	300
Appendix	301
Index	303

1

Big Data Analytics

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Analyze the growing level of data.
- Demonstrate the characteristics and types of big data.
- Learn why traditional data storage cannot be used to store big data.
- Learn the availability of modern technologies developed to handle big data.
- Learn Big Data solution application for different case studies.

1.1 Introduction to Big Data

Big data is a relative term. If big data is referred by “volume” of transactions and transaction history, then hundreds of terabytes (10^{12} bytes) may be considered “big data” for a pharmaceutical company and volume of transactions in petabytes (10^{15} bytes) may be considered small for a government agency.

1.1.1 So What is Big Data?

Big data refers to the massive datasets that are collected from a variety of data sources for business needs to reveal new insights for optimized decision making.

According to IBM sources, e-business and consumer life create 2.5 exabytes (10^{18} bytes) of data per day. It is predicted that 8 zeta bytes (10^{21} bytes) of data will be produced by 2015 and 90% of these will be from the last 5 years. These data have to be stored for analysis to reveal hidden correlations and patterns which are termed as Big Data Analytics.

Suppose personal computers (PCs) can hold 500 GB of data; it would require 20 billion PCs to store zeta bytes of data. Google stores its data in millions of servers around the world. Everyday around 10 million text messages are sent; Facebook has millions of active accounts and friends share content, photos and videos. These data are mostly related to human behavior and interactions and this helps the Information Technology (IT) analysts to store and analyze data to extract useful information that drives the business.

Big Data Analytics as shown in Fig. 1.1 is the result of three major trends in computing: *Mobile Computing* using hand-held devices, such as smartphone and tablets; *Social Networking*, such as Facebook and Pinterest; and *Cloud Computing* by which one can rent or lease the hardware setup for storing and computing.

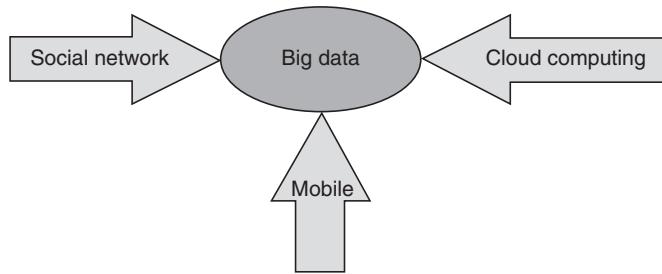


Figure 1.1 Big Data: Result of three computing trends.

1.2 Big Data Characteristics

1.2.1 Volume of Data

Normally transactional or operational data in any ERP kind of applications is in megabytes, customer-related data stored in CRM (Customer Relationship Management) applications are in gigabytes, all web-related data is in terabytes and real-time data, such as clickstream, sensor, mobile applications are in petabytes as shown in Fig. 1.2.

As we have seen, a huge volume of information is generated through social media, and the velocity in which the information gets uploaded is also high. This information can be in the form of pictures, videos and unstructured texts via social media. The statements above articulate three “Vs”: volume, velocity and variety (format) that add complexity to the analysis of information and data management.

Volume as the mathematical notation goes is $\text{length} \times \text{breadth} \times \text{depth}$. Since data is real time, we can interpret it as “How long you need to collect the data?” and so the interpretation of length, breadth and depth is as follows:

1. *The length is the time dimension.*

What are the different sources or from where the data has to be collected and how should the data be integrated?

2. *Discovery and integration form the breadth.*

To what depth you need to understand and analyze data?

3. *Analysis and insights form the depth.*

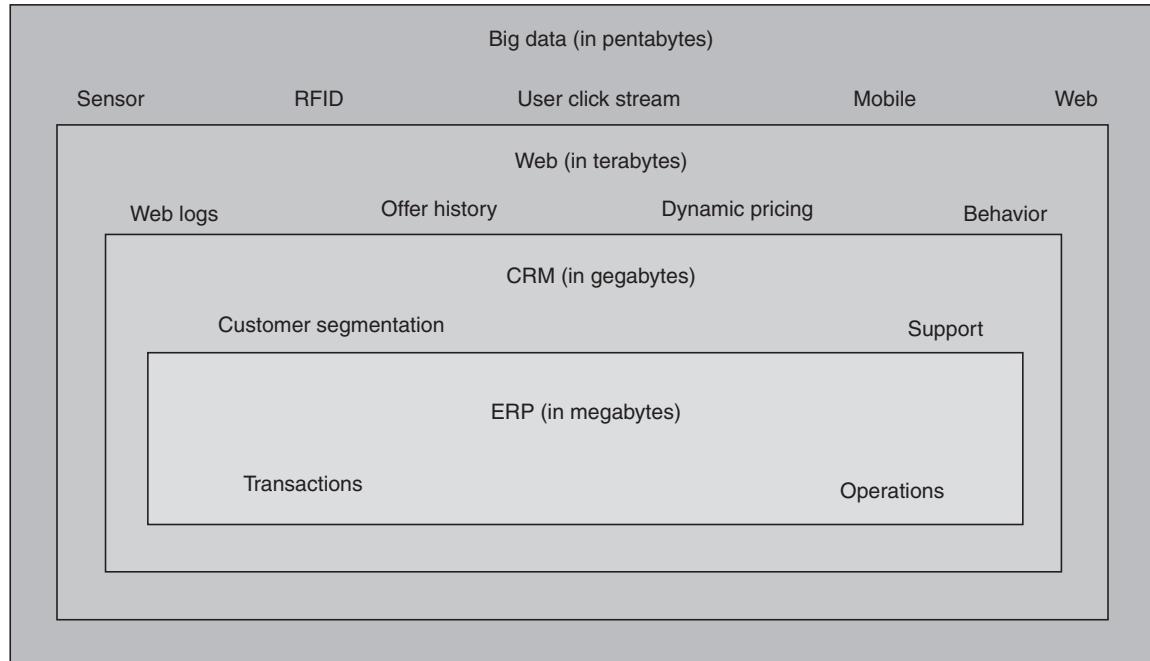


Figure 1.2 Volume of data.

The data that comes from a wide variety of sources is random and unstructured. The increasing volume of data outstrips traditional data store. Hence data loading in warehouse will become slow. It is difficult to control the data flow because genuineness of the data (veracity) has to be verified before loading the data. Velocity is taken into account for time-limited process too, since it streams data from real world into the organization to maximize the value of information. In all, big data has five major components: *Volume, Velocity, Variety, Veracity and Value*.

1.3 Types of Big Data

Traditionally in database terminology, datasets are rows of the table corresponding to the transactions; they are also called the operational data. These are structured and have the known data-types such as numeric, floating point, character, string, etc. Transactions and its history are extracted, transformed and loaded (ETL process) into the data warehouse. Using data mining algorithms, the hidden information or pattern is derived and used for business predictions.

Couple of decades back, a lot of merger and acquisition happened in the industry. Merged companies had huge transactional data with different structure. Migrating data from one company's database to another was a huge issue. At that time, IBM and W3C consortiums came up with open standards

such as XML and RDF data models. XML data is semi-structured. Semi-structured data does not have fixed fields but contains tags to separate data elements. RDF is a graph data. Apart from these, there is “streaming” data which is an ordered sequence of instances that are scanned only once using limited computing and storage capabilities. Examples of data streams include phone conversations, ATM transactions, computer network traffic, web searches, and sensor data.

But what if the data is not available in the known data type and has a variety of other formats which the business needs for its decision making? So apart from the data available from the transactions, there are information present in the form of emails, audio and video images, logs, blogs and forums, social networking sites, clickstreams, sensors, statistical data centers and mobile phone applications, which are required for making business decisions in organizations. The arriving speed of data in data streams is really fast and it is not feasible to store such huge amount of data and so “real-time” processing is done. Columnar databases are used for this purpose. Therefore, data generated can be classified as real time, event, structured, semi-structured, unstructured, complex, etc.

1.4 Traditional Versus Big Data Approach

Traditional data management and analytics store structured data in data marts and data warehouses. Traditional data management, too, was able to handle huge volume of transactions but up to an extent; for example, billions of credit-card transactions worldwide could be handled but not peta or zeta bytes of data and that too in variety of formats.

Most important problem of the business organization is to understand the true customer experience. Business organizations have multiple customer inputs, including transactional systems, customer service (call centers), web sites, online chat services, retail stores and partner services. Customers may use one or many of these systems. What is necessary is to find the overall customer experience or to understand the combined effects of all these systems.

1.4.1 Traditional Data Warehouse Approach

Hundreds of these systems are distributed throughout the organization and its partners. Each of these systems has its own silos of data and many of these silos may contain information about customer experience that is required for making business decisions. The traditional data warehouse system approach has extensive data definition with each of these systems and vast transfer of data from each other. Many of the data sources do not use the same definitions. Copying all the data from each of these systems to a centralized location and keeping it updated is not an easy task. Moreover, sampling the data will not serve the purpose of extracting required information. The objective of big data is to construct a customer experience view over a period of time from all the events that took place. The duration for implementing such a project will be at least 1 year with traditional systems.

1.4.2 Big Data Approach

The alternative to this problem is the big data approach. Many IT tools are available for big data projects. The storage requirements of big data are taken care of by Hadoop cluster. Apache Spark is capable of stream processing (e.g., advertisement data). When used, these tools can dramatically reduce the time-to-value – in most cases from more than 2 years to less than 4 months. The benefit is that many speculative projects can be approved or abandoned based on the result.

Organizations whose data workloads are constant and predictable are better served by the traditional database, whereas organizations challenged by increasing data demands will want to take advantage of the Hadoop's scalable infrastructure. Scalability allows servers to be added on demand to accommodate the growing workloads.

There are hybrid systems, which integrate Hadoop platforms with traditional (relational) databases, that are gaining popularity as the cost-effective systems for organizations to leverage the benefits of both the platforms.

1.4.3 Advantage of "Big Data" Analytics

Big Data Analytics is advantageous in the following two ways when compared to the traditional analytical model:

1. Big Data Analytics uses a simple model that can be applied to volumes of data that would be too large for the traditional analytical environment. Researchers say that a simple algorithm with a large volume of data produces more accurate results than a sophisticated algorithm with a small volume of data. The algorithm by itself is not too great but its ability to apply it to huge amounts of data without compromising on performance gives the competitive edge.
2. Big Data Analytics has sophisticated model developed for it. Present day's database management system (DBMS) vendors provide analysis algorithms directly. Most companies go well beyond this and develop newer, more sophisticated statistical analysis models.

Advantages of using Hadoop over traditional storage systems such as relational DBMS (RDBMS) are as follows:

1. Scalability – nodes can be added to scale the system with little administration.
2. Unlike traditional RDBMS, no pre-processing is required before storing.
3. Any unstructured data such as text, images and videos can be stored.
4. There is no limit to how much data needs to be stored and for how long.
5. Protection against hardware failure – in case of any node failure, it is redirected to other nodes. Multiple copies of the data are automatically stored.

For analyzing big data, it is important to ascertain the following areas while acquiring systems:

1. Support to extract data from different data sources and different internal and external platforms.
2. Support different data types, for example, text documents, existing databases, data streams, image, voice and video.
3. Data integrator to combine the information from different sources to do analysis.
4. A simple interface to carry out analysis.
5. Facility to view results in different ways as per user's needs.

1.5 Technologies Available for Big Data

Google was processing 100 TB data per day in 2004 and 20 PB of day per day in 2008 using MapReduce. Facebook too had similar statistics of data; it generated 2.5 PB of user data, with every day upload of about 15 TB.

Concepts such as Hadoop and NoSQL give us a new alternative to address the big data. With an advent of these technologies, it has become economically feasible to store history data. Raw data can be stored. There is no need to model data and store it in data warehouse in order to analyze. There is no need to make decisions about how much of history data to be stored.

Huge volume of data is fragmented and processing is done in parallel in many servers. Using a framework, these fragmented files containing data are mapped. An open-source software framework Hadoop is used for storing and processing big data across large clusters of servers. MapReduce is a programming paradigm that assists massive scalability across thousands of servers in a Hadoop cluster. Hadoop distributed file system (HDFS) manages the storage and retrieval of data and metadata required for computation. Hadoop database (HBase) is a non-relational database. Real-time data is stored in column-oriented tables in HBase. It is a backend system for MapReduce jobs output. Sqoop is a tool used for data transfer from Hadoop to relational databases and data warehouses. Data can also be imported from relational databases to HDFS (or related systems such as HBase or Hive).

Hive is a data warehouse platform built on the top of Hadoop. It supports querying and managing large datasets across distributed storage. Hive leverages a SQL-like language called HiveQL. When it is inconvenient to express their logic in HiveQL, it allows MapReduce programmers to plug in custom mappers and reducers. Huge data from transactional databases cannot use Hadoop, since it is not real time. NoSQL databases can be used if transactional systems do not require ACID (Atomicity, Consistency, Isolation and Durability) properties.

Mahout is a data-mining library. It is used to implement the popular data mining algorithms such as clustering, classification, frequent pattern mining and collective filtering using MapReduce model.

1.6 Case Study of Big Data Solutions

We say “Time is money” but so is data. Digital information gives organizations insight to customer needs and, thereby, leads to innovation and productivity. It helps them in identifying new opportunities so as to sustain in the market. These cutting-edge technologies are used to analyze the customer-related datasets in timely fashion for an effective decision making. A few of the domain that can benefit from Big Data Analytics are mentioned below:

1. Insurance companies can understand the likelihood of fraud by accessing the internal and external data while processing the claims. This will help them to speed up the handling of the simple claims and analyze the complex or fraudulent ones.
2. Manufacturers and distributors can be benefitted by realizing supply chain issues earlier so that they can take decisions on different logistical approaches to avoid the additional costs associated with material delays, overstock or stock-out conditions.
3. Firms such as hotels, telecommunications companies, retailers and restaurants that serve customers likely to have better clarity on customer needs to build a strong customer base and loyalty.
4. Public services such as traffic, ambulance, transportations, etc. can optimize their delivery mechanisms by measuring the usage of these essential services.
5. Smart-city is the buzz word today. The idea is make cities more efficient and sustainable to improve the lives of the citizens. Data related to censors, crime, emergency services, real-estate, energy, financial transactions, call details, astronomy, data.gov, customs.gov and scientific data are all used for analysis to do the improvement.

1.6.1 Case Study 1

What problem does Hadoop solve? Businesses and governments have a large amount of data that needs to be analyzed and processed very quickly. If this data is fragmented into smaller chunks and spread over many machines, all those machines process their portion of the data in parallel and the results are obtained extremely fast.

For example, a huge data file containing feedback mails is sent to the customer service department. The objective is to find the number of times goods were returned and refund requested. This will help the business to find the performance of the vendor or supplier.

It is a simple word count exercise. The client will load the data into the cluster (Feedback.txt), submit a job describing how to analyze that data (word count), the cluster will store the results in a new file (Returned.txt), and the client will read the results file.

1.6.2 Case Study 2

1.6.2.1 Clickstream Analytics

Everyday millions of people visit organizations' website and this forms the face of the organization to the public. The website informs the public about various products and services that are available with

them. Some organizations allow people to transact through their website. Clickstream data is generated when the customers or visitors interact through the website. These data include the pages they load, the time spent by them on each page, the links they clicked, the frequency of their visit, from which page do they exit, etc. This gives good amount of information about their customers and helps in understanding them better. Eventually website content, its organization, navigation and transaction completion can be improved. This ensures that the customers can easily find what they want. The website owner can understand the customers' buying behavior.

Researchers are working on the mathematical model to predict the customer loyalty. This can help the sales and marketing team for preparing their campaigning strategies. For e-commerce sites, these amount to a few billion clicks per month. For analysis and prediction, more than 3-years data is collected taking into account seasonality, trend, etc. Analysts use these kind of data for finding correlation. This is an extremely complex calculation since the data is not collected from static environment of an application; instead they are from the collection of different services. Pages are generated dynamically depending on the customer choice or input data. If data is stored in the traditional database and a query is performed, then it would take days of processing to bring out the result. By using massively parallel processing system, they can be made 100 times faster, reducing the time to hours or minutes. In turn, this helps the organizations to address the issues very fast. Hadoop can load both structured and unstructured data efficiently. This also helps in performing repetitive queries by analysts.

Earlier data mining algorithms were used to categorize customers based on their profile and guide them through their orders. Most of the time it so happens that the customers may not fit into these categories. With Big Data Analytics, the scenario has changed; an interactive and customized experience is given to the customers.

Summary

- The basic idea is that everything that we do when we use the web is leaving a digital trace (Data) that grows everyday (Big Data). We need to analyze this to get information.
- The problems in handling Big Data by traditional database systems are addressed here and the need for different technologies to analyze big data is understood.
- An overview of the technologies that support in handling and analyzing Big Data is seen in this chapter.
- Few case studies that require Big Data solution are also discussed.

Exercises

1. A bookmarking site permits to bookmark, review, rate and search various links on any topic. Analyze social bookmarking sites like

stumbleupon.com or reddit.com to find insights.

2. A Consumer Complaints site like complaintsboard.com, consumercourtforum.in allows to register complaints. Find the attributes available for analytics. Write a few queries.
3. *Tourism Data Analysis:* Analyze the tourism data and identify the ones which are difficult in storing in RDBMS and the need for Big Data technologies for storing and analyzing them.

2

Hadoop

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Understand the need for Hadoop.
- Learn the assumptions and goals of Hadoop.
- Learn about the Hadoop components.
- Understand and learn Hadoop Distributed File System.
- Learn how HDFS handles job processing.
- Learn about the MapReduce components.
- Learn the use of Hadoop ecosystem.
- Understand the limitation of Hadoop.

2.1 Introduction

In the late 1990s, search engines and indexes were created for helping people to find relevant information about the content searched. Open-source web search engine was invented to return results faster by distributing the data across different machines to process the tasks simultaneously. Web search engine used web crawlers to copy all the web pages visited; later the search engine processed and indexed the downloaded pages. Nutch was one such engine developed by Doug Cutting and Mike Cafarella. During the same period, Google also worked on the same concept – storing and processing data in a distributed environment so that more relevant search results could be fetched faster in an automated way.

Doug Cutting joined Yahoo in 2006 and he retained the name Nutch (a word for “meal” that Doug’s son used as a toddler) for the web crawler portion of the project. He named the storage and distributed processing portion of the project as Hadoop (Hadoop was the name of Doug’s son’s toy elephant). In 2008, Yahoo released Hadoop as an open-source project. Today, Hadoop is developed as a framework by a non-profit organization Apache Software Foundation (ASF), a global community of software developers.

2.2 What is Hadoop?

Apache Hadoop is a framework that allows distributed processing of large datasets across clusters of commodity computers using a simple programming model. It is designed to scale-up from single servers to thousands of machines, each providing computation and storage. Rather than rely on hardware to deliver high-availability, the framework itself is designed to detect and handle failures at the application

layer, thus delivering a highly available service on top of a cluster of computers, each of which may be prone to failures.

In short, Hadoop is an open-source software framework for storing and processing big data in a distributed way on large clusters of commodity hardware. Basically, it accomplishes the following two tasks:

1. Massive data storage.
2. Faster processing.

2.2.1 Why Hadoop?

Problems in data transfer made the organizations to think about an alternate way.

Example 1

1. The transfer speed is around 100 MB/s and a standard disk is 1 TB.
2. Time to read entire disk = 10,000 s or 3 h!
3. Increase in processing time may not be very helpful because of two reasons:
 - Network bandwidth is now more of a limiting factor.
 - Physical limits of processor chips are reached.

Example 2

If 100 TB of datasets are to be scanned on a 1000 node cluster, then in case of

1. remote storage with 10 Mbps bandwidth, it would take 165 min.
2. local storage with 50 Mbps, it will take 33 min.

So it is better to move computation rather than moving data.

Taking care of hardware failure cannot be made optional in Big Data Analytics but has to be made as a rule. In case of 1000 nodes, we need to consider say 4000 disks, 8000 core, 25 switches, 1000 NICs and 2000 RAMs (16 TB). Meantime between failures could be even less than a day since commodity hardware is used. There is a need for fault tolerant store to guarantee reasonable availability.

2.2.2 Hadoop Goals

The main goals of Hadoop are listed below:

1. **Scalable:** It can scale up from a single server to thousands of servers.
2. **Fault tolerance:** It is designed with very high degree of fault tolerance.

3. **Economical:** It uses commodity hardware instead of high-end hardware.
4. **Handle hardware failures:** The resiliency of these clusters comes from the software's ability to detect and handle failures at the application layer.

The Hadoop framework can store huge amounts of data by dividing the data into blocks and storing it across multiple computers, and computations can be run in parallel across multiple connected machines.

Hadoop gained its importance because of its ability to process huge amount of variety of data generated every day especially from automated sensors and social media using low-cost commodity hardware.

Since processing is done in batches, throughput is high but latency is low. Latency is the time (minutes/seconds or clock period) to perform some action or produce some result whereas throughput is the number of such actions executed or result produced per unit of time. The throughput of memory system is termed as memory bandwidth.

2.2.3 Hadoop Assumptions

Hadoop was developed with large clusters of computers in mind with the following assumptions:

1. Hardware will fail, since it considers a large cluster of computers.
2. Processing will be run in batches; so aims at high throughput as opposed to low latency.
3. Applications that run on Hadoop Distributed File System (HDFS) have large datasets typically from gigabytes to terabytes in size.
4. Portability is important.
5. Availability of high-aggregate data bandwidth and scale to hundreds of nodes in a single cluster.
6. Should support tens of millions of files in a single instance.
7. Applications need a write-once-read-many access model.

2.3

Core Hadoop Components

Hadoop consists of the following components:

1. **Hadoop Common:** This package provides file system and OS level abstractions. It contains libraries and utilities required by other Hadoop modules.
2. **Hadoop Distributed File System (HDFS):** HDFS is a distributed file system that provides a limited interface for managing the file system.
3. **Hadoop MapReduce:** MapReduce is the key algorithm that the Hadoop MapReduce engine uses to distribute work around a cluster.

4. **Hadoop Yet Another Resource Negotiator (YARN) (MapReduce 2.0):** It is a resource-management platform responsible for managing compute resources in clusters and using them for scheduling of users' applications.

2.3.1 Hadoop Common Package

This consists of necessary Java archive (JAR) files and scripts needed to start Hadoop. Hadoop requires Java Runtime Environment (JRE) 1.6 or higher version. The standard start-up and shut-down scripts need Secure Shell (SSH) to be setup between the nodes in the cluster.

HDFS (storage) and MapReduce (processing) are the two core components of Apache Hadoop. Both HDFS and MapReduce work in unison and they are co-deployed, such that there is a single cluster that provides the ability to move computation to the data. Thus, the storage system HDFS is not physically separate from a processing system MapReduce.

2.3.2 Hadoop Distributed File System (HDFS)

HDFS is a distributed file system that provides a limited interface for managing the file system to allow it to scale and provide high throughput. HDFS creates multiple replicas of each data block and distributes them on computers throughout a cluster to enable reliable and rapid access. When a file is loaded into HDFS, it is replicated and fragmented into “blocks” of data, which are stored across the cluster nodes; the cluster nodes are also called the DataNodes. The NameNode is responsible for storage and management of metadata, so that when MapReduce or another execution framework calls for the data, the NameNode informs it where the data that is needed resides. Figure 2.1 shows the NameNode and DataNode block replication in HDFS architecture.

1. HDFS creates multiple replicas of data blocks for reliability, placing them on the computer nodes around the cluster.
2. Hadoop's target is to run on clusters of the order of 10,000 nodes.
3. A file consists of many 64 MB blocks.

2.3.2.1 Main Components of HDFS

2.3.2.1.1 NameNode

NameNode is the master that contains the metadata. In general, it maintains the directories and files and manages the blocks which are present on the DataNode. The following are the functions of NameNode:

1. Manages namespace of the file system in memory.
2. Maintains “inode” information.
3. Maps *inode* to the list of blocks and locations.

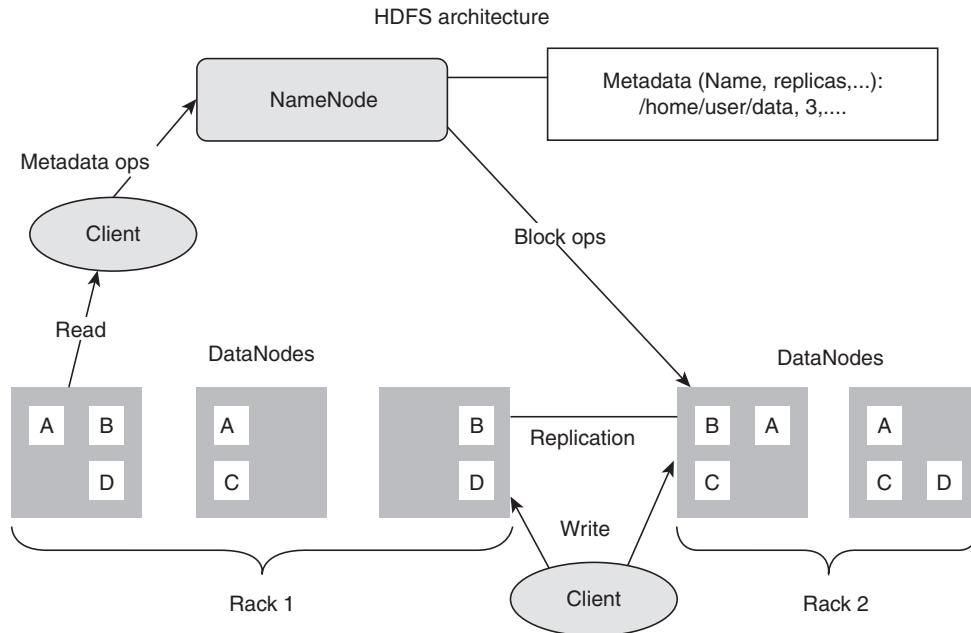


Figure 2.1 NameNode and DataNode block replication.

4. Takes care of authorization and authentication.
5. Creates checkpoints and logs the namespace changes.

So the NameNode maps DataNode to the list of blocks, monitors status (health) of DataNode and replicates the missing blocks.

2.3.2.1.2 DataNodes

DataNodes are the slaves which provide the actual storage and are deployed on each machine. They are responsible for processing read and write requests for the clients. The following are the other functions of DataNode:

1. Handles block storage on multiple volumes and also maintain block integrity.
2. Periodically sends heartbeats and also the block reports to NameNode.

Figure 2.2 shows how HDFS handles job processing requests from the user in the form of Sequence Diagram. User copies the input files into DFS and submits the job to the client. Client gets the input file information from DFS, creates splits and uploads the job information to DFS. JobTracker puts ready job into the internal queue. JobScheduler picks job from the queue and initializes the job by creating job object. JobTracker creates a list of tasks and assigns one map task for each input split.

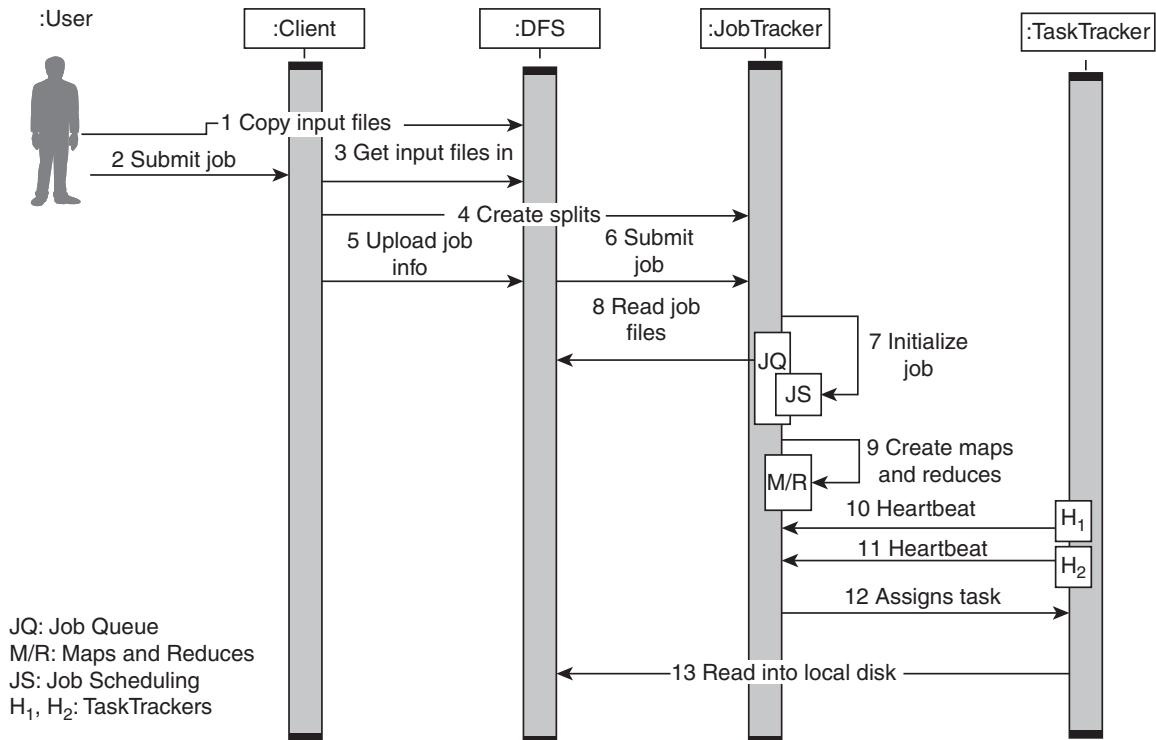


Figure 2.2 Sequence diagram depicting job processing requests from the user.

TaskTrackers send heartbeat to JobTracker to indicate if ready to run new tasks. JobTracker chooses task from first job in priority-queue and assigns it to the TaskTracker.

Secondary NameNode is responsible for performing periodic checkpoints. These are used to restart the NameNode in case of failure. MapReduce can then process the data where it is located.

2.3.3 MapReduce

The MapReduce algorithm aids in parallel processing and basically comprises two sequential phases: map and reduce.

1. In the map phase, a set of key–value pairs forms the input and over each key–value pair, the desired function is executed so as to generate a set of intermediate key–value pairs.
2. In the reduce phase, the intermediate key–value pairs are grouped by key and the values are combined together according to the reduce algorithm provided by the user. Sometimes no reduce phase is required, given the type of operation coded by the user.

MapReduce processes are divided between two applications, JobTracker and TaskTracker at the cluster level. JobTracker is responsible for scheduling job runs and managing computational resources across the cluster; hence it runs on only one node of the cluster. Each MapReduce job is split into a number of tasks which are assigned to the various TaskTrackers depending on which data is stored on that node. So TaskTracker runs on every slave node in the cluster. JobTracker oversees the progress of each TaskTracker as they complete their individual tasks.

The following points summarize the above discussion:

1. Hadoop implements Google's MapReduce, using HDFS.
2. MapReduce divides applications into many small blocks of work.
3. Performs Sort/merge-based distributed computing.
4. Follows functional style programming and so naturally is parallelizable across a large cluster of workstations or PCs.

In the MapReduce paradigm, each job has a user-defined map phase followed by a user-defined reduce phase as follows:

1. Map phase is a parallel, share-nothing processing of input.
2. In the reduce phase, the output of the map phase is aggregated.

HDFS is the storage system for both input and output of the MapReduce jobs.

2.3.3.1 Main Components of MapReduce

The main components of MapReduce are listed below:

1. **JobTrackers:** JobTracker is the master which manages the jobs and resources in the cluster. The JobTracker tries to schedule each map on the TaskTracker which is running on the same DataNode as the underlying block.
2. **TaskTrackers:** TaskTrackers are slaves which are deployed on each machine in the cluster. They are responsible for running the map and reduce tasks as instructed by the JobTracker.
3. **JobHistoryServer:** JobHistoryServer is a daemon that saves historical information about completed tasks/applications.

Note: If the map phase has M fragments and the reduce phase has R fragments, then M and R should be much larger than the number of worker machines. R is often decided by the users, because the output of each reduce task ends up in a separate output file. Typically (at Google), $M = 2,00,000$ and $R = 5000$, using 2000 worker machines.

2.3.4 Yet Another Resource Negotiator (YARN)

YARN addresses problems with MapReduce 1.0s architecture, specifically the one faced by JobTracker service.

Hadoop generally has up to tens of thousands of nodes in the cluster. Obviously, MapReduce 1.0 had issues with scalability, memory usage, synchronization, and also Single Point of Failure (SPOF) issues. In effect, YARN became another core component of Apache Hadoop.

2.3.4.1 What does YARN do?

It splits up the two major functionalities “resource management” and “job scheduling and monitoring” of the JobTracker into two separate daemons. One acts as a “global Resource Manager (RM)” and the other as a “ApplicationMaster (AM)” per application. Thus, instead of having a single node to handle both scheduling and resource management for the entire cluster, YARN distributes this responsibility across the cluster.

The RM and the NodeManager manage the applications in a distributed manner. The RM is the one that arbitrates resources among all the applications in the system. The per-application AM negotiates resources from the RM and works with the NodeManager(s) to execute and monitor the component tasks.

1. The RM has a scheduler that takes into account constraints such as queue capacities, user-limits, etc. before allocating resources to the various running applications.
2. The scheduler performs its scheduling function based on the resource requirements of the applications.
3. The NodeManager is responsible for launching the applications’ containers. It monitors the application’s resource usage (CPU, memory, disk, network) and reports the information to the RM.
4. Each AM runs as a normal container. It has the responsibility of negotiating appropriate resource containers from the scheduler, tracking their status and monitoring their progress.

2.4 Hadoop Ecosystem

Apart from HDFS and MapReduce, the other components of Hadoop ecosystem are shown in Fig. 2.3. The main ecosystems components of Hadoop architecture are as follows:

1. **Apache HBase:** Columnar (Non-relational) database.
2. **Apache Hive:** Data access and query.

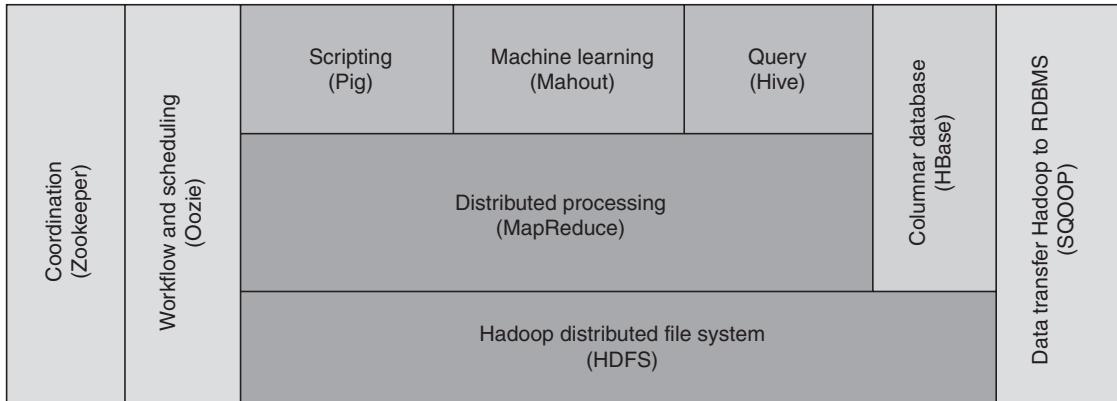


Figure 2.3 Hadoop ecosystem.

3. **Apache HCatalog:** Metadata services.
4. **Apache Pig:** Scripting platform.
5. **Apache Mahout:** Machine learning libraries for Data Mining.
6. **Apache Oozie:** Workflow and scheduling services.
7. **Apache ZooKeeper:** Cluster coordination.
8. **Apache Sqoop:** Data integration services.

These components are discussed in detail in the following subsections.

2.4.1 HBase

HBase “is an open-source, distributed, versioned, column-oriented store” that sits on top of HDFS. HBase is based on Google’s Bigtable. HBase is based on columns rather than rows. This essentially increases the speed of execution of operations if they are need to be performed on similar values across massive datasets; for example, read/write operations that involve all rows but only a small subset of all columns. HBase does not provide its own query or scripting language, but is accessible through Java, Thrift and REST APIs.

2.4.2 Hive

Hive provides a warehouse structure for other Hadoop input sources and SQL-like access for data in HDFS. Hive’s query language, HiveQL, compiles to MapReduce and also allows user-defined functions (UDFs). Hive’s data model is based primarily on three related data structures: tables, partitions and buckets. Tables correspond to HDFS directories that are divided into partitions, which in turn can be divided into buckets.

2.4.3 HCatalog

HCatalog is a metadata and table storage management service for HDFS. HCatalog's goal is to simplify the user's interaction with HDFS data and enable data sharing between tools and execution platforms.

2.4.4 Pig

Pig is a run-time environment that allows users to execute MapReduce on a Hadoop cluster. Pig Latin is a high-level scripting language on Pig platform. Like HiveQL in Hive, Pig Latin is a higher-level language that compiles to MapReduce.

Pig is more flexible with respect to possible data format than Hive due to its data model. Pig's data model is similar to the relational data model, but here tuples can be nested. For example, a table of tuples can have a table in the third field of each tuple. In Pig, tables are called bags. Pig also has a "map" data type, which is useful in representing semi-structured data such as JSON or XML."

2.4.5 Sqoop

Sqoop ("SQL-to-Hadoop") is a tool which transfers data in both ways between relational systems and HDFS or other Hadoop data stores such as Hive or HBase. Sqoop can be used to import data from external structured databases into HDFS or any other related systems such as Hive and HBase. On the other hand, Sqoop can also be used to extract data from Hadoop and export it to external structured databases such as relational databases and enterprise data warehouses.

2.4.6 Oozie

Oozie is a job coordinator and workflow manager for jobs executed in Hadoop. It is integrated with the rest of the Apache Hadoop stack. It supports several types of Hadoop jobs, such as Java map-reduce, Streaming map-reduce, Pig, Hive and Sqoop as well as system-specific jobs such as Java programs and shell scripts. An Oozie workflow is a collection of actions and Hadoop jobs arranged in a Directed Acyclic Graph (DAG), since tasks are executed in a sequence and also are subject to certain constraints.

2.4.7 Mahout

Mahout is a scalable machine-learning and data-mining library. There are currently following four main groups of algorithms in Mahout:

1. Recommendations/Collective filtering.
2. Classification/Categorization.
3. Clustering.
4. Frequent item-set mining/Parallel frequent pattern mining.

Mahout is not simply a collection of pre-existing data mining algorithms. Many machine learning algorithms are non-scalable; that is, given the types of operations they perform, they cannot be executed as a set of parallel processes. But algorithms in the Mahout library can be executed in a distributed fashion, and have been written for MapReduce.

2.4.8 ZooKeeper

ZooKeeper is a distributed service with master and slave nodes for storing and maintaining configuration information, naming, providing distributed synchronization and providing group services in memory on ZooKeeper servers. ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical name space of data registers called znodes. HBase depends on ZooKeeper and runs a ZooKeeper instance by default.

2.5 Physical Architecture

Organizations tend to store more and more data in cloud environments, since clouds offer business users scalable resources on demand. Combining processor-based servers and storage, along with networking resources used in cloud environments, with big data processing tools such as Apache Hadoop software, provides the high-performance computing power needed to analyze vast amounts of data efficiently and cost-effectively. The machine configuration for storage and computing servers typically are 32 GB memory, four core processors and 200–320 GB hard disk. Running Hadoop in virtualized environments continues to evolve and mature with initiatives from open-source software projects. Figure 2.4 shows cloud computing infrastructure required for Big Data Analytics.

Every Hadoop-compatible file system should provide location awareness for effective scheduling of work: the name of the rack or the network switch where a worker node is. Hadoop application uses this information to find the data node and run the task. HDFS replicates data to keep different copies of the data on different racks. The goal is to reduce the impact of a rack power or switch failure (see Fig. 2.5).

A small Hadoop cluster includes a single master and multiple worker nodes. The master node consists of a JobTracker, TaskTracker, NameNode and DataNode. A slave or *worker node* acts as both a DataNode and TaskTracker, though it is possible to have data-only worker nodes and compute-only worker nodes.

In case of a larger cluster, the HDFS is managed through a dedicated NameNode server to host the file system index, and a secondary NameNode that can generate snapshots of the NameNode's memory structures, thus reducing the impact of loss of data. Similarly, a standalone JobTracker server can manage job scheduling. In clusters where the Hadoop MapReduce engine is deployed against an alternate file system, the NameNode, secondary NameNode and DataNode architecture of HDFS are replaced by the file-system-specific equivalents.

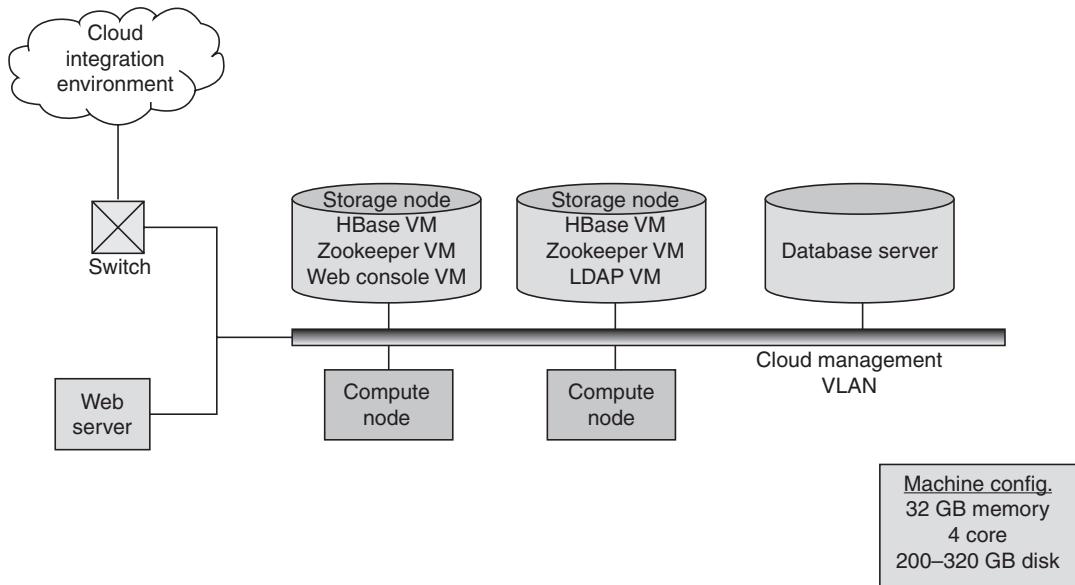


Figure 2.4 Cloud computing infrastructure to support Big Data Analytics.

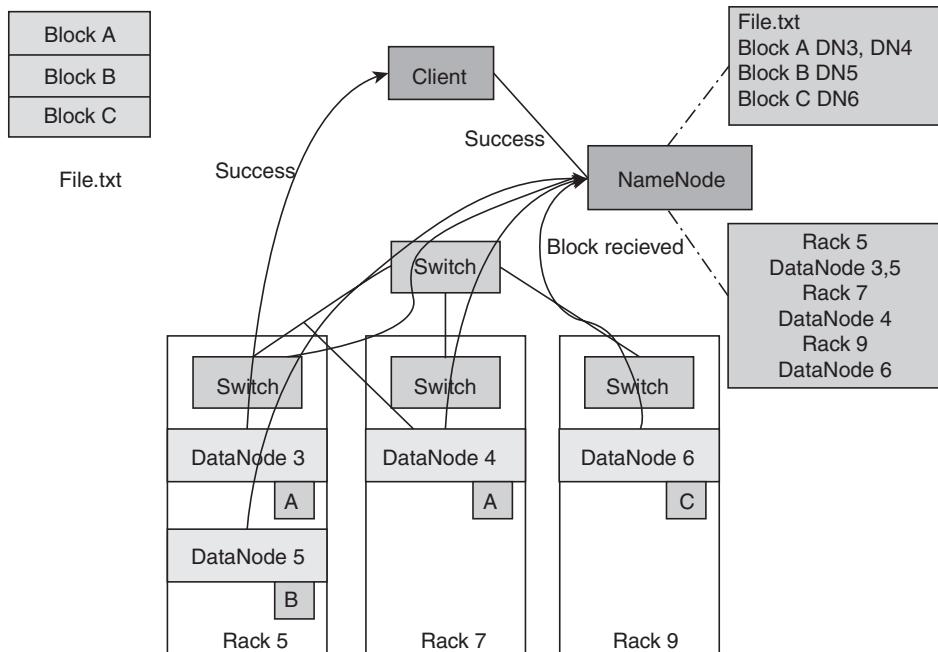


Figure 2.5 Hadoop-compatible file system provides location awareness.

HDFS stores large files in the range of gigabytes to terabytes across multiple machines. It achieves reliability by replicating the data across multiple hosts. Data is replicated on three nodes: two on the same rack and one on a different rack. Data nodes can communicate with each other to re-balance data and to move copies around. HDFS is not fully POSIX-compliant to achieve increased performance for data throughput and support for non-POSIX operations such as Append.

The HDFS file system includes a so-called secondary NameNode, which regularly connects with the primary NameNode and builds snapshots of the primary NameNode directory information, which the system then saves to local or remote directories. These check-pointed images can be used to restart a failed primary NameNode without having to replay the entire journal of file-system actions, then to edit the log to create an up-to-date directory structure.

An advantage of using HDFS is data awareness between the JobTracker and TaskTracker. The JobTracker schedules map or reduce jobs to TaskTrackers with an awareness of the data location. For example, if node *A* contains data (x, y, z) and node *B* contains data (a, b, c) , the JobTracker schedules node *B* to perform map or reduce tasks on (a, b, c) and node *A* would be scheduled to perform map or reduce tasks on (x, y, z) . This reduces the amount of traffic that goes over the network and prevents unnecessary data transfer.

When Hadoop is used with other file system, this advantage is not always available. This can have a significant impact on job-completion times, which has been demonstrated when running data-intensive jobs. HDFS was designed for mostly immutable files and may not be suitable for systems requiring concurrent write-operations.

2.6

Hadoop Limitations

HDFS cannot be mounted directly by an existing operating system. Getting data into and out of the HDFS file system can be inconvenient. In Linux and other Unix systems, a file system in Userspace (FUSE) virtual file system is developed to address this problem.

File access can be achieved through the native Java API, to generate a client in the language of the users' choice (C++, Java, Python, PHP, Ruby, etc.), in the command-line interface or browsed through the HDFS-UI web app over HTTP.

2.6.1 Security Concerns

Hadoop security model is disabled by default due to sheer complexity. Whoever's managing the platform should know how to enable it; else data could be at huge risk. Hadoop does not provide encryption at the storage and network levels, which is a major reason for the government agencies and others not to prefer to keep their data in Hadoop framework.

2.6.2 Vulnerable By Nature

Hadoop framework is written almost entirely in Java, one of the most widely used programming languages by cyber-criminals. For this reason, several experts have suggested dumping it in favor of safer, more efficient alternatives.

2.6.3 Not Fit for Small Data

While big data is not exclusively made for big businesses, not all big data platforms are suitable for handling small files. Due to its high capacity design, the HDFS lacks the ability to efficiently support the random reading of small files. As a result, it is not recommended for organizations with small quantities of data.

2.6.4 Potential Stability Issues

Hadoop is an open-source platform necessarily created by the contributions of many developers who continue to work on the project. While improvements are constantly being made, like all open-source software, Hadoop has stability issues. To avoid these issues, organizations are strongly recommended to make sure they are running the latest stable version or run it under a third-party vendor equipped to handle such problems.

2.6.5 General Limitations

Google mentions in its article that Hadoop may not be the only answer for big data. Google has its own Cloud Dataflow as a possible solution. The main point the article stresses is that companies could be missing out on many other benefits by using Hadoop alone.

Summary

- MapReduce brings compute to the data in contrast to traditional distributed system, which brings data to the compute resources.
- Hadoop stores data in a replicated and distributed way on HDFS. HDFS stores files in chunks which are physically stored on multiple compute nodes.
- MapReduce is ideal for operating on very large, unstructured datasets when aggregation across large datasets is required and this is accomplished by using the power of Reducers.
- Hadoop jobs go through a map stage and a reduce stage where
 - the mapper transforms the input data into key–value pairs where multiple values for the same key may occur.
 - the reducer transforms all of the key–value pairs sharing a common key into a single key–value.
- There are specialized services that form the Hadoop ecosystem to complement the Hadoop modules. These are HBase, Hive, Pig, Sqoop, Mahout, Oozie, Spark, Ambari to name a few.

Review Questions

1. What is Hadoop?
2. Why do we need Hadoop?
3. What are core components of Hadoop framework?
4. Give a brief overview of Hadoop.
5. What is the basic difference between traditional RDBMS and Hadoop?
6. What is structured, semi-structured and unstructured data? Give an example and explain.
7. What is HDFS and what are its features?
8. What is Fault Tolerance?
9. If replication causes data redundancy, then why is it pursued in HDFS?
10. What is a NameNode?
11. What is a DataNode?
12. Why is HDFS more suited for applications having large datasets and not when there are small files?
13. What is a JobTracker?
14. What is a TaskTracker?
15. What is a “block” in HDFS?
16. What are the benefits of block transfer?
17. What is a secondary NameNode? Does it substitute a NameNode?
18. What is MapReduce? Explain how do “map” and “reduce” work?
19. What sorts of actions does the JobTracker process perform?

Laboratory Exercise

A. Instructions to learners on how to run wordcount program on Cloudera

To start working with Hadoop for beginners, the best choice is to download ClouderaVM from their official website (and it is free).

Download Link:

http://www.cloudera.com/content/cloudera/en/downloads/quickstart_vms/cdh-5-4-x.html

Pre-requisite: Install VM Ware Player or Oracle Virtual Box.

For the above version of Cloudera we need virtual box.

INSTALLING AND OPENNING Cloudera in VIRTUAL BOX

STEP1: EXTRACT the downloaded zip file in the same folder or in home directory.

STEP2: Open virtualbox. Then

- Click New button which is at toolbar menu.
- A new window will open. Type name in the Name field, for example, “Cloudera”. Next in Type field select the type as “Linux”. In the version field select “Other Linux(64 bit)”.

- Click on Next Button. A new window will open. Select the RAM size. Click on Next Button.
- Here you have three options, out of which select “use an existing virtual Hard drive file”. Browse your Cloudera folder for file with .vmdk extension. Select that file and press ENTER.

Now as we have successfully created vm we can start Cloudera. So start it by clicking on start button. It will take some time to open. Wait for 2 to 3 minutes. Here the operating system is CentOS.

Once the system gets loaded we will start with the simple program called “wordcount” using MapReduce function which is a simple “hello world” kind of program for Hadoop.

STEPS FOR RUNNING WORDCOUNT PROGRAM:

1. OPEN the Terminal. Install a package “wget” by typing the following command:

```
$ sudo yum -y install wget
```

2. Make directory:

```
$ mkdir temp
```

3. Goto temp:

```
$ cd temp
```

4. Create a file with some content in it:

```
$ echo "This is SPIT and you can call me Sushil. I am good at statistical modeling and data analysis" > wordcount.txt
```

5. Make input directory in the HDFS system:

```
$ hdfsdfs -mkdir /user/cloudera/input
```

6. Copy file from local directory to HDFS file system:

```
$ hdfsdfs -put /home/cloudera/temp/wordcount.txt /user/cloudera/input/
```

7. To check if your file is successfully copied or not, use:

```
$ hdfsdfs -ls /user/cloudera/input/
```

8. To check hadoop-mapreduce-examples, use:

```
$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar
```

9. Run the wordcount program by typing the following command:

```
$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar wordcount /user/cloudera/input/wordcount.txt /user/cloudera/output
```

Note: The output will be generated in the output directory in HDFS file system and stored in part file “part-r-00000”.

- 10.** Check output directory:

```
$hdfsdfs -ls /user/cloudera/output
```

- 11.** Open the part file to check the output:

```
$ hdfsdfs -cat /user/cloudera/output/part-r-00000
```

Note: The above command will display the file content on the terminal. If you want to open it in the text editor then we need to copy it to our local file system. To do so, use the following command:

```
$ hdfsdfs -copyToLocal /user/cloudera/output/part-r-00000 /home/cloudera
```

B. Guidelines to Install Hadoop 2.5.2 on top of Ubuntu 14.04 and write WordCount Program in Java using MapReduce structure and test it over HDFS

Pre-requisite: Apache, JAVA, ssh packages must be installed. If not then follow the following steps.

B1. Steps for Installing Above Packages

- 1.** Before installing above packages, Create a new user to run the Hadoop (hduser or huser) and give it sudo rights:

- Create group name hadoop:

```
$ sudoaddgrouphadoop
```

- To create user and add it in group named Hadoop use

```
$ sudoadduser --ingrouphadoophduser
```

- To give sudo rights to hduser use

```
$ sudoadduserhdusersudo
```

- To switch user to hduser use

```
$ suhduser
```

- 2.** Install the following software:

```
# Update the source list
```

```
$ sudo apt-get update
```

2.1 Apache

```
$ sudo apt-get install apache2
```

```
# The OpenJDK project is the default version of Java.
```

```
# It is provided from a supported Ubuntu repository.
```

2.2 Java

```
$ sudo apt-get install default-jdk  
$ java -version
```

2.3 Installing SSH: ssh has two main components, namely,

- ssh: The command we use to connect to remote machines – the client.
- sshd: The daemon that is running on the server and allows clients to connect to the server.

The ssh is pre-enabled on Linux, but in order to start sshd daemon, we need to install ssh first. Use the following command to do so:

```
$ sudo apt-get install ssh
```

This will install ssh on our machine. Verify if ssh is installed properly with which command:

```
$ whichssh  
o/p:/usr/bin/ssh
```

```
$ whichsshd  
o/p:/usr/sbin/sshd
```

Create and Setup SSH Certificates: Hadoop requires SSH access to manage its nodes, that is, remote machines plus our local machine. For our single-node setup of Hadoop, we therefore need to configure SSH access to local host. So, we need to have SSH up and running on our machine and configured to allow SSH public key authentication. Hadoop uses SSH (to access its nodes) which would normally require the user to enter a password. However, this requirement can be eliminated by creating and setting up SSH certificates using the following commands. If asked for a filename just leave it blank and press the enter key to continue.

```
$ ssh-keygen -t rsa -P ""
```

Note: After typing the above command just press Enter two times.

```
$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

The second command adds the newly created key to the list of authorized keys so that Hadoop can use ssh without prompting for a password.

We can check if ssh works using the following command:

```
$ ssh localhost  
o/p:
```

The authenticity of host 'localhost (127.0.0.1)' cannot be established.

```
ECDSA key fingerprint is e1:8b:a0:a5:75:ef:f4:b4:5e:a9:ed:be:64:be:5c:2f.
```

```
Are you sure you want to continue connecting (yes/no)? yes
```

```
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
```

```
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-40-generic x86_64)
```

B2. Installing Hadoop

1. Download and extract the hadoop-2.5.2.tar.gz to the Downloads directory from the link given below:

<https://archive.apache.org/dist/hadoop/core/hadoop-2.5.2/>

2. To switch user to hduser use

```
$ sudo suhduser
```

3. To move hadoop to /usr/local/Hadoop use

```
$ sudo mv /home/admin/Downloads/* /usr/local/hadoop
```

4. To change the access rights use

```
sudo chown -R hduser:hadoop /usr/local/hadoop
```

B3. Setup Configuration Files

The following files will have to be modified to complete the Hadoop setup:

```
~/.bashrc  
/usr/local/hadoop/etc/hadoop/hadoop-env.sh  
/usr/local/hadoop/etc/hadoop/core-site.xml  
/usr/local/hadoop/etc/hadoop/mapred-site.xml.template  
/usr/local/hadoop/etc/hadoop/hdfs-site.xml
```

1. `~/.bashrc`: Before editing the `.bashrc` file in our home directory, we need to find the path where Java has been installed to set the `JAVA_HOME` environment variable using the following command:

```
$ update-alternatives --config java
```

Now we can append the following to the end of `~/.bashrc`:

```
$ vi ~/.bashrc
```

```
#HADOOP VARIABLES START  
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64  
export HADOOP_INSTALL=/usr/local/hadoop  
export PATH=$PATH:$HADOOP_INSTALL/bin  
export PATH=$PATH:$HADOOP_INSTALL/sbin  
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL  
export HADOOP_COMMON_HOME=$HADOOP_INSTALL  
export HADOOP_HDFS_HOME=$HADOOP_INSTALL  
export YARN_HOME=$HADOOP_INSTALL  
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_INSTALL/lib/native  
export HADOOP_OPTS="-Djava.library.path=$HADOOP_INSTALL/lib"  
#HADOOP VARIABLES END
```

```
$ source ~/.bashrc
```

Note that the JAVA_HOME should be set as the path just before the ‘.../bin/’:

```
$ javac -version  
$ whichjavac  
$ readlink -f /usr/bin/javac
```

2. /usr/local/hadoop/etc/hadoop/hadoop-env.sh: We need to set JAVA_HOME by modifying hadoop-env.sh file.

```
$ vi /usr/local/hadoop/etc/hadoop/hadoop-env.sh
```

Add the following configuration:

```
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
```

3. /usr/local/hadoop/etc/hadoop/core-site.xml: This file contains configuration properties that Hadoop uses when starting up. This file can be used to override the default settings that Hadoop starts with.

```
$ sudo mkdir -p /app/hadoop/tmp
```

```
$ sudo chown hduser:hadoop /app/hadoop/tmp
```

Open the file and enter the following in between the <configuration></configuration> tag:

```
$ vi /usr/local/hadoop/etc/hadoop/core-site.xml
```

```
<configuration>  
<property>  
<name>hadoop.tmp.dir</name>  
<value>/app/hadoop/tmp</value>  
<description>A base for other temporary directories.</description>  
</property>
```

```
<property>  
<name>fs.default.name</name>  
<value>hdfs://localhost:54310</value>  
<description>The name of the default file system. A URI whose  
scheme and authority determine the FileSystem implementation. The  
uri's scheme determines the config property (fs.SCHEME.impl) naming  
the FileSystem implementation class. The uri's authority is used to  
determine the host, port, etc. for a filesystem.</description>  
</property>  
</configuration>
```

4. /usr/local/hadoop/etc/hadoop/mapred-site.xml: By default, the /usr/local/hadoop/etc/hadoop/ folder contains /usr/local/hadoop/etc/hadoop/mapred-site.xml.template file which has to be renamed/copied with the name mapred-site.xml.

```
$ cp /usr/local/hadoop/etc/hadoop/mapred-site.xml.template /usr/local/hadoop/etc/hadoop/mapred-site.xml
```

The mapred-site.xml file is used to specify which framework is being used for MapReduce.

We need to enter the following content in between the <configuration></configuration> tag:

```
<configuration>
<property>
<name>mapred.job.tracker</name>
<value>localhost:54311</value>
<description>The host and port that the MapReduce job tracker runs at. If "local", then jobs are run in-process as a single map and reduce task.
</description>
</property>
</configuration>
```

5. /usr/local/hadoop/etc/hadoop/hdfs-site.xml: This file needs to be configured for each host in the cluster that is being used. It is used to specify the directories which will be used as the NameNode and the DataNode on that host. Before editing this file, we need to create two directories which will contain the NameNode and the DataNode for this Hadoop installation. This can be done using the following commands:

```
$ sudomkdir -p /usr/local/hadoop_store/hdfs/namenode
$ sudomkdir -p /usr/local/hadoop_store/hdfs/datanode
$ sudochown -R hduser:hadoop /usr/local/hadoop_store
```

Open the file and enter the following content in between the <configuration></configuration> tag:

```
$ vi /usr/local/hadoop/etc/hadoop/hdfs-site.xml
```

```
<configuration>
<property>
<name>dfs.replication</name>
<value>1</value>
<description>Default block replication.
```

The actual number of replications can be specified when the file is created. The default is used if replication is not specified in create time.

```
</description>
</property>
<property>
<name>dfs.namenode.name.dir</name>
<value>file:/usr/local/hadoop_store/hdfs/namenode</value>
</property>
<property>
<name>dfs.datanode.data.dir</name>
<value>file:/usr/local/hadoop_store/hdfs/datanode</value>
</property>
</configuration>
```

- B4. Format the New Hadoop File System:** Now, the Hadoop file system needs to be formatted so that we can start to use it. The format command should be issued with write permission since it creates current directory:

under /usr/local/hadoop_store/hdfs/namenode folder:

```
$ hadoop namenode -format
```

Note that hadoopnamenode -format command should be executed once before we start using Hadoop. If this command is executed again after Hadoop has been used, it will destroy all the data on the Hadoop file system.

Starting Hadoop: Now it is time to start the newly installed single node cluster. We can use start-all.sh or (start-dfs.sh and start-yarn.sh)

```
$ cd /usr/local/hadoop/sbin
```

```
$ start-all.sh
```

We can check if it is really up and running using the following command:

```
$ jps
```

o/p:

```
9026 NodeManager
7348 NameNode
9766 Jps
8887 ResourceManager
7507 DataNode
```

The output means that we now have a functional instance of Hadoop running on our VPS (Virtual private server).

```
$ netstat -plten | grep java
```

Stopping Hadoop

```
$ cd /usr/local/hadoop/sbin  
$ stop-all.sh
```

B5. Running Wordcount on Hadoop 2.5.2:

Wordcount is the hello_world program for MapReduce.
Code for wordcount program is:

```
package org.myorg;  
import java.io.IOException;  
import java.util.*;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.conf.*;  
import org.apache.hadoop.io.*;  
import org.apache.hadoop.util.*;  
import org.apache.hadoop.mapreduce.Mapper;  
import org.apache.hadoop.mapreduce.Reducer;  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.conf.Configured;  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
import org.apache.hadoop.mapred.JobConf;  
public class myWordCount {  
    public static class Map extends Mapper  
        <LongWritable, Text, Text, IntWritable> {  
        private final static IntWritable one = new IntWritable(1);  
        private Text word = new Text();  
        public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  
            String line = value.toString();  
            StringTokenizer tokenizer = new StringTokenizer(line);  
            while (tokenizer.hasMoreTokens()) {  
                word.set(tokenizer.nextToken());  
                context.write(word, one);  
            }  
        }  
    }  
    public static class Reduce extends Reducer  
        <Text, IntWritable, Text, IntWritable> {  
        public void reduce(Text key, Iterator<IntWritable> values, Context context) throws IOException, InterruptedException {
```

```
int sum = 0;
while (values.hasNext()) {
    sum += values.next().get();
}
context.write(key, new IntWritable(sum));
}
}

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    conf.set("mapreduce.job.queuename", "apg_p7");
    System.out.println("This is a new version");
    Job job = new Job(conf);
    job.setJarByClass(myWordCount.class);
    job.setJobName("myWordCount");
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setMapperClass(myWordCount.Map.class);
    job.setCombinerClass(myWordCount.Reduce.class);
    job.setReducerClass(myWordCount.Reduce.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.waitForCompletion(true);
}
}
```

Note: Copy the above code and save it with .java extension. To run the program under MapReduce, the following steps needs to be done:

1. Put source code under this location

```
/project/src/org/myorg/myWordCount.java
```

2. Compile java code

```
$ mkdir /project/class;
$ cd /project;
$ javac -classpath `yarn classpath` -d ./src/org/myorg/*.java
```

3. Create manifest.txt file

```
$ cd project/class;
$ sudo vim manifest.txt;
The content of manifest.txt is
Main-Class: org.myorg.myWordCount
Leave an empty line at the end of manifest.txt
```

4. To Generate jar file

```
$ jar -cvmf manifest.txt myWordCount.jar org
```

5. Put input data on HDFS

```
$ mkdir input
```

```
$ echo "hadoop is fast and hadoop is amazing, hadoop is new Technology for Big Data Processing" > input/file1
```

```
$ hadoop fs -put input /user/hadoop
```

6. Run the program

```
hadoop jar myWordCount.jar /user/hadoop/input /user/hadoop/output
```


3

What is NoSQL?

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Understand NoSQL business drivers.
- Learn the desirable features of NoSQL that drive business.
- Learn the need for NoSQL through case studies.
- Learn NoSQL data architectural pattern.
- Learn the variations in NoSQL architectural pattern.
- Learn how NoSQL is used to manage big data.
- Learn how NoSQL system handles big data problems.

3.1 What is NoSQL?

NoSQL is database management system that provides mechanism for storage and retrieval of massive amount of unstructured data in a distributed environment on virtual servers with the focus to provide high scalability, performance, availability and agility.

In other words, NoSQL was developed in response to a large volume of data stored about users, objects and products that need to be frequently accessed and processed. Relational databases are not designed to scale and change easily to cope up with the needs of the modern industry. Also they do not take advantage of the cheap storage and processing power available today by using commodity hardware.

NoSQL database is also referred as **Not only SQL**. Most NoSQL systems are entirely non-relational; they do not have fixed schemas or JOIN operations. Instead they use objects, key-value pairs, or tuples.

Some of the NoSQL implementations are SimpleDB, Google BigTable, Apache Hadoop, MapReduce and MemcacheDB. There are approximately 150 NoSQL databases available in the market. Companies that largely use NoSQL are NetFlix, LinkedIn and Twitter for analyzing their social network data.

In short:

1. NoSQL is next generation database which is completely different from the traditional database.
2. NoSQL stands for Not only SQL. SQL as well as other query languages can be used with NoSQL databases.

3. NoSQL is non-relational database, and it is schema-free.
4. NoSQL is free of JOINs.
5. NoSQL uses distributed architecture and works on multiple processors to give high performance.
6. NoSQL databases are horizontally scalable.
7. Many open-source NoSQL databases are available.
8. Data file can be easily replicated.
9. NoSQL uses simple API.
10. NoSQL can manage huge amount of data.
11. NoSQL can be implemented on commodity hardware which has separate RAM and disk (shared-nothing concept).

3.1.1 Why NoSQL?

The reality is that a traditional database model does not offer the best solution for all scenarios in applications. A relational database product can cater to a more predictable, structured data. NoSQL is required because today's industry needs a very agile system that can process unstructured and unpredictable data dynamically. NoSQL is known for its high performance with high availability, rich query language, and easy scalability which fits the need. NoSQL may not provide atomicity, consistency, isolation, durability (ACID) properties but guarantees eventual consistency, basically available, soft state (BASE), by having a distributed and fault-tolerant architecture.

3.1.2 CAP Theorem

Consistency, Availability, Partition tolerance (**CAP theorem**, also called as **Brewer's theorem**, states that it is not possible for a distributed system to provide all three of the following guarantees simultaneously:

1. Consistency guarantees all storage and their replicated nodes have the same data at the same time.
2. Availability means every request is guaranteed to receive a success or failure response.
3. Partition tolerance guarantees that the system continues to operate in spite of arbitrary partitioning due to network failures.

3.2

NoSQL Business Drivers

Enterprises today need highly reliable, scalable and available data storage across a configurable set of systems that act as storage nodes. The needs of organizations are changing rapidly. Many organizations operating with single CPU and Relational database management systems (RDBMS) were not able to

cope up with the speed in which information needs to be extracted. Businesses have to capture and analyze a large amount of variable data, and make immediate changes in their business based on their findings.

Figure 3.1 shows RDBMS with the business drivers velocity, volume, variability and agility necessitates the emergence of NoSQL solutions. All of these drivers apply pressure to single CPU relational model and eventually make the system less stable.

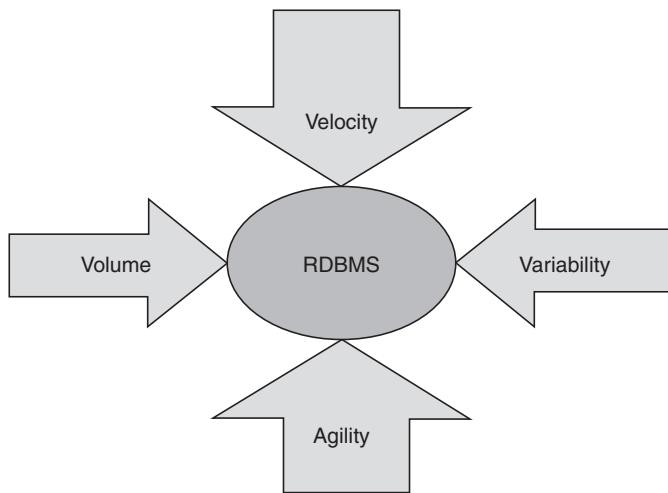


Figure 3.1 NoSQL business drivers.

3.2.1 Volume

There are two ways to look into data processing to improve performance. If the key factor is only speed, a faster processor could be used. If the processing involves complex (heavy) computation, Graphic Processing Unit (GPU) could be used along with the CPU. But the volume of data is limited to on-board GPU memory. The main reason for organizations to look at an alternative to their current RDBMSs is the need to query big data. The need to horizontal scaling made organizations to move from serial to distributed parallel processing where big data is fragmented and processed using clusters of commodity machines. This is made possible by the development of technologies like Apache Hadoop, HDFS, MapR, HBase, etc.

3.2.2 Velocity

Velocity becomes the key factor when frequency in which online queries to the database made by social networking and e-commerce web sites have to be read and written in real time. Many single CPU, RDBMS systems are unable to cope up with the demands of real-time inserts. RDBMS systems frequently index on many columns that decrease the system performance. For example, when online shopping sites introduce great discount schemes, the random bursts in web traffic will slow down the

response for every user and tuning these systems as demand increases can be costly when both high read and write is required.

3.2.3 Variability

Organizations that need to capture and report on certain uncommon data, struggle when attempting to use RDBMS fixed schema. For example, if a business process wants to store a few special attributes for a few set of customers, then it needs to alter its schema definition. If a change is made to the schema, all customer rows within the database will also have this column. If there is no value related to this for most of the customers, then the row column representation will have sparse matrix. In addition to this, new columns to an RDBMS require to execute ALTER TABLE command. This cannot be done on the fly since the present executing transaction has to complete and database has to be closed, and then schema can be altered. This process affects system availability, which means losing business.

3.2.4 Agility

The process of storing and retrieving data for complex queries in RDBMS is quite cumbersome. If it is a nested query, data will have nested and repeated subgroups of data structures that are included in an object-relational mapping layer. This layer is responsible to generate the exact combination of SELECT, INSERT, DELETE and UPDATE SQL statements to move the object data from and to the backend RDBMS layer. This process is not simple and requires experienced developers with the knowledge of object-relational frameworks such as Java Hibernate. Even then, these change requests can cause slowdowns in implementation and testing.

Desirable features of NoSQL that drive business are listed below:

1. **24 × 7 Data availability:** In the highly competitive world today, downtime is equated to real dollars lost and is deadly to a company's reputation. Hardware failures are bound to occur. Care has to be taken that there is no single point of failure and system needs to show fault tolerance. For this, both function and data are to be replicated so that if database servers or "nodes" fail, the other nodes in the system are able to continue with operations without data loss. NoSQL database environments are able to provide this facility. System updates can be made dynamically without having to take the database offline.
2. **Location transparency:** The ability to read and write to a storage node regardless of where that I/O operation physically occurs is termed as "Location Transparency or Location Independence". Customers in many different geographies need to keep data local at those sites for fast access. Any write functionality that updates a node in one location, is propagated out from that location so that it is available to users and systems at other locations.
3. **Schema-less data model:** Most of the business data is unstructured and unpredictable which a RDBMS cannot cater to. NoSQL database system is a schema-free flexible data model that can easily accept all types of structured, semi-structured and unstructured data. Also relational model has scalability and performance problems when it has to manage large data volumes.

NoSQL data model can handle this easily to deliver very fast performance for both read and write operations.

4. **Modern day transaction analysis:** Most of the transaction details relate to customer profile, reviews on products, branding, reputation, building business strategy, trading decisions, etc. that do not require ACID transactions. The data consistency denoted by “C” in ACID property in RDBMSs is enforced via foreign keys/referential integrity constraints. This type of consistency is not required to be used in progressive data management systems such as NoSQL databases since there is no JOIN operation. Here, the “Consistency” is stated in the CAP theorem that signifies the immediate or eventual consistency of data across all nodes that participate in a distributed database.
5. **Architecture that suits big data:** NoSQL solutions provide modern architectures for applications that require high degrees of scale, data distribution and continuous availability. For this multiple data center support with which a NoSQL environment complies is one of the requirements. The solution should not only look into today's big data needs but also suit greater time horizons. Hence big data brings four major considerations in enterprise architecture which are as follows:
 - *Scale of data sources:* Many companies work in the multi-terabyte and even petabyte data.
 - *Speed is essential:* Overnight extract-transform-load (ETL) batches are insufficient and real-time streaming is required.
 - *Change in storage models:* Solutions like Hadoop Distributed File System (HDFS) and unstructured data stores like Apache Cassandra, MongoDB, Neo4j provide new options.
 - Multiple compute methods for Big Data Analytics must be supported.

Figure 3.2 shows the architecture that suits big data.

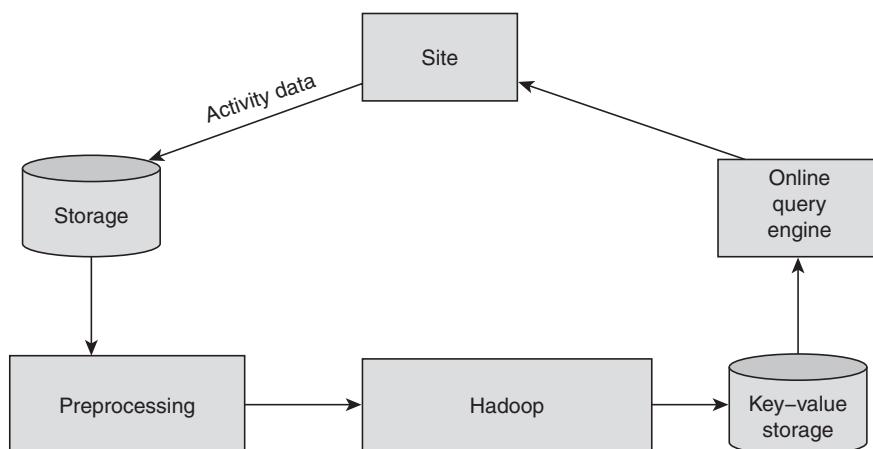


Figure 3.2 The architecture that suits big data.

- 6. Analytics and business intelligence:** A key business strategic driver that suggests the implementation of a NoSQL database environment is the need to mine the data that is being collected in order to derive insights to gain competitive advantage. Traditional relational database system poses great difficulty in extracting meaningful business intelligence from very high volumes of data. NoSQL database systems not only provide storage and management of big data but also deliver integrated data analytics that provides instant understanding of complex datasets and facilitate various options for easy decision-making.

3.3 NoSQL Case Studies

Four case studies are discussed in the following subsections and each of them follow different architectural pattern, namely, key-value store, Column Family/BigTable, Document store and Graph store.

3.3.1 Amazon DynamoDB

Amazon.com has one of the largest e-commerce operations in the world. Customers from all around the world shop all hours of the day. So the site has to be up 24×7 . Initially Amazon used RDBMS system for shopping cart and checkout system. Amazon DynamoDB, a NoSQL store brought a turning point.

DynamoDB addresses performance, scalability and reliability, the core problems of RDBMS when it comes to growing data. Developers can store unlimited amount of data by creating a database table and DynamoDB automatically saves it at multiple servers specified by the customer and also replicates them across multiple “Available” Zones. It can handle the data and traffic while maintaining consistent, fast performance. The cart data and session data are stored in the key-value store and the final (completed) order is saved in the RDBMS as shown in Fig. 3.3.

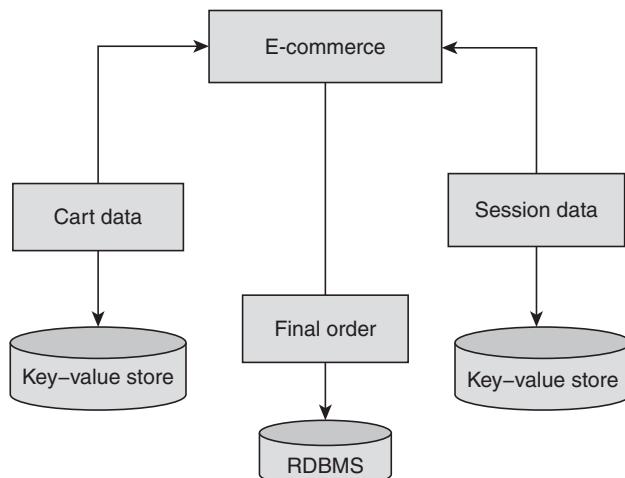


Figure 3.3 E-commerce shopping cart uses key-value store.

The salient features of key-value store are as follows:

1. **Scalable:** If application's requirements change, using the AWS Management Console or the DynamoDB APIs table throughput capacity can be updated.
2. **Automated storage scaling:** Using the DynamoDB write APIs, more storage can be obtained, whenever additional storage is required.
3. **Distributed horizontally shared nothing:** Seamlessly scales a single table over hundreds of commodity servers.
4. **Built-in fault tolerance:** DynamoDB automatically replicates data across multiple available zones in synchronous manner.
5. **Flexible:** DynamoDB has schema-free format. Multiple data types (strings, numbers, binary and sets) can be used and each data item can have different number of attributes.
6. **Efficient indexing:** Every item is identified by a primary key. It allows secondary indexes on non-key attributes, and query data using an alternate key.
7. **Strong consistency, Atomic counters:** DynamoDB ensures strong consistency on reads (only the latest values are read). DynamoDB service also supports atomic counters to atomically increment or decrement numerical attributes with a single API call.
8. **Secure:** DynamoDB uses cryptography to authenticate users. Access control for users within organization can be secured by integrating with AWS Identity and Access Management.
9. **Resource consumption monitoring:** AWS Management Console displays request throughput and latency for each DynamoDB table, since it is integrated with CloudWatch.
10. **Data warehouse – Business intelligence facility:** Amazon Redshift Integration – data from DynamoDB tables can be loaded into Amazon Redshift (data warehouse service).
11. **MapReduce integration:** DynamoDB is also integrated with Amazon Elastic MapReduce to perform complex analytics (hosted pay-as-you-go Hadoop framework on AWS).

3.3.2 Google's BigTable

Google's motivation for developing BigTable is driven by its need for massive scalability, better performance characteristics, and ability run on commodity hardware. Each time when a new service or increase in load happens, its solution BigTable would result in only a small incremental cost. Volume of Google's data generally is in petabytes and is distributed over 100,000 nodes.

BigTable is built on top of Google's other services that have been in active use since 2005, namely, Google File System, Scheduler, MapReduce and Chubby Lock Service. BigTable is a column-family database which is designed to store data tables (sparse matrix of row, column values) as section of column of data. It is distributed (high volume of data), persistent, multi-dimensional sorted map. The map is indexed by a row key, column key and a timestamp (int64).

3.3.3 MongoDB

MongoDB was designed by Eliot Horowitz with his team in 10gen. MongoDB was built based on their experiences in building large scale, high availability, robust systems. MongoDB was thought of changing the data model of MySQL from relational to document based, to achieve speed, manageability, agility, schema-less databases and easier horizontal scalability (also JOIN free). Relational databases like MySQL or Oracle work well with, say, indexes, dynamic queries and updates. MongoDB works exactly the same way but has the option of indexing an embedded field.

MongoDB is a document data model that stores data in JSON documents. JSON model seamlessly maps to native programming languages and allows dynamic schema which helps the data model to evolve than have fixed schema like RDBMS.

3.3.4 Neo4j

Neo4j is an open-source (source code is available in github) sponsored by Neo Technology. Its NoSQL graph database is implemented in Java and Scala. Its development started in 2003; it was made publicly available since 2007. Neo4j is used today by hundreds to thousands of enterprises. To name a few: scientific research, routing, matchmaking, network management, recommendations, social networks, software analytics, organizational and project management.

In live scenario, many domains remain connected and graph database embraces relationships that exist between them as a core aspect of its data model. It is able to store, process, and query such connections efficiently whereas other databases compute relationships expensively at query time. Accessing those already persistent connections is efficient for any “join-like” navigation operation that allows us to quickly traverse millions of connections per second per core.

Irrespective of the total size of the dataset, graph databases manage highly connected data and complex queries. Using a pattern and a set of starting points, graph databases explore the neighborhood that exists around the initial starting points, collects and aggregates information from millions of nodes and relationships.

The property graph contains nodes (entities) that are connected and can hold any number of key-value pairs (attributes). Nodes can be tagged with labels representing their roles to contextualize node and relationship properties. Labels also serve to attach metadata, index, or constraint information to nodes if required.

Neo4j implements the Property Graph Model efficiently down to the storage level. In contrast to graph processing or in-memory libraries, Neo4j supports ACID transaction compliance and runtime failover, making it suitable for production scenarios.

A relationship has a direction, a type, a *start node* and an *end node*. Relationships have quantitative properties, such as weights, costs, distances, ratings, time intervals or strengths. As relationships provide semantically relevance to connections, two nodes can share any number or type of relationships without sacrificing performance. Relationships can be navigated regardless of direction.

3.4**NoSQL Data Architectural Patterns****3.4.1 Types of NoSQL Data Stores**

In this section we will discuss the following types of NoSQL data stores:

1. Key-value store.
2. Column store.
3. Document store.
4. Graph store.

3.4.1.1 Key-Value Store

The key-value store uses a key to access a value. The key-value store has a schema-less format. The key can be artificially generated or auto-generated while the value can be String, JSON, BLOB, etc. For example, key could be a web page, file path, REST call, image name, SQL query, etc.

The key-value type uses a hash table with a unique key and a pointer to a particular item of data. A bucket is a logical group (not physical) of keys and so different buckets can have identical keys. The real key is a hash (Bucket + Key). Use of the cache mechanisms for the mappings improves the performance.

The clients can read and write values using a key as shown below:

1. To fetch the value associated with the key use – Get(key).
2. To associate the value with the key use – Put(key, value).
3. To fetch the list of values associated with the list of keys use – Multi-get (key1, key2, ..., keyN).
4. To remove the entry for the key from the data store use – Delete(key).

In addition to Get, Put, and Delete API, key-value store has two rules:

1. *Distinct keys*: All keys in key-value store are unique.
2. *No queries on values*: No queries can be performed on the values of the table.

Weakness of Key-Value Stores

Key-value stores work around the Availability and Partition aspects but lack in Consistency. Hence, they cannot be used for updating part of a value or query the database. In other words, key-value model cannot provide any traditional database capabilities.

Another weakness of key-value model is that as volume of data increases it will be difficult to maintain unique value as keys.

Examples of Key–Value Stores

Redis, Amazon Dynamo, Azure Table Storage (ATS), Riak, Memcache, etc.

Uses of Key–Value Stores

Dictionary, image store, lookup tables, cache query, etc. A key–value store is similar to Dictionary where for a word (key) all the associated words (noun/verb forms, plural, picture, phrase in which the word is used, etc.) and meaning (values) are given.

External websites are stored as key–value store in Google’s database. Amazon S3 (simple storage service) makes use of key–value store to save the digital media content like photos, music, videos in the cloud.

In a key–value store, static component is the URL of the website and images. The dynamic component of the website generated by scripts is not stored in key–value store.

3.4.1.2 Column Family Store/Wide Column Store

Apache Cassandra, named after the Greek mythological prophet, was developed at Facebook by Avinash Lakshman and Prashant Malik and later released as an open-source project on Google code in July 2008.

Column family is different from column store. Instead of storing data in rows, these column store databases are designed to store data tables as section of column of data, rather than as rows of data. They are generally used in sparse matrix system. Wide-column store offers very high performance and a highly scalable architecture. Column stores are used in OLAP systems for their ability to rapidly aggregate column data.

Column family store uses the same concept of spreadsheet where a cell is identified by row (number) and column (name) identifier. Similarly, a column family store uses a row and a column as keys. In addition to this, a column family to group similar column names together. A timestamp is also included to store multiple versions of a value over time. Therefore <key, value> is <(rowed, column family, column name, timestamp), value>. Like traditional databases all the column values for each row need not be stored. Examples include: BigTable from Google, HBase, Accumulo, HyperTable, Cassandra.

Figure 3.4 shows the sample of tweets stored as column family; key and value are marked.

Cassandra’s data model is a column-family-based one. Cassandra has a multi-dimensional data model. Table defines a high-level grouping of the data and in Cassandra it is referred to as a keyspace. There’s one keyspace for each application.

The column family is the basic unit of data organization and it lies within a keyspace. A column family stores together a set of columns for a given row. In other words, in a column-oriented database, data is stored by column, with all rows grouped together. This optimizes the column design by grouping commonly queried columns together. Consider the following:

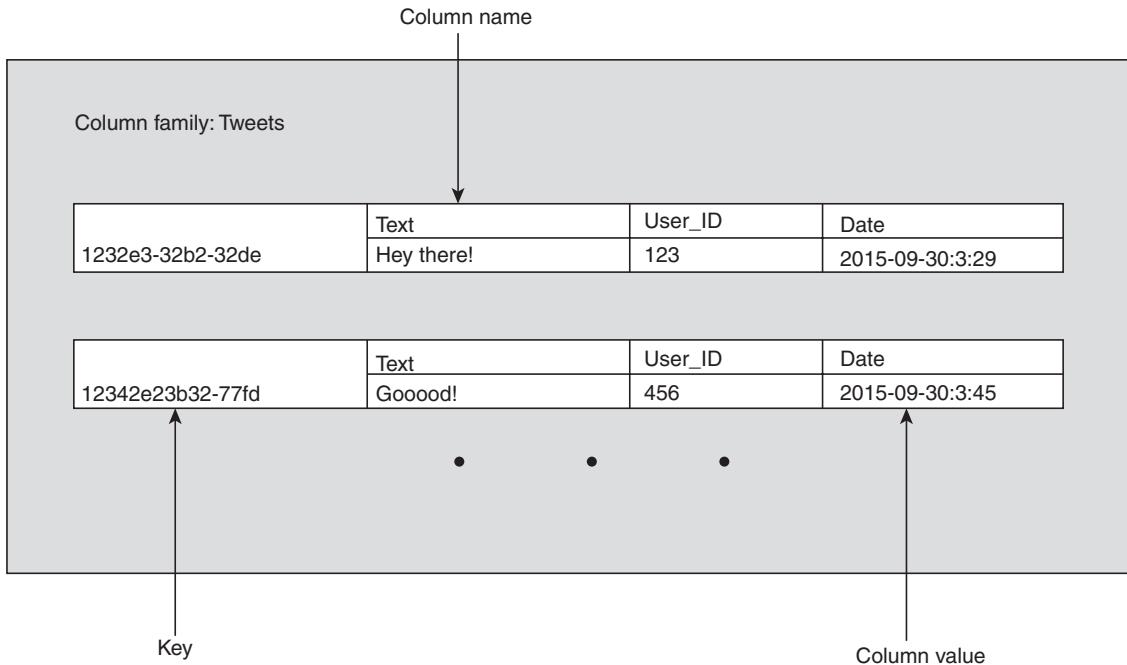


Figure 3.4 Sample column family store.

```
EmployeeIndia : {
    address: {
        city: Mumbai
        pincode: 400058
    },
    projectDetails: {
        durationDays: 100
        cost: 50000
    }
}
```

Here

1. The outermost key EmployeeIndia is analogous to row.
2. “address” and “projectDetails” are called column families.
3. The column-family “address” has columns “city” and “pincode”.
4. The column-family “projectDetails” has columns “durationDays” and “cost”.
5. Columns can be referenced using ColumnFamily.

Key is the unique identifier for a record. Cassandra will index the key. Data (record) is stored in a column, and a column is expressed as a basic key–value relationship. All column entries are timestamped. Columns can be stored sorted alphabetically, or by timestamp. Columns can be defined on the fly.

Keyspaces and column families must be defined in the storage-conf.xml file before startup.

3.4.1.3 Document Store

Document store basically expands on the basic idea of key–value stores where “documents” are more complex, in that they contain data and each document is assigned a unique key, which is used to retrieve the document. These are designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data. Hierarchical (tree) data structures can be directly stored in document database.

The column family and key–value store lack a formal structure and so cannot be indexed. Hence searching is not possible. This problem is resolved in document store. Key here is a simple id. Irrespective of the id, a query can result in getting any item (value or content) out of a document store.

This is made possible in a document store because everything inside a document is automatically indexed. So if you want to search for a particular property, all the documents with the same property can be quickly found. The difference in key–value store and document store is that the key–value store stores into memory the entire document in the value portion whereas document store extracts subsections of all documents.

Document store uses a tree structure. “Document path” is used like a key to access the leaf values of a document tree structure. So the search item’s exact location can also be found. For example, if the root is Employee, the path can be

```
Employee[id='2300']/Address/street/BuildingName/text()
```

Though the document store tree structure is complex the search API is simple.

Document structure uses JSON (JavaScript Object Notation) format for deep nesting of tree structures associated with serialized objects. But JSON does not support document attributes such as bold, hyperlinks, etc.

Examples include: MongoDB, CouchBase, and CouchDB.

MongoDB uses a document model. Document can be thought of as a row or tuple in RDBMS. Document (object) map to data type in programming language. A MongoDB database holds a collection of documents. Embedded documents and arrays reduce need for JOINs, which is a key factor for improving performance and speed. MongoDB focuses on speed, power, ease of use and flexibility.

MongoDB became a popular NoSQL product because of its Ad serving. This service of MongoDB will quickly send at the same time a banner Ad to millions of users who are interested in the Ad product or service, when they browse a web page. The Ad servers have to be up and available 24 × 7. They should apply complex business rules to select the most appropriate Ad and place it on the user’s

webpage when it loads. MongoDB also services content management, real-time sentiment analysis, product management, to store user data in video games and similar domains.

MongoDB stores document in Binary JSON (BSON) format. Documents with similar structure are organized as collections, similar to table in RDBMS; documents are similar to rows and fields are similar to columns. All the data is spread across many tables after normalization in RDBMS, whereas in MongoDB all data related to a single record reside in a single document. In a document, multiple arrays of sub-documents can be embedded. Single read statement can get the entire document. Data in MongoDB is localized; hence there is no need for any JOIN operations. Documents are self-describing, so schema can vary from document to document.

3.4.1.4 Graph Store

Graph store is based on graph theory. These databases are designed for data whose relations are represented as a graph and have elements which are interconnected, with an undetermined number of relations between them.

Examples include: Neo4j, AllegroGraph, TeradataAster.

Graph databases are used when a business problem has complex relationship among their objects, especially in social networks and rule-based engines. Social networking apps like Facebook, Google+, LinkedIn, Twitter, etc. and video hosting applications like YouTube, Flickr, etc. use graph database to store their data. In Google+, profile data are stored internally as nodes and nodes are connected with each other using relationships. Graph databases store nodes and links in an optimized way so that querying these graph stores (graph traversal) is simpler. It does not require complex JOINS or indexes to retrieve connected data from its adjacent node.

Graph stores contain sequence of nodes and relations that form the graph. Both nodes and relationships contain properties like follows, friend, family, etc. So a graph store has three fields: nodes, relationships and properties. Properties are key-value pairs that represent data. For example, Node Name = “Employee” and (empno:2300, empname: “Tom”, deptno: 20) are a set of properties as key-value pairs. Dept can be another node. The relationship that connects them could be “works_for”. “works_for” relationship has empno: 2300 property as key-value pair. Each relationship has start/from (here “Employee”) node and end/to (“Dept”) node. In Neo4j relationships should be directional (unidirectional or bidirectional), else it will throw error. Neo4j does not support Sharding.

Business Use Case:

1. *Link analysis:* To search and look for patterns and relationships in social networking applications. To know mutual friends in LinkedIn to get introduced. The pattern of incoming/outgoing calls in crime and detection. Money transfer patterns in credit card transactions.
2. Rules and inference are used when queries have to be run on complex structures such as taxonomies, class libraries and rule-based systems. RDF graphs are used to store rules and logic. Friends refer trusted review of movies, books, music, restaurants, etc. to select from alternatives.

3. Integrating linked data with publically available data in realtime and build mashups without storing data. Business analytics like trend analysis, customer targeting, and sentimental analysis need to integrate with public datasets.

Some of Neo4j features are listed below:

1. Neo4j has CQL, Cypher query language much like SQL.
2. Neo4j supports Indexes by using Apache Lucence.
3. It supports UNIQUE constraints.
4. Neo4j Data Browser is the UI to execute CQL Commands.
5. It supports ACID properties of RDBMS.
6. It uses Native Graph Processing Engine to store graphs.
7. It can export query data to JSON and XLS format.
8. It provides REST API to Java, Scala, etc.

3.5

Variations of NoSQL Architectural Patterns

The key-value store, column family store, document store and graph store patterns can be modified based on different aspects of the system and its implementation. Database architecture could be distributed (manages single database distributed in multiple servers located at various sites) or federated (manages independent and heterogeneous databases at multiple sites).

In Internet of Things (IoT) architecture a virtual sensor has to integrate multiple data streams from real sensors into a single data stream. Results of the queries can be stored temporarily and consumed by the application or stored permanently if required for later use. For this, it uses data and sources-centric IoT middleware.

Scalable architectural patterns can be used to form new scalable architectures. For example, a combination of load balancer and shared-nothing architecture; distributed Hash Table and Content Addressable network (Cassandra); Publish/Subscribe (EventJava); etc.

The variations in architecture are based on system requirements like agility, availability (anywhere, anytime), intelligence, scalability, collaboration and low latency. Various technologies support the architectural strategies to satisfy the above requirement. For example, agility is given as a service using virtualization or cloud computing; availability is the service given by internet and mobility; intelligence is given by machine learning and predictive analytics; scalability (flexibility of using commodity machines) is given by Big Data Technologies/cloud platforms; collaboration is given by (enterprise-wide) social network application; and low latency (event driven) is provided by in-memory databases.

3.6 Using NoSQL to Manage Big Data

NoSQL solution is used to handle and manage big data. NoSQL with their inherently horizontal scale-out architectures solves big data problems by moving data to queries, uses hash rings to distribute the load, replicates the scale reads, and allows the database to distribute queries evenly to DataNodes in order to make systems run fast.

3.6.1 What is a Big Data NoSQL Solution?

A decade ago, NoSQL was deployed in companies such as Google, Amazon, Facebook and LinkedIn. Nowadays, most enterprises that are customer-centric and revenue-driving applications that serve millions of consumers are adopting this database. The move is motivated by the explosive growth of mobile devices, the IoT and cloud infrastructure. The need of industries for scalability and performance requirements was rising which the relational database technology was never designed to address. Thus, enterprises are turning to NoSQL to overcome these limitations.

A few of the case studies which require NoSQL kind of databases are listed in the following subsections.

3.6.1.1 Recommendation

Movie, music, books, products in e-commerce sites, articles for researchers, e-learning materials, travel/holiday packages are a few examples where recommendation plays an important part to improve business. Personalization (an advertisement, discount coupon or recommendation) is an opportunity to assist the right customer at the right time with the right product. But the ability to consume, process and use the necessary amount of data to create personalized experiences is a great challenge for relational databases. A NoSQL database has flexible data models to build and update user profile on the fly and so can elastically scale to meet the most data workload demand, and delivers the low latency required for real-time data management.

3.6.1.2 User Profile Management

User profile management is a key process in web and mobile applications as it enables other processes like online transactions, user preferences, user authentication, etc. As the number of users increases, the complexity of user profile data and user experience accelerate the expectations. It becomes extremely difficult for RDBMS to keep up with scalability, data flexibility, and performance requirements. NoSQL can easily scale-out its architecture to handle flexibility in changing data types and delivers faster read/write performance.

3.6.1.3 Real-Time Data Handling

An agile enterprise needs to extract information from the current transactional data in real time to increase the efficiency, reduce costs and increase revenue. Hadoop is designed for Big Data Analytics, but it is not real time. NoSQL is designed for real-time data, but it is operational rather than analytical. Therefore, NoSQL together with Hadoop can handle real-time big data.

3.6.1.4 Content Management

Every enterprise tries to give user rich, informative content to satisfy his/her experience. Content includes all kinds of semi-structured and unstructured data such as images, audio, video presentations, as well as user-generated content, such as reviews, photos, videos, ratings, and comments. Relational databases are not designed to manage different content types due to their fixed data model whereas NoSQL databases have flexible data model to handle variety of data.

3.6.1.5 Catalog Management

Enterprises that offer more products and services have to collect more reference data. Catalogs have to be classified by application and business unit. It is expected that multiple applications should be able to access multiple databases, which creates complexity and data management issues. NoSQL enables enterprises to more easily aggregate catalog data within a single database with its flexible data model.

3.6.1.6 360-Degree Customer View

The 360-degree view of customers also often requires combining structured data from various applications with unstructured data that resides on social media platforms. Many companies are trying to develop a framework to combine these sources of data and to analyze them in a central location. It has become necessary for this software to integrate with other platforms to enable data sharing and a cohesive, up-to-date, accurate view of customers. In some cases, this may involve the use of application programming interfaces to enable applications to share data. Customers expect a consistent experience, while the enterprise wants to provide the highest level of customer service to capitalize on up-sell/cross-sell opportunities. As the number of services and products, channels, brands and business units increases, customer data is stored in silos. NoSQL enables multiple applications to access the same customer data, but to add new attributes without affecting other applications.

3.6.1.7 Mobile Applications

Most of the e-commerce sites have taken their business to mobile platforms. Statistics say that the mobile users spend only 20% of their time in browsing and 80% on mobile applications. Business on Mobile apps presents a number of challenges. Mobile apps require scalability, performance, and availability that RDBMS are not equipped to address. With NoSQL, scalability is made possible by starting with a small deployment and expanding as the number of users increase. Small version can be developed faster and launched as early as possible to capture business, and not lose it to their competitors.

3.6.1.8 Internet of Things

IoT helps companies to reduce costs and time-to-market their new products and services, and enhance customer satisfaction. IoT's ability to access real time, global operational data increases business agility. RDBMs struggle with the volume, variety and velocity of the IoT data. NoSQL helps in scaling and storing this data. It also enables concurrent data access to millions of network connected devices and systems.

3.6.1.9 Fraud Detection

Fraud detection is essential for organizations offering financial services to reduce loss, minimize financial exposure and comply with regulations. When customers use payment gateway to pay with a credit or debit card, it is necessary to give immediate confirmation. Before giving confirmation, the transaction must be processed by a fraud detection system, which has to assess customer data and process it in less than a millisecond. RDBMS cannot meet this low-latency requirement. NoSQL can provide data access at the speed of memory by making use of an integrated cache mechanism.

3.6.2 Understanding Types of Big Data Problems

After e-commerce boom, single database does not fit the entire web applications. While designing a software solution, more than one storage mechanism is thought of based on the needs. Polyglot persistence takes a hybrid approach to persistence. It leverages the strength of multiple data stores. In fact, it is about selecting the best data store for a given data type and purpose. NoSQL implies this. Big data problems are of many types, each requiring a different combination of NoSQL systems. First step is to categorize the data and determine its type. Big Data classification system is based on the way the organization uses its data; it can be “Read (mostly)” or “Read-Write”. “Read-Write” is for transaction data (ACID) and so requires high availability.

Read (mostly) data is read once and rarely changed. This type of data is related to data warehouse (non-RDBMS) applications and for item-sets like images, event-logging data (clickstreams, sales events, real-time sensor data, etc.), documents or graph data.

Log file records all the events along with their timestamp when a transaction happens or any operation done. Log events can be an error/warning or a normal event like a click on a web page. But this constitutes a large volume of data and was expensive to store in earlier days. NoSQL system enabled the organizations to store this data and analyze at low cost.

Using NoSQL databases, application developers can handle both relational data structures and the in-memory data structures of the application. Developers can do this without having to convert in-memory structures to relational structures, using NoSQL. RDBMS is more suited for ERP kind of applications but not suited for Facebook, Twitter kind of data.

NoSQL databases like key-value, document, and column-family databases are forms of aggregate-oriented database. Aggregate-oriented databases can be easily stored over clusters, on any machine and can be retrieved with all the related data. An important aspect of document database is that the entire contents of document can be queried similar to rows in your SQL system. Reports can be made combining the traditional data as well as NoSQL data. For example, a single query can extract all the authors of presentations made on a particular topic (content search). Using this result the other details like skill set, organization where he works, designation, contact number, etc. about the authors can be queried from RDBMS.

3.6.3 Analyzing Big Data with a Shared Nothing Architecture

In the distributed computing architecture, there are two ways of resource sharing possible or share nothing. The RAM can be shared or disk can be shared (by CPUs); or no resources shared. The three of them can be considered as *shared RAM*, *shared disk*, and *shared-nothing*. Each of these architectures works with different types of data to solve big data problems. In shared RAM, many CPUs access a single shared RAM over a high-speed bus. This system is ideal for large computation and also for graph stores. For graph traversals to be fast, the entire graph should be in main memory. The shared disk system, processors have independent RAM but shares disk space using a storage area network (SAN). Big data uses commodity machines which shares nothing (shares no resources).

Out of all the architectural data patterns, only two (key-value store and document store) are cache-friendly. BigTable stores can scale well and so use shared-nothing architecture. Their row-column identifiers are similar to key-value stores. Row stores and graph stores are not cache-friendly as they cannot be referenced by a short key and so it can be stored in the cache.

3.6.4 Choosing Distribution Models

NoSQL database makes distribution of data easier, since it has to move only aggregate data and not all the related data that is used in aggregation. There are two styles of distributing data: Sharding and replication. A system may use either or both techniques. Like Riak database shards the data and also replicates it.

1. **Sharding:** Horizontal partitioning of a large database leads to partitioning of rows of the database. Each partition forms part of a shard, meaning small part of the whole. Each part (shard) can be located on a separate database server or any physical location.
2. **Replication:** Replication copies entire data across multiple servers. So the data is replicated and available in multiple places. Replication comes in two forms: master-slave and peer-to-peer.
 - *Master-slave replication:* One node has the authoritative copy that handles writes. Slaves synchronize with the master and handle reads.
 - *Peer-to-peer replication:* This allows writes to any node; the nodes coordinate between themselves to synchronize their copies of the data.

Master-slave replication reduces conflicts during updates. In master-slave distribution model, a single master node manages the cluster nodes. If master node crashes Single point of failure (SPOF) occurs. But this can be avoided either by using RAID drives or by using a standby master that is continually updated from the original master node. But the real concern is that the standby master has to take over if the master fails for high availability. The other concern with the standby master is that it is difficult to test the standby master without putting the cluster into risk.

Peer-to-peer replication does not load all writes onto a single server and so avoids single point of failure. In peer-to-peer model, each node in the cluster takes the responsibility of the master, so even if

one node fails other nodes can continue to function. But all the nodes have to be kept up-to-date with the cluster status which increases the communication overhead.

The right distribution model is chosen based on the business requirement. For example, for batch processing jobs, master-slave is chosen and for high availability jobs, peer-to-peer distribution model is chosen. Hadoop's initial version has master-slave architecture where the NameNode acting as master manages the status of the cluster. Later versions removed SPOF. HBase has a master-slave model, while Cassandra has a peer-to-peer model.

3.6.5 Four Ways that NoSQL System Handles Big Data Problems

Every business needs to find the technology trends that have impact on its revenue. Modern business not only needs data warehouse but also requires web/mobile application generated data and social networking data to understand their customer's need. NoSQL systems help to analyze such data. IT executives select the right NoSQL systems and set up and configure them.

3.6.5.1 Manage Data Better by Moving Queries to the Data

NoSQL system uses commodity hardware to store fragmented data on their shared-nothing architecture except for graph databases which require specialized processors. NoSQL databases improve performances drastically over RDBMS systems by moving the query to each node for processing and not transfer the huge data to a single processor. Traditional RDBMS are not capable of distributing the query and aggregate the results without paying a price for it. In traditional systems, tables have to be extracted, serialized, using network interface transmitted over network, and assembled on the server once again. It then executes the query. In NoSQL system, only query and results are moved over network.

3.6.5.2 Using Consistent Hashing Data on a Cluster

A server in a distributed system is identified by a key to store or retrieve data. The most challenging problem here is when servers become unreachable through network partitions or when server fails. Suppose there are “ n ” servers to store or retrieve a value. Server is identified by hashing the value's key modulo s . But when server fails, the server no longer fills the hash space. The only option is to invalidate the cache on all servers, renumber them, and start once again. This solution is not feasible if the system has hundreds of servers and one or the other server fails.

Consistent hashing consistently maps objects to the same server, and this can be explained as follows: When a server is added, it takes its good share of objects from all the other servers. Also, when server is removed, its objects are shared between the remaining servers.

Using a hash ring technique, big data is evenly distributed over many servers with a randomly generated key. Hash rings consistently determine how to assign a subset of data to a specific node. In consistent hashing, the servers and the keys are hashed, and it is by this hash that they are referred. The reason to do this is to map the server to an interval, which contains a number of object hashes.

If the server is removed then its interval is taken over by a server with an adjacent interval. All the other servers remain unchanged.

For example, Fig. 3.5 demonstrates this: Both objects and servers are mapped to points on a circle using a hash function. An object is assigned to the closest server going clockwise in a circle. Objects *i*, *j* and *k* are mapped to server *A*. Objects *l* and *m* are mapped to server *B*. When a new server is added, the only objects that are reassigned are those closest to the new server going clockwise in that circle. Here when the new server is added only objects *i* and *j* move to the new server *C*. Objects do not move between previously existing servers.

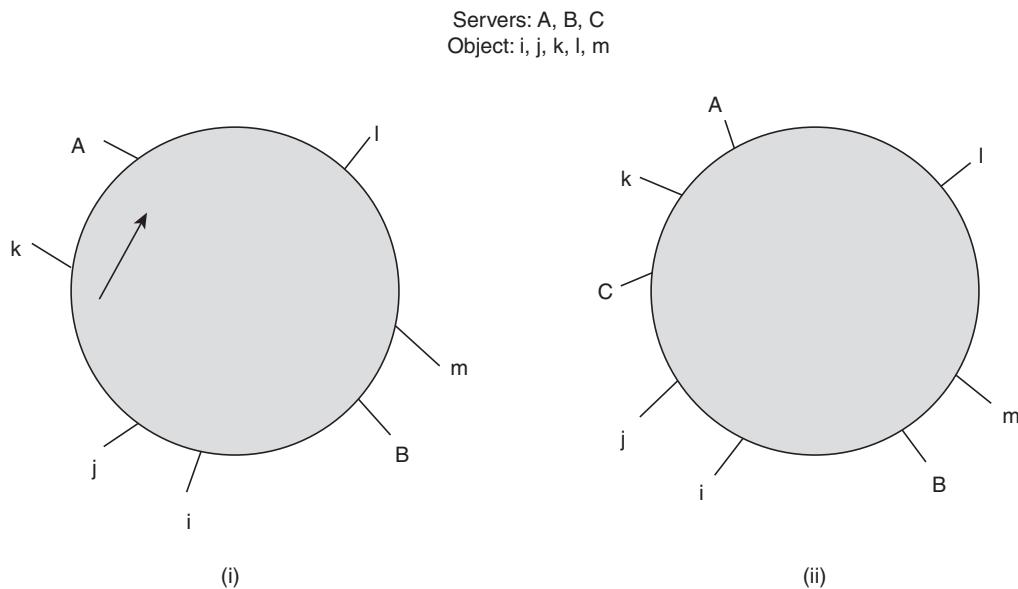


Figure 3.5 Consistent hashing.

3.6.5.3 Using Replication to Scale Reads

Replication improves read performance and database server availability. Replication can be used as a scale-out solution where you want to split up database queries across multiple database servers. Replication works by distributing the load of one master to one or more slaves. This works best in an environment where there are high number of reads and low number of writes or updates. Most users browse the website for reading articles, posts or view products. Writes occur only when making a purchase (during session management) or when adding a comment or sending message to a forum.

Replication distributes the reads over the replication slaves, and when a write is required, the servers communicate with the replication master.

When write happens, master will record this change-data to its binary log. The slave copies the change-data from master's binlog to its relay log. Then the slave applies the change-data recorded in its relay log to its own data.

This gives way to two problems. One is replication-lag and another is missing or inconsistent data. Asynchronous replication will introduce lag between reads and writes.

Most of the times after a write operation, there is a read of that same dataset. If a client does a write and then immediately a read from that same node, there will be read consistency. But in case a read occurs from a replication slave before the update happens, it will be an inconsistent read.

3.6.5.4 Letting the Database Distribute Queries Evenly to DataNodes

The most important strategy of NoSQL data store is moving query to the database server and not vice versa. Every node in the cluster in shared-nothing architecture is identical; all the nodes are peers. Data is distributed evenly across all the nodes in a cluster using extremely random hash function and so there are no bottlenecks.

In ideal scale-out architecture “shared-nothing” concept is used. Since no resource is shared, there is no bottleneck and all the nodes in this architecture act as peers. Data is evenly distributed among peers through a process called sharding. Sharding can be manual or on auto mode. In auto mode, data is automatically hashed for even distribution in the cluster. An algorithm is written to route the request to the particular node from which data can be accessed. In scale-out architecture, if a node fails it can continue processing with the rest of the cluster. Auto-sharding is advantageous over manual during node failures for replicating and moving the data. Rebalancing is a big overhead in manually sharded architecture when datasets become very large.

3.6.5.5 Querying Large Datasets

In a document database key access pattern accesses data through a single key and navigates to another document through a related key. No JOIN is required. For example, Employee record is accessed with EmployeeId (Eno) and then it navigates to find the Department using DepartmentId (Dno) as shown in Fig. 3.6.

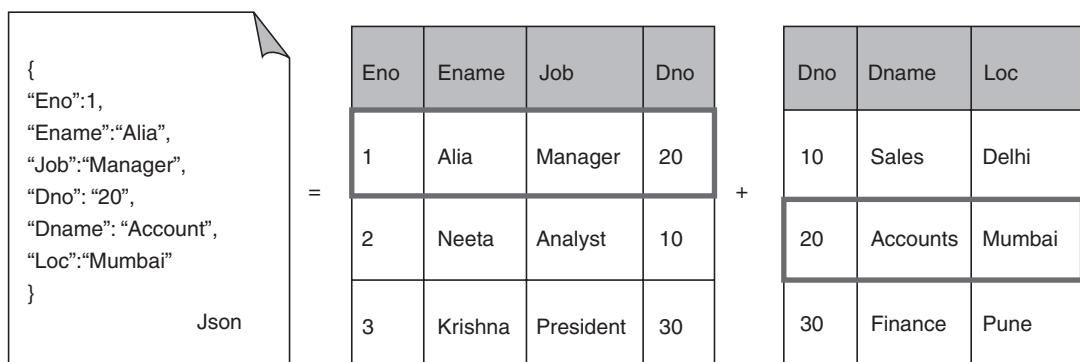


Figure 3.6 Document store.

Summary

- Velocity, variety, volume, and agility are the business drivers for NoSQL.
- SQL and NoSQL are complementary.
- NoSQL is not a general-purpose datastore which provides ACID properties. It is eventually consistent model.
- NoSQL is schema-less.
- It is free of JOIN operations.
- NoSQL databases use distributed architecture and work on multiple processors to give high performance.
- NoSQL databases are horizontally scalable based on the data loads.
- NoSQL databases manage data better by moving queries to the data and not data to queries.
- NoSQL databases use consistent hashing data on a cluster to evenly distribute data in a cluster.
- Similar to data, the queries are also evenly distributed in the cluster.
- Four NoSQL architectural patterns discussed are key-value store, column-family store, document store and graph store.
- Four ways used by NoSQL for handling big data problems are discussed.

Review Questions

1. What are the business drivers for NoSQL?
2. Explain CAP. How is Cap different from ACID property in databases?
3. Explain why NoSQL is schema-less.
4. When it comes to big data how NoSQL scores over RDBMS?
5. Discuss the four different architectural patterns of NoSQL.
6. Mention some uses of key-value store and also state its weakness.
7. Specify few business use-cases for NoSQL.
8. What are the variations of NoSQL architectural patterns?
9. What is big data NoSQL solution?
10. Discuss a few types of big data problems.
11. Give an example and explain shared and shared-nothing architecture.
12. What is the mechanism used by NoSQL to evenly distribute data in a cluster?
13. What is the difference between replication and sharding?
14. Explain the four ways by which big data problems are handled by NoSQL.

Laboratory Exercise

I. MongoDB installation steps

Installing MongoDB on Ubuntu 14.04

1. Create new user and group as follows:

(a) To create group name mongo:

```
$ sudoaddgroup mongo
```

(b) To create user and add it in group named mongo:

```
$ sudoadduser --ingroup mongo muser
```

(c) To give sudo rights to muser:

```
$ sudoaddusermusersudo
```

(d) To switch user to muser:

```
$ sumuser
```

2. Import the public key used by the package management system:

Issue the following command to import the MongoDB public GPG Key:

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
```

3. Create a list file for MongoDB:

Create the /etc/apt/sources.list.d/mongodb-org-3.0.list list file using the command

```
$ echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list
```

4. Reload local package database:

Issue the following command to reload the local package database:

```
$ sudo apt-get update
```

5. Install the latest stable version of MongoDB:

Issue the following command:

```
$ sudo apt-get install -y mongodb-org
```

6. Run MongoDB:

The MongoDB instance stores its data files in /var/lib/mongodb and its log files in /var/log/mongodb by default, and runs using the mongodb user account. You can specify alternate log and data file directories in /etc/mongod.conf. See systemLog.path and storage.dbPath for additional information.

If you change the user that runs the MongoDB process, you **must** modify the access control rights to the /var/lib/mongodb and /var/log/mongodb directories to give this user access to these directories.

7. Start MongoDB: Issue the following command to start mongod:

```
$ sudo service mongod start
```

8. Stop MongoDB: As needed, we can stop the mongod process by issuing the following command:

```
$ sudo service mongod stop
```

9. Restart MongoDB: Issue the following command to restart mongod:

```
$ sudo service mongod restart
```

II. Working with MongoDB

(a) Once the installation of MongoDB is completed type the following command to run it:

```
$ mongo
```

O/P:

```
MongoDB shell version: 2.6.11
```

```
connecting to: test
```

```
>
```

(b) To check the different commands available in MongoDB type the command as shown below:

```
>db.help()
```

O/P:

(c) DB methods:

```
db.adminCommand(nameOrDocument) – switches to ‘admin’ db, and runs command [ just calls db.runCommand(...)]
```

```
db.auth(username, password)
```

```
db.cloneDatabase(fromhost)
```

```
db.commandHelp(name) – returns the help for the command
```

```
db.copyDatabase(fromdb, todb, fromhost)
```

```
db.createCollection(name, { size : ..., capped : ..., max : ... } )
```

```
db.createUser(userDocument)
```

```
db.currentOp() – displays currently executing operations in the db
```

```
db.dropDatabase()
```

```
db.eval(func, args) – runs code server-side
```

```
db.fsyncLock() – flushes data to disk and lock server for backups
```

```
db.fsyncUnlock() – unlocks server following a db.fsyncLock()
```

```
db.getCollection(cname) same as db[‘cname’] or db.cname
```

```
db.getCollectionInfos()
```

```
db.getCollectionNames()
```

```
db.getLastError() – just returns the err msg string
```

db.getLastErrorObj() – returns full status object
db.getMongo() – gets the server connection object
db.getMongo().setSlaveOk() – allows queries on a replication slave server
db.getName()
db.getPrevError()
db.getProfilingLevel() – deprecated
db.getProfilingStatus() – returns if profiling is on and slow threshold
db.getReplicationInfo()
db.getSiblingDB(name) – gets the db at the same server as this one
db.getWriteConcern() – returns the write concern used for any operations on this db, inherited from server object if set
db.hostInfo() – gets details about the server's host
db.isMaster() – checks replica primary status
db.killOp(opid) – kills the current operation in the db
db.listCommands() – lists all the db commands
db.loadServerScripts() – loads all the scripts in db.system.js
db.logout()
db.printCollectionStats()
db.printReplicationInfo()
db.printShardingStatus()
db.printSlaveReplicationInfo()
db.dropUser(username)
db.repairDatabase()
db.resetError()
db.runCommand(cmdObj) – runs a database command. If cmdObj is a string, turns it into { cmdObj : 1 }
db.serverStatus()
db.setProfilingLevel(level,<slowms>) 0=off 1=slow 2=all
db.setWriteConcern(<write concern doc>) – sets the write concern for writes to the db
db.unsetWriteConcern(<write concern doc>) – unsets the write concern for writes to the db
db.setVerboseShell(flag) – displays extra information in shell output
db.shutdownServer()
db.stats()
db.version() – current version of the server
>

- (d) To check the current statistic of database type the command as follows:

```
>db.stats()  
O/P:
```

```
{  
  "db" : "test",  
  "collections" : 0,  
  "objects" : 0,  
  "avgObjSize" : 0,  
  "dataSize" : 0,  
  "storageSize" : 0,  
  "numExtents" : 0,  
  "indexes" : 0,  
  "indexSize" : 0,  
  "fileSize" : 0,  
  "dataFileVersion" : {  
  },  
  "ok" : 1  
}  
>
```

Note: In the output we can see that everything is “0”. This is because we haven’t yet created any collection.

Some considerations while designing schema in MongoDB:

1. Design your schema as per the user’s requirements.
2. Combine the objects into one document if you are going to use them together. Otherwise separate them.
3. Duplicate the data (but in limit) because disk space is cheap as compare to compute time.
4. Optimize your schema for the most frequent use cases.
5. Do join while write and not on read.
6. Do complex aggregation in the schema.

For example: Let us say that a client needs a database design for his blog and see the differences between RDBMS and MongoDB schema. Website has the following requirements:

1. Every post can have one or more tag.
2. Every post has the unique title, description and url.
3. Every post has the name of its publisher and total number of likes.
4. On each post there can be zero or more comments.

In RDBMS schema design for above requirements will have minimum three tables.

Comment(comment_id,post_id,by_user,date_time,likes,messages)
post(id,title,description,like,url,post_by)

```
tag_list(id,post_id,tag)
```

While in MongoDB schema design will have one collection (i.e., Post) and has the following structure:

```
{
    _id: POST_ID
    title: TITLE_OF_POST,
    description: POST_DESCRIPTION,
    by: POST_BY,
    url: URL_OF_POST,
    tags: [TAG1, TAG2, TAG3],
    likes: TOTAL_LIKES,
    comments: [
        {
            user: COMMENT_BY,
            message: TEXT,
            dateCreated: DATE_TIME,
            like: LIKES
        },
        {
            user: COMMENT_BY,
            message: TEXT,
            dateCreated: DATE_TIME,
            like: LIKES
        }
    ]
}
```

The table given below shows the basic terminology in MongoDB in relation with RDBMS:

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)

Use command is used to create a new database or open an existing one.

1. **The use command:** In MongoDB use command is used to create the new database. The command creates new database, if it doesnot exist; otherwise it will return the existing database.

Syntax:

```
use DATABASE_NAME
```

Example: If you want to create a database with name <mydatabase1>, then use DATABASE statement as follows:

```
> use mydatabase1  
switched to db mydatabase1
```

To check your currently selected database type “db”.

```
>db  
mydatabase1
```

To check your database list type the following command:

```
> show dbs  
admin (empty)  
local 0.078GB  
test (empty)
```

Our created database (mydatabase1) is not present in list. To display database we need to insert atleast one document into it.

```
>db.students.insert({“name”:”Sushil”, “place”：“Mumbai”})  
WriteResult({ “nInserted” : 1 })  
> show dbs  
admin (empty)  
local 0.078GB  
mydatabase1 0.078GB  
test (empty)
```

2. **The dropDatabase() Method:** In MongoDB, db.dropDatabase() command is used to drop an existing database.

Syntax:

```
db.dropDatabase()
```

Basically it will delete the selected database, but if you have not selected any database, then it will delete the default test database.

For example, to do so, first check the list of available databases by typing the command:

```
> show dbs  
admin (empty)  
local 0.078GB  
mydatabase1 0.078GB  
test (empty)
```

Suppose you want to delete newly created database (i.e. mydatabase1) then

```
> use mydatabase1
switched to db mydatabase1
>db.dropDatabase()
{ "dropped" : "mydatabase1", "ok" : 1 }
```

Now just check the list of databases. You will find that the database name mydatabase1 is not present in the list. This is because it got deleted.

```
> show dbs
admin (empty)
local 0.078GB
test (empty)
```

- 3. The `createCollection()` Method:** In MongoDB, `db.createCollection(name, options)` is used to create collection.

Syntax:

```
db.createCollection(name,options)
```

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Following is the list of options you can use:

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
autoIndexID	Boolean	(Optional) If true, automatically creates index on _id field.s Default value is false.
Size	number	(Optional) Specifies a maximum size in bytes for a capped collection. If capped is true, then you need to specify this field also.
Max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

For example, basic syntax of **createCollection()** method without options is as follows:

```
> use test1
switched to db test1
>db.createCollection("mycollection1")
{ "ok" : 1 }
```

-To check the created collection:

```
> show collections
mycollection1
system.indexes
```

Note: In MongoDB you do not need to create collection. It creates collection automatically when you insert some document. For example,

```
>db.students.insert({ "name" : "sushil" })
WriteResult({ "nInserted" : 1 })
> show collections
mycollection1
students
system.indexes
```

4. **The `drop()` Method:** MongoDB's **db.collection.drop()** is used to drop a collection from the database.

Syntax:

```
db.COLLECTION_NAME.drop()
```

For example,

-First check the available collections:

```
> show collections
mycollection1
students
system.indexes
```

-Delete the collection named mycollection1:

```
>db.mycollection1.drop()
```

```
true
```

-Again check the list of collection:

```
> show collections
students
system.indexes
>
```

5. The supported datatype in MongoDB are as follows:

- **String:** This is the most commonly used datatype to store the data. String in mongodb must be UTF-8 valid.
- **Integer:** This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean:** This type is used to store a Boolean (true/ false) value.
- **Double:** It is used to store floating point values.
- **Min/Max keys:** This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays:** This type is used to store arrays or list or multiple values into one key.
- **Timestamp:** ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object:** This datatype is used for embedded documents.
- **Null:** This type is used to store a Null value.
- **Symbol:** Its use is identically to a string. However, it is generally reserved for languages that use a specific symbol type.
- **Date:** This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID:** This datatype is used to store the document's ID.
- **Binary data:** This datatype is used to store binary data.
- **Code:** This datatype is used to store javascript code into document.
- **Regular expression:** This datatype is used to store regular expression.
- **RDBMS where Clause Equivalents in MongoDB:** To query the document on the basis of some condition, you can use following operations:

<i>Operation</i>	<i>Syntax</i>	<i>Example</i>	<i>RDBMS Equivalent</i>
Equality	{<key>:<value>}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{\$lt:<value>}}	db.mycol.find({"likes":{\$lt:50}}).pretty()	where likes <50

(Continued)

(Continued)

<i>Operation</i>	<i>Syntax</i>	<i>Example</i>	<i>RDBMS Equivalent</i>
Less Than	{<key>:{\$lte:<value>}}	db.mycol. find({"likes":{\$lte:50}}).pretty()	where likes <= 50
Equals			
Greater Than	{<key>:{\$gt:<value>}}	db.mycol. find({"likes":{\$gt:50}}).pretty()	where likes >50
Greater Than	{<key>:{\$gte:<value>}}	db.mycol. find({"likes":{\$gte:50}}).pretty()	where likes >= 50
Equals			
Not Equals	{<key>:{\$ne:<value>}}	db.mycol. find({"likes":{\$ne:50}}).pretty()	where likes != 50

4

MapReduce

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Learn the need for MapReduce.
- Understand Map task, Reducer task and Combiner task.
- Learn various MapReduce functions.
- Learn MapReduce algorithm for relational algebra operations.
- Learn MapReduce algorithm for matrix multiplication.

4.1

MapReduce and The New Software Stack

Businesses and governments need to analyze and process a tremendous amount of data in a very short period of time. The processing is to be done on a large amount of data, which will take a huge amount of time if done on a single machine. So the idea is to divide the data into smaller chunks and send to a cluster of machines, where they can be processed simultaneously and then the results can be combined.

Huge increase in data generated from social network and other blogging sites, for example, “Friends” on social networking sites, has led to increase in the volume of graphic data with millions of nodes and edges. This led to the creation of a new software stack. This new software stack provides parallelism by using several commodity hardware connected by Ethernet or switches. Hadoop is a framework for large-scale distributed batch processing. Hadoop can be deployed on a single machine if the data can be handled by the machine, but it is mainly designed to efficiently distribute a large amount of data for processing across a set of machines. Hadoop includes a distributed file system (DFS) that splits the input data and sends these portions of the original data to several machines in the defined cluster to hold. Main focus of this new software stack is MapReduce, a high-level programming system. This helps in doing the computation of the problem in parallel using all the connected machines so that the output, results are obtained in an efficient manner. DFS also provides data replication up to three times to avoid data loss in case of media failures.

Figure 4.1 shows the role of client machines, Master and Slave Nodes in Hadoop deployment. The MasterNode stores the huge data Hadoop Distributed File System (HDFS) and runs parallel computations on all that data (MapReduce).

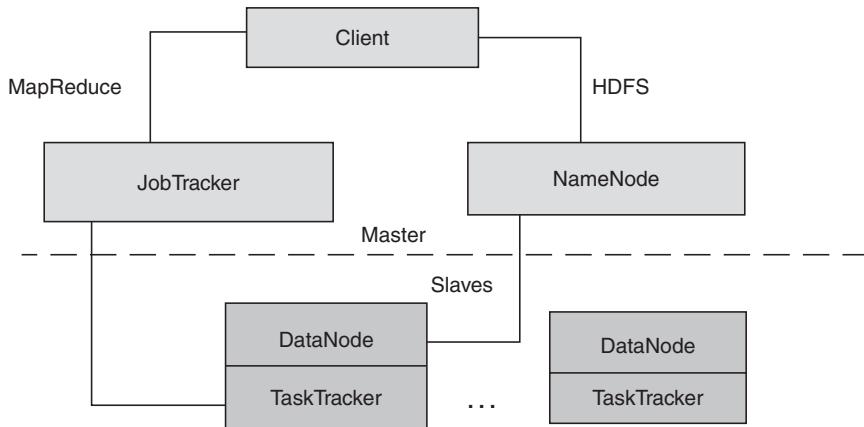


Figure 4.1 Hadoop high-level architecture.

1. The NameNode coordinates and monitors the data storage function (HDFS), while the JobTracker coordinates the parallel processing of data using MapReduce.
2. SlaveNode does the actual work of storing the data and running the computations. Master-Nodes give instructions to their SlaveNodes. Each slave runs both a DataNode and a TaskTracker daemon that communicate with their respective MasterNodes.
3. The DataNode is a slave to the NameNode.
4. The TaskTracker is a slave to the JobTracker.

4.1.1 Distributed File Systems

Most scientific applications in the past, required to do parallel processing for fast computing, used special-purpose computers. Web services enabled the use of commodity nodes (having RAM, CPU and hard disk) to execute the chosen services independently on the nodes and this reduced the cost of using special-purpose machines for parallel computing. In recent times, the new parallel-computing architecture called *cluster computing* is in use. Compute nodes typically in the range of 8–64 are stored in racks and are connected with each other by Ethernet or switch to the network. Failure at the node level (disk failure) and at the rack level (network failure) is taken care of by replicating data in secondary nodes. All the tasks are completed independently and so if any task fails, it can be re-started without affecting the other tasks.

File system stores data permanently. The system has logical drives and is layered on top of physical storage medium. It is addressed by a file name under a directory that supports hierarchical nesting. Access to the file is through file path consisting of drive, directory(s) and filename.

DFS supports access to files that are stored on remote servers. It also offers support for replication and local caching. Concurrent access to files read/write has to be taken care of using locking conditions. Different types of implementations are available based on the complexity of applications.

4.1.1.1 Google File System

Google had to store a massive amount of data. It needs a good DFS with cheap commodity computers to reduce cost. These commodity computers are unreliable, hence redundant storage is required to manage failures. Most of the files in Google file system (GFS) are written only once and sometimes appended. But it needs to allow large streaming reads and so high-sustained throughput is required over low latency. File sizes are typically in gigabytes and are stored as chunks of 64 MB each. Each of these chunks is replicated thrice to avoid information loss due to the failure of the commodity hardware. These chunks are centrally managed through a single master that stores the metadata information about the chunks. Metadata stored on the master has file and chunk namespaces, namely, mapping of file to chunks and location of the replicas of each chunk. Since Google users do a lot of streaming read of large data sets, caching has no importance or benefit. What if the master fails? Master is replicated in shadow master. Also the master involvement is reduced by not moving data through it; metadata from master is cached at clients. Master chooses one of the replicas of chunk as primary and delegates the authority for taking care of the data mutations.

4.1.1.2 Hadoop Distributed File System

HDFS is very similar to GFS. Here, the master is called NameNode and shadow master is called Secondary NameNode. Chunks are called blocks and chunk server is called DataNode. DataNode stores and retrieves blocks, and also reports the list of blocks it is storing to NameNode. Unlike GFS, only single-writers per file is allowed and no append record operation is possible. Since HDFS is an open-source, interface, libraries for different file systems are provided.

4.1.2 Physical Organization of Compute Nodes

Hadoop runs best on Linux machines. Hadoop is installed in client machines with all the cluster settings. The client machine loads data and MapReduce program into the cluster, and then retrieves or views the results once the program is executed. For smaller clusters, where the number of nodes is less than 40, a single physical server can host both JobTracker and NameNode. For medium and large clusters, both of them can be in different physical servers. The “server virtualization” or “hypervisor layer” adds to overhead and impedes the Hadoop performance. Hadoop does work in a virtual machine. Cluster (with a few nodes) can be up and running in VMware Workstation on a laptop machine.

4.1.2.1 Case Study

What problem does Hadoop solve? Businesses and governments have a large amount of data that needs to be analyzed and processed very quickly. If this data is fragmented into small chunks and spread over many machines, all those machines process their portion of the data in parallel and the results are obtained extremely fast.

For example, a huge data file containing feedback mails is sent to the customer service department. The objective is to find the number of times goods were returned and refund requested. This will help the business to find the performance of the vendor or the supplier.

It is a simple word count exercise. The client will load the data into the cluster (Feedback.txt), submit a job describing how to analyze that data (word count), the cluster will store the results in a new file (Returned.txt), and the client will read the results file.

The client is going to break the data file into smaller “Blocks”, and place those blocks on different machines throughout the cluster. Every block of data is on multiple machines at once to avoid data loss. So each block will be replicated in the cluster as it is loaded. The standard setting for Hadoop is to have (three) copies of each block in the cluster. This can be configured with the dfs.replication parameter in the file hdfs-site.xml.

The client breaks Feedback.txt into three blocks. For each block, the client consults the NameNode and receives a list of three DataNodes that should have a copy of this block. The client then writes the block directly to the DataNode. The receiving DataNode replicates the block to other DataNodes, and the cycle repeats for the remaining blocks. Two of these DataNodes, where the data is replicated, are in the same rack and the third one is in another rack in the network topology to prevent loss due to network failure. The NameNode as it is seen is not in the data path. The NameNode only provides the metadata, that is, the map of where data is and where data should be in the cluster (such as IP address, port number, Host names and rack numbers).

The client will initiate TCP to DataNode 1 and sends DataNode 1 the location details of the other two DataNodes. DataNode 1 will initiate TCP to DataNode 2, handshake and also provide DataNode 2 information about DataNode 3. DataNode 2 ACKs and will initiate TCP to DataNode 3, handshake and provide DataNode 3 information about the client which DataNode 3 ACKs.

On successful completion of the three replications, “Block Received” report is sent to the NameNode. “Success” message is also sent to the Client to close down the TCP sessions. The Client informs the NameNode that the block was successfully written. The NameNode updates its metadata info with the node locations of Block A in Feedback.txt. The Client is ready to start the process once again for the next block of data.

The above process shows that Hadoop uses a lot of network bandwidth and storage.

The NameNode not only holds all the file system metadata for the cluster, but also oversees the health of DataNodes and coordinates access to data. The NameNode acts as the central controller of HDFS. DataNodes send heartbeats to the NameNode every 3 seconds via a TCP handshake using the same port number defined for the NameNode daemon. Every 10th heartbeat is a Block Report, where the DataNode tells the NameNode about all the blocks it has.

1. DataNode sends “hearts beat” or “block report”.
2. NameNode ACK.
3. DataNode acknowledges the ACK.

Every hour, by default the Secondary NameNode connects to the NameNode and copies the in-memory metadata information contained in the NameNode and files that used to store metadata (both

may and may not be in sync). The Secondary NameNode combines this information in a fresh set of files and delivers them back to the NameNode, while keeping a copy for itself.

4.1.2.2 Receiving the Output

When a Client wants to retrieve the output of a job, it again communicates to the NameNode and asks for the block locations of the results file. The NameNode in turn provides the Client a unique list of three DataNodes for each block. Client chooses the first DataNode by default in each list. Blocks are read sequentially. Subsequent blocks are read only after the previous block is read completely.

DataNode requests the NameNode for location of block data. The NameNode will first check for DataNode in the same rack. If it is present, the NameNode provides the in-rack location from which to retrieve the data. This prevents the flow from traversing two more switches and congested links to find the data (in another rack). With the data retrieved quicker in-rack, data processing can begin sooner and the job completes that much faster.

4.1.2.3 Map Process

MapReduce is the parallel processing framework along with Hadoop, named after two important processes: Map and Reduce.

Map process runs computation on their local block of data. The MapReduce program needs to count the number of occurrences of the word “Refund” in the data blocks of Feedback.txt. Following are the steps to do this:

1. Client machine submits the MapReduce job to the JobTracker, asking “How many times does Refund occur in Feedback.txt?”
2. The JobTracker finds from the NameNode which DataNodes have blocks of Feedback.txt.
3. The JobTracker then provides the TaskTracker running on those nodes with the required Java code to execute the Map computation on their local data.
4. The TaskTracker starts a Map task and monitors the tasks progress.
5. The TaskTracker provides heartbeats and task status back to the JobTracker.
6. As each Map task completes, each node stores the result of its local computation as “intermediate data” in temporary local storage.
7. This intermediate data is sent over the network to a node running a Reduce task for final computation.

Note: If the nodes with local data already have too many other tasks running and cannot accept anymore, then the JobTracker will consult the NameNode whose Rack Awareness knowledge can suggest other nodes in the same rack. In-rack switching ensures single hop and so high bandwidth.

4.1.2.4 The Reduce Process

The Reduce process needs to gather all intermediate data from the Map tasks to combine them and have a single result. Following are the steps:

1. The JobTracker starts a Reduce task on any one of the nodes in the cluster and instructs the Reduce task to go and grab the intermediate data from all of the completed Map tasks.
2. The Map tasks may respond to the Reducer almost simultaneously, resulting in a situation where you have a number of nodes sending TCP data to a single node, all at once. This traffic condition is often referred to as “In-cast” or “fan-in”.
3. The network switches have to manage the internal traffic and adequate to handle all in-cast conditions.
4. The Reducer, task after collecting all of the intermediate data from the Map tasks, starts the final computation phase. In this example, the final result is adding up the sum total occurrences of the word “Refund” and writing the result to a file called Results.txt.
5. The output from the job is a file called Results.txt that is written to HDFS following all of the processes (split file, pipeline replication, etc.).
6. Then the Client machine can read the Results.txt file from HDFS, and the job is said to be completed.

4.1.2.5 Points to Ponder

1. Abstract complexity of distributed and concurrent applications makes it harder and more expensive to scale-up (scale vertically) using this approach.
2. Additional resources need to be added on an existing node (CPU, RAM) to keep up with data growth (Moore’s Law).
3. Hadoop follows the Scale-Out approach:
 - More nodes (machines) can be added to an existing distributed application.
 - Software layer is designed for node addition or removal.
 - A set of nodes is bonded together as a single-distributed system.
 - Easy-to-scale down as well.

RDBMS products scale up but are expensive to scale for larger installations and also are difficult when storage reaches 100s of terabytes. Hadoop clusters can scale-out to 100s of machines and to petabytes of storage.

4. Hadoop co-locates processors and storage. Code is moved to data. Processors access underlying local storage and execute the code.
5. Hadoop abstracts many complexities in distributed and concurrent applications.

Defines small number of components and provides simple and well-defined interfaces of interactions between these components.

6. Frees developer from worrying about system level challenges such as race conditions, data starvation, processing pipelines, data partitioning, code distribution, etc.
7. Allows developers to focus on application development and business logic.

4.2 MapReduce

Google's programming model MapReduce works on sort/merge-based distributed computing. Google used MapReduce for their internal search/indexing but now MapReduce is extensively used by other organizations such as Yahoo, Amazon, IBM.

It uses functional programming language such as LISP that is naturally parallelizable across large cluster of workstations. The Hadoop system partitions the input data, schedules the program execution across several workstations, manages inter-process communications and also handles machine failures.

Most business organizations require analytics to be done on a large-scale data. MapReduce framework is used to solve these computational problems. MapReduce works on divide-and-conquer principle. Huge input data are split into smaller chunks of 64 MB that are processed by mappers in parallel. Execution of map is co-located with data chunk.

The framework then shuffles/sorts the results (intermediate data) of maps and sends them as input to reducers. Programmers have to implement mappers and reducers by extending the base classes provided by Hadoop to solve a specific problem.

Each of the Map tasks is given to one or more chunks from a DFS. 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. Let us assume mapper takes (k_1, v_1) as input in the form of (key, value) pair. Let (k_2, v_2) be the transformed key–value pair by mapper.

$$(k_1, v_1) \rightarrow \text{Map} \rightarrow (k_2, v_2) \rightarrow \text{Sort} \rightarrow (k_2, (v_2, v_2, \dots, v_2)) \rightarrow \text{Reduce} \rightarrow (k_3, v_3)$$

The key–value pairs from each Map task are collected by a master controller and sorted by key. It combines each unique key with all its values, that is $(k_2, (v_2, v_2, \dots, v_2))$. The key–value combinations are delivered to Reduces, so all key–value pairs with the same key wind up at the same Reduce task. The way of combination of values is determined by the code written by the programmer for the Reduce function. Reduce tasks work on one key at a time. The Reducers again translates them into another key–value pair (k_3, v_3) which is the result.

Mapper is the mandatory part of a Hadoop job and can produce zero or more key–value pairs (k_2, v_2) . Reducer is the optional part of a Hadoop job and can produce zero or more key–value pairs (k_3, v_3) . The driver program for initializing and controlling the MapReduce execution is also written by user.

4.2.1 The Map Tasks

The Map task consists of elements that can be a tuple, a line or a document. A chunk is the collection of elements. All inputs to Map task are key–value pairs. The Map function converts the elements to zero or more key–value pairs. Keys are not similar to the way they are defined in traditional databases. They are not unique. Several same key–value pairs from same element is possible.

The standard example used to explain the MapReduce function is to count the number of words in a document. So here the document is an element. The Map function has to read a document and break it into sequence of words. Each word is counted as one. Each word is different and so the key is not unique. Hence, the output of the Map function, which is a key–value pair, can be denoted as $(\text{word}_1, 1), (\text{word}_2, 1), \dots, (\text{word}_n, 1)$. Input can be a repository or collection of documents and output is the number of occurrences of each word and a single Map task can process all the documents in one or more chunks.

4.2.2 Grouping by Key

When all the Map tasks have been completed, 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 automatically, regardless of what the Map and Reduce tasks do. The master controller process knows the value of the iterator “ i ” or how many Reduce tasks have to be performed from the user’s input. Each key from Map task results is hashed and the key–value pair is saved in one of the “ i ” (0 to $i - 1$) local files. To perform the grouping by key, the master controller merges the files from the Map task that sends the result as input to Reduce task in the form of key–value pairs. For example, for each key, the input to the reduce task that handles the key say (word_1) is a pair of the form $(\text{word}_1, [\nu_1, \nu_2, \dots, \nu_n])$, where $(\text{word}_1, \nu_1), (\text{word}_1, \nu_2), \dots, (\text{word}_1, \nu_n)$ are the key–value pairs coming from all the Map tasks.

4.2.3 The Reduce Tasks

The argument to Reduce function is the key and a list of its associated values. Each Reduce function uses one or more reducers. The outputs of all Reduce tasks are merged into a single file. The result of Reduce function is a sequence of zero or more key–value pairs, one for each key. In the word count example, the Reduce function will add up all the values in the list. The output of Reduce task is a sequence of (word, ν) , where “ word ” is the key that appears at least once among all input documents and “ ν ” is the total number of times the “ word ” has appeared among all those input documents.

4.2.4 Combiners

When the Reduce function is both associative and commutative (e.g., sum, max, average, etc.), then some of the tasks of Reduce function are assigned to combiner. Instead of sending all the Mapper data to Reducers, some values are computed in the Map side itself by using combiners and then they are sent to the Reducer. This reduces the input–output operations between Mapper and Reducer. Combiner takes the mapper instances as input and combines the values with the same key to reduce the number

of keys (key space) that must be sorted. To optimize the Map process, if a particular word w appears k times among all the documents assigned to the process, then there will be k times (word, 1) key–value pairs as a result of Map execution, which can be grouped into a single pair (word, k) provided in the addition process (Reduce task); associative and commutative properties are satisfied. Figure 4.2 shows word count using MapReduce algorithm with all the intermediate values obtained in mapping, shuffling and reducing.

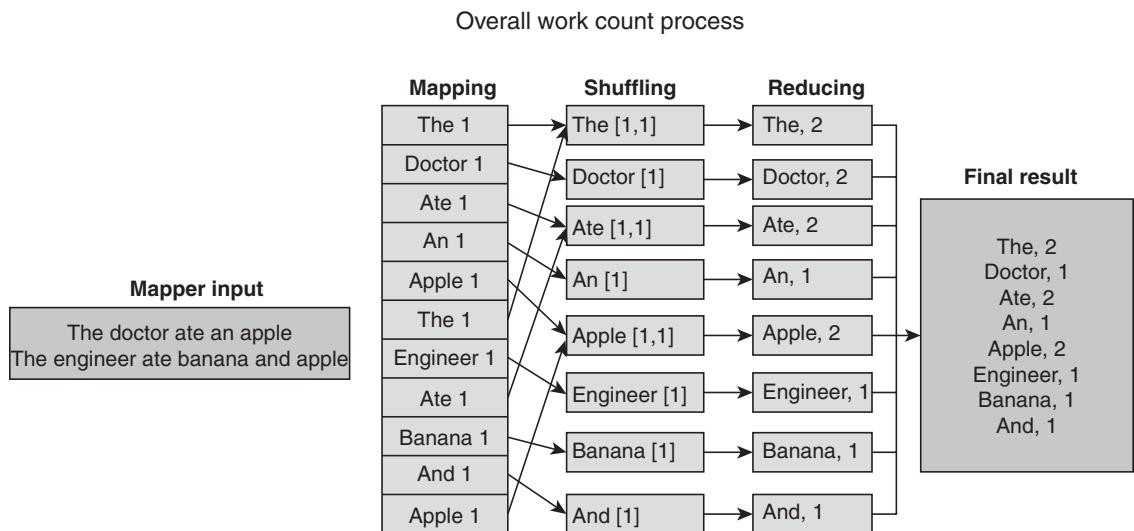


Figure 4.2 Word count using MapReduce algorithm.

Algorithm

Pseudo Code for Word Count Using MapReduce Algorithm

Word Count using MapReduce

→ count occurrences of each word input,

M = large corpus of text.

Mapper:

each word in $M \rightarrow (\text{“word”, } 1)$

Reducer:

for all $(\text{“word”, } v_i)$

$\rightarrow (\text{“word”, } \text{sum_}iv_i)$

“aggregate” in Hadoop.

Points to Note

1. For maximum parallelism, each reduce task is executed by one reducer on one compute node, that is a particular key along with its associated list of values. But there is an associated overhead when a task is created. To reduce the overhead, number of Reduce task is always kept less than the number of keys. In addition, number of keys is far more than the number of available compute nodes.
2. The length of the value list can be different for different keys. So the amount of time taken by the reducer can also vary. Hence, when keys are sent randomly to Reduce task, some averaging can happen on the total time taken. By keeping number of compute nodes less, one node with several shorter tasks and the other with longer task might complete almost in parallel.

4.2.5 Details of MapReduce Execution

4.2.5.1 Run-Time Coordination in MapReduce

MapReduce handles the distributed code execution on the cluster transparently once the user submits his “jar” file. MapReduce takes care of both scheduling and synchronization. MapReduce has to ensure that all the jobs submitted by all the users get fairly equal share of cluster’s execution. MapReduce implements scheduling optimization by speculative execution explained as follows: If a machine executes the task very slowly, the JobTracker assigns the additional instance of the same task to another node using a different TaskTracker. The speculative execution is set to “true” by default and can be disabled by setting `mapred.map.tasks.speculative.execution` job option false. Same can be done for Reduce task too. MapReduce execution needs to synchronize the map and reduce processes. The reduce phase cannot start till all the Map processes are completed. The intermediate data, that is, the key–value pair from Map processes have to be grouped by key. This is done by shuffle/sort involving all the nodes where Map tasks are executed and the nodes where Reduce tasks will be executed.

4.2.5.2 Responsibilities of MapReduce Framework

The framework takes care of scheduling, monitoring and rescheduling of failed tasks.

1. Provides overall coordination of execution.
2. Selects nodes for running mappers.
3. Starts and monitors mapper’s execution.
4. Sorts and shuffles output of mappers.
5. Chooses locations for reducer’s execution.
6. Delivers the output of mapper to reducer node.
7. Starts and monitors reducer’s execution.

4.2.5.3 MapReduce Execution Pipeline

Main components of MapReduce execution pipeline are as follows:

1. **Driver:** Driver is the main program that initializes a MapReduce job and gets back the status of job execution. For each job, it defines the configuration and specification of all its components (Mapper, Reducer, Combiner and Custom partitioner) including the input–output formats.

The driver, mapper and reducer are executed in different machines. A context object provides a mechanism for exchanging required information. Context coordination happens when a MapReduce job starts an appropriate phase (Driver, Mapper and Reducer). This is to say that output of one mapper is available to reducer and not to another mapper.

2. **Input data:** Input data can reside in HDFS or any other storage such as HBase. *InputFormat* defines how to read the input and define the split. Based on the split, *InputFormat* defines the number of map tasks in the mapping phase. The job Driver invokes the *InputFormat* directly to decide the number (*InputSplits*) and location of the map task execution. *InputSplit* defines the unit of work for the corresponding single map task. That is, each map task is given a single *InputSplit* to work on. Similarly, MapReducer job starts, in different locations, the required number (*InputSplit*) of mapper jobs. *RecordReader* class, defined by *InputFormat*, reads the data that is inside the mapper task. *RecordReader* converts the data into key–value pairs and delivers it to map method.
3. **Mapper:** For each map task, a new instance of mapper is instantiated. As said earlier, individual mappers do not communicate with each other. The *partition* of the key space produced by the mapper, that is every intermediate data from the mapper, is given as input to reducer. The *partitioner* determines the reduce node for the given key–value pair. All map values of the same key are reduced together and so all the map nodes must be aware of the reducer node.
4. **Shuffle and sort:** Shuffling is the process of moving map outputs to reducers. Shuffle/Sort is triggered when mapper completes its job. As soon as all the map tasks are completed, sort process groups the key–value pairs to form the list of values. The grouping is performed regardless of what Map and Reduce tasks do. Map script models the data into key–value pairs for the reducer to aggregate. All data from a partition goes to the same reducer. Shuffling basically means that pairs with same key are grouped together and passed to a single machine that will run the reducer script over them.
5. **Reducer:** Reducer executes the user-defined code. The reducers *reduce()* method receives a key along with an iterator over all the values associated with the key and produces the output key–value pairs. *RecordWriter* is used for storing data in a location specified by *OutputFormat*. Output can be from reducer or mapper, if reducer is not present.
6. **Optimizing MapReduce process by using Combiners (optional):** Combiners pushes some of the Reduce task to Map task. Combiner takes the mapper instances as input and combines the values with the same key to reduce the number of keys (key space) that must be sorted.

7. **Distributed cache:** Distributed cache is a resource used for sharing data globally by all nodes in the cluster. This can be a shared library that each task can access. The user's code for driver, map and reduce along with the configuring parameters can be packaged into a single jar file and placed in this cache.

4.2.5.4 Process Pipeline

1. Job driver uses *InputFormat* to partition a map's execution and initiate a JobClient.
2. JobClient communicates with JobTracker and submits the job for execution.
3. JobTracker creates one Map task for each split as well as a set of reducer tasks.
4. TaskTracker that is present on every node of the cluster controls the actual execution of Map job.
5. Once the TaskTracker starts the Map job, it periodically sends a *heartbeat* message to the JobTracker to communicate that it is alive and also to indicate that it is ready to accept a new job for execution.
6. JobTracker then uses a scheduler to allocate the task to the TaskTracker by using *heartbeat* return value.
7. Once the task is assigned to the TaskTracker, it copies the job jar file to TaskTracker's local file system. Along with other files needed for the execution of the task, it creates an instance of task runner (a child process).
8. The child process informs the parent (TaskTracker) about the task progress every few seconds till it completes the task.
9. When the last task of the job is complete, JobTracker receives a notification and it changes the status of the job as "completed".
10. By periodically polling the JobTracker, the JobClient recognizes the job status.

4.2.6 Coping with Node Failures

MapReduce jobs are submitted and tracked by JobTracker. There is only one JobTracker for a Hadoop cluster and JobTracker runs on its own JVM. All the slave nodes are configured with JobTracker node location. So if JobTracker fails, all the jobs running in its slave are halted. The whole MapReduce job is re-started.

If at a node the TaskTracker (actual job execution) run fails, then JobTracker monitors (periodically pings) all the TaskTrackers and so detects the failure. Now only all the tasks that are run in this node are re-started. Even if some of the tasks are completed in this node, they have to be re-done because the output destined for Reduce tasks still resides there and now they (the output) become unavailable. The JobTracker has to inform all the Reduce tasks that the input for them will be available from another location. If there is a failure at the Reduce node, then JobTracker sets it to "idle" and reschedules the Reduce tasks on another node.

4.3 Algorithms Using MapReduce

It is good to use MapReduce algorithm where there is data-intensive computation. DFS makes sense only when files are very large and are rarely updated in place.

MapReduce algorithm or a DFS is not well-suited for on-line retail sales where data involve responding to searches for products, recording sales, etc. and processes that involve relatively little calculation and that change the database. But MapReduce can be used to solve analytic queries on large amounts of data, for example, finding users with most similar buying patterns. Google mainly implemented MapReduce to execute very large matrix-vector multiplications that are needed for PageRank calculation. MapReduce algorithm is also effectively used for relational-algebra operations.

Generally a MapReduce program has three main components: driver, mapper and reducer. The driver initializes the job configuration, defines the mapper and the reducer and specifies the paths of the input and output files. The mapper and reducer classes extend classes from the package **org.apache.hadoop.mapreduce**. For Mapper and Reducer classes, the user can specify type information during the class declarant Id the generic type arguments are <KEYIN, VALUEIN, KEYOUT, VALUEOUT>.

Algorithm

map (InputKeyType inputKey, InputValueType inputValue):

```
// process the inputKey and inputValue and result will be intermediateKey , intermediateValue pair
Emit(intermediateKey, intermediateValue);
// map can emit more than one intermediate key–value pairs
```

The types of the input key and input value to the reduce method is the same as the types of the output key and output value of the map method.

Algorithm

reduce (IntermediateKeyType intermediateKey, Iterator values):

```
// all the values for a particular intermediateKey is first iterated and a user-defined operation is
// performed over the values.
// the number of reducers is specified by the user and they run in parallel
// outputValue will contain the value that is output for that outputKey
//Emit(outputKey, outputValue);
// reduce method can emit more than one output key–value pairs
```

4.3.1 Matrix-Vector Multiplication by MapReduce

Let A and B be the two matrices to be multiplied and the result be matrix C. Matrix A has dimensions L, M and matrix B has dimensions M, N . In the Map phase:

1. For each element (i,j) of I, emit $((i,k), A[i,j])$ for k in $1, \dots, N$.
2. For each element (j,k) of B, emit $((i,k), B[j,k])$ for i in $1, \dots, L$.

In the reduce phase, elit

$$\text{key} = (i,k)$$

$$\text{valuI} = \text{Sum}_j (A[i,j] * B[j,k])$$

One reducer is used per output cell

$$\text{Each reducer computes } \text{Sum}_j (A[i,j] * B[j,k])$$

The block diagram of MapReduce multiplication algorithm is shown in Fig. 4.3.

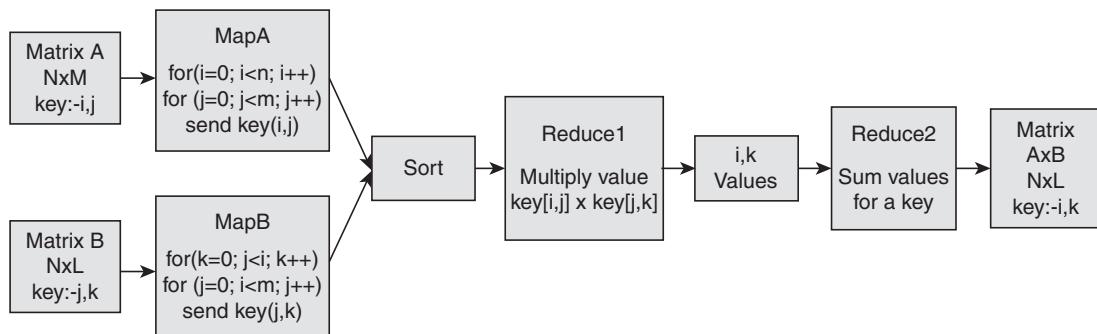


Figure 4.3 Matrix-multiplication by MapReduce.

Algorithm

Pseudo Code for Matrix-Vector Multiplication by MapReduce

```

map(key, value):
    for (i, j, aij) in value:
        emit(i, aij * v[j])
reduce(key, values):
    result = 0
    for value in values:
        result += value
    emit(key, result)
  
```

4.3.2 MapReduce and Relational Operators

MapReduce algorithms can be used for processing relational data:

1. Shuffle/Sort automatically handles group by sorting and partitioning in MapReduce.
2. The following operations are performed either in mapper or in reducer:
 - Selection
 - Projection
 - Union, intersection and difference
 - Natural join
 - Grouping and aggregation
3. Multiple strategies such as Reduce-side join, Map-side join and In-memory join (Striped variant, Memcached variant) are used for relational joins.

Multiple MapReduce jobs are required for complex operations. For example: Top 10 URLs in terms of average time spent.

4.3.3 Computing Selections by MapReduce

Selections really may not need both the Map and Reduce tasks. They can be done mostly in the map portion alone.

1. **The Map Function:** For each tuple t in R , test if it satisfies condition C . If so, produce the key–value pair (t, t) . That is, both the key and value are t .
2. **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.

Algorithm

Pseudo Code for Selection

```
map(key, value):
    for tuple in value:
        if tuple satisfies C:
            emit(tuple, tuple)
reduce(key, values):
    emit(key, key)
```

In other words:

```
class Mapper  
    method Map(rowkey key, tuple t)  
        if t satisfies the predicate  
            Emit(tuple t, null)
```

4.3.4 Computing Projections by MapReduce

Projection is also a simple operation in MapReduce. Here, the Reduce function is used to eliminate duplicates since projection may cause the same tuple to appear several times.

- The Map Function:** For each tuple t in R, construct a tuple ts by eliminating from t those components whose attributes are not in S. Output the key–value pair (ts, ts) .
- The Reduce Function:** For each key ts produced by any of the Map tasks, there will be one or more key–value pairs (ts, ts) . The Reduce function turns $[ts, ts, \dots, ts]$ into (ts, ts) , so it produces exactly one pair (ts, ts) for this key ts .

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.

Algorithm

Pseudo Code for Projection

```
map(key, value):  
    for tuple in value:  
        ts = tuple with only the components for the attributes in S  
        emit(ts, ts)  
  
reduce(key, values):  
    emit(key, key)
```

In other words:

```
class Mapper  
    method Map(rowkey key, tuple t)  
        tupleg = project(t) // extract required fields to tuple g  
        Emit(tuple g, null)  
  
class Reducer  
    method Reduce(tuple t, array n) // n is an array of nulls  
        Emit(tuple t, null)
```

4.3.5 Union, Intersection and Difference by MapReduce

4.3.5.1 Union

For union operation, both the relations R and S need to have the same schema.

Map tasks will be assigned chunks from either R or S relation. The Map tasks do not really do anything except pass their input tuple as key–value pairs to the Reduce tasks. Reducer is used to eliminate duplicates.

Mappers are fed by all tuples of two sets to be united. Reducer is used to eliminate duplicates.

- 1. The Map Function:** Turn each input tuple t into a key–value pair (t, t) .
- 2. The Reduce Function:** Associated with each key t there will be either one or two values. Produce output (t, t) in either case.

Algorithm

Pseudo Code for Union

```
map(key, value):
    for tuple in value:
        emit(tuple, tuple)

reduce(key, values):
    emit(key, key)
```

In other words:

```
class Mapper
    method Map(rowkey key, tuple t)
        Emit(tuple t, null)

class Reducer
    method Reduce(tuple t, array n) // n is an array of one or two nulls
        Emit(tuple t, null)
```

4.3.5.2 Intersection

Mappers are fed by all tuples of both R and S relations to be intersected. Reducer emits only tuples that occurred twice. It is possible only if both the sets contain this tuple because tuples include primary key and can occur in one set only once in each relation.

To compute the intersection, we can use the same the 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 do not want to produce a tuple for the intersection.

- 1. The Map function:** Turn each tuple t into a key–value pair (t, t) .
- 2. The Reduce function:** If key t has value list $[t, t]$, then produce (t, t) . Otherwise, produce nothing.

Algorithm

Pseudo Code for Intersection

```
map(key, value):
    for tuple in value:
        emit(tuple, tuple)

reduce(key, values):
    if values == [key, key]
        emit(key, key)
```

In other words:

```
class Mapper
    method Map(rowkey key, tuple t)
        Emit(tuple t, null)

class Reducer
    method Reduce(tuple t, array n) // n is an array of one or two nulls
        if n.size() = 2
            Emit(tuple t, null)
```

4.3.5.3 Difference

The difference $R - S$ a tuple t can appear in the output is if it is in relation R , but not in relation 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 . Using the relation as the value associated with the key t , the two functions are specified as follows:

- 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 and 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.

Algorithm

Pseudo Code for Difference

```
map(key, value):
    if key == R:
        for tuple in value:
            emit(tuple, R)
```

```

else:
    for tuple in value:
        emit(tuple, S)

reduce(key, values):
    if values == [R]
        emit(key, key)

```

In other words:

class Mapper

```

method Map(rowkey key, tuple t)
    Emit(tuple t, string tSetName) // t.SetName is there in "R" or "S"

```

class Reducer

```

method Reduce(tuple t, array n) // array n can be [""], ["S"], [R"S"], or ["S", "R"]
    if n.size() = 1 and n[1] = "R"
        Emit(tuple t, null)

```

4.3.6 Computing Natural Join by MapReduce

For doing Natural join, the relation R(A,B) with S(B,C), it is required to 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. Using the B-value of tuples from either relation as the key, the value will be the other component along with the name of the relation, so that 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.

Algorithm

Pseudo Code for Natural Join

```

map(key, value):
    if key == R:
        for (a, b) in value:
            emit(b, (R, a))

```

```

else:
    for (b, c) in value:
        emit(b, (S, c))

reduce(key, values):
    list_R = [a for (x, a) in values if x == R]
    list_S = [c for (x, c) in values if x == S]
    for a in list_R:
        for c in list_S:
            emit(key, (a, key, c))

```

4.3.7 Grouping and Aggregation by MapReduce

The following steps show grouping and aggregation by MapReduce:

1. Grouping and aggregation can be performed in one MapReduce job.
2. Mapper extracts from each tuple values to group by and aggregate and emits them.
3. Reducer receives values to be aggregated that are already grouped and calculates an aggregation function.

Let $R(A,B,C)$ be a relation to which apply the Irator $\gamma A, \theta(B)(R)$. Map will perform the grouping, while Reduce does the aggregation.

1. **The Map function:** For each tuple (a, b, c) produce the key–value pair (a, b) .
2. **The Reduce function:** Each key “ a ” represents a group. Apply the aggregation operator θ to the list $[b_1, b_2, \dots, b_n]$ 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 $x = b_1 + b_2 + \dots + b_n$, and if θ is MAX, then x is the largest of b_1, b_2, \dots, b_n .

Algorithm

Pseudo Code for Grouping and Aggregation

```

map(key, value):
    for (a, b, c) in value:
        emit(a, b)

reduce(key, values):
    emit(key, theta(values))

```

In other words:

```
class Mapper
    method Map(null, tuple [value GroupBy, IlueAggregateBy, value ...])
        Emit(value GroupBy, value AggregateBy)
class Reducer
    method RICe(value GroupBy, [v1, v2,...])
        Emit(vaIGroupBy, aggregate( [v1, v2,...] )) // aggregate() : sum(), max(),...
```

Note: If there are several grouping attributes, then the key is the list of the values of a tuple for all these attributes.

Mapper(M) $\rightarrow \{(key, value)\}$
 Sort ($\{(key, value)\}$) \rightarrow group by “key”
 Reducer ($\{“key, value_i\}$) $\rightarrow (“key, f(value_i))$
 Can repeat, constant number of rounds

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.

4.3.8 Matrix Multiplication of Large Matrices

Consider two large matrices A and B. Matrix A has dimension $I \times J$ with element $a(i, j)$, where i varies from 0 to $j - 1$. Matrix B has dimension $J \times K$ with elements $b(j, k)$, where j varies from 0 to $k - 1$. Then, matrix $C = A^*B$ has dimension $I \times K$ and elements $c(i, k)$ are defined as

$$c(i, k) = \text{sum of } a(i, j)^*b(j, k) \text{ for } j \text{ varying from } 0 \text{ to } j - 1$$

Note: Split A and B into blocks (sub-matrices) small enough so that a pair of blocks can be multiplied in memory on a single node in the cluster.

Let

IB = Number of rows per A block and C block.

JB = Number of columns per A block = Number of rows per B block.

KB = Number of columns per B block and C block.

NIB = number of A row and C row partitions = $(I - 1)/IB + 1$

NJB = number of A column = number of B row partitions = $(J - 1)/JB + 1$

NKB = number of B column and C column partitions = $(K - 1)/KB + 1$

Following notations are used for the blocks:

ib varies from 0 to NIB – 1

jb varies from 0 to NJB – 1

kb varies from 0 to NKB – 1

$A[ib,jb]$ = The block of A consisting of rows $IB * ib$ through $\min(IB * (ib + 1), I) - 1$ columns $JB * jb$ through $\min(JB * (jb + 1), J) - 1$

$B[jb,kb]$ = The block of B consisting of rows $JB * jb$ through $\min(JB * (jb + 1), J) - 1$ columns $KB * kb$ through $\min(KB * (kb + 1), K) - 1$

$C[ib,kb]$ = The block of C consisting of rows $IB * ib$ through $\min(IB * (ib + 1), I) - 1$ columns $KB * kb$ through $\min(KB * (kb + 1), K) - 1$

$C[ib,jb,kb] = A[ib,jb] * B[jb,kb]$

$C[ib,kb] = \text{sum of } A[ib,jb] * B[jb,kb] \text{ for } jb \text{ varying from } 0 \text{ to } JB - 1$
 $= \text{sum of } C[ib,jb,kb] \text{ for } jb \text{ varying from } 0 \text{ to } JB - 1$

Note:

A blocks have dimension $IB \times JB$ and $NIB * NJB$ blocks.

B blocks have dimension $JB \times KB$ and $NJB * NKB$ blocks.

C blocks have dimension $IB \times KB$ and $NIB * NKB$ blocks.

This leaves the “remainder” blocks at the bottom and right edges of A, B and C, which may have smaller dimension.

The matrix cell $A(i,k)$ is the block cell $A[i/IB, k/KB](i \bmod IB, k \bmod KB)$, and similarly for I and C.

The block cell $A[Ib,kb](i,k)$ is the matrix cell $A(ib * IB + i, kb * KB + k)$, and similarly for B and C.

4.3.9 MapReduce Job Structure

Suppose two MapReduce Jobs are done. The first Job does the block multiplications and the second Job sums up the results. The mappers are responsible for distributing the block data to the reducers, with the help of a carefully chosen intermediate Key structure and Key comparator and partitioning functions. The reducers do the block multiplications.

Block multiplication tasks are assigned to the reducers in more than one way. Each one can be chosen based on different performance characteristics and tradeoffs.

Case 1: The simplest way is to have each reducer do just one of the block multiplications. Reducer $R[ib,jb,kb]$ is responsible for multiplying $A[ib,jb]$ times $B[jb,kb]$ to produce $C[ib,jb,kb]$. So maximum of $NIB * NJB * NKB$ reducers multiplying blocks are required to work in parallel.

This requires a large amount of network traffic since it uses lots of reducers, which makes good use of parallelization. Network traffic can be reduced at the expense of fewer reducers with a lower level of parallelization; this is discussed in Case 2.

Case 2: Here a single reducer to multiply a single A block times a whole row of B blocks. That is, for each A block $A[ib,jb]$, a single reducer $R[ib,jb]$ can be used. This involves a maximum of $NIB * NJB$ reducers. Here, the data for an A block $A[ib,jb]$ only has to be routed to the single reducer $R[ib,jb]$, and the data for a B block $B[jb,kb]$ has to be routed to the NIB reducers $R[ib,jb]$ where ib varies from 0 to $NIB - 1$.

The worst-case number of intermediate Key/Value pairs transferred over the network is $I * J + NIB * J * K = J * (I + NIB * K)$. This is a considerable improvement over Case 1 in terms of network traffic, with fewer reducers doing more work, resulting in a lower level of parallelization.

Note: If $(K + NKB * I) < (I + NIB * K)$, then use a single reducer to multiply a single B block times a whole column of A blocks. For each B block $B[jb,kb]$, a single reducer $R[jb,kb]$ can be used. This involves a maximum of $NJB * NKB$ reducers.

Case 3: In the first two cases, each reducer emits one or more $C[ib,jb,kb]$ blocks, and a second MapReduce Job is used to sum up over jb to produce the final $C[ib,kb]$ blocks.

In this case, a single reducer $R[ib,kb]$ is used to compute the final $C[ib,kb]$ block. The reducer receives from the mappers all the $A[ib,jb]$ and $B[jb,kb]$ blocks for $0 \leq jb < NJB$, interleaved in the following order:

$A[ib,0] B[0,kb] A[ib,1] B[1,kb] \dots A[ib,NJB - 1] B[NJB - 1, kb]$

The reducer multiplies A and B block pairs and add up the results. That is, it computes and emits the sum over $0 \leq jb < JB$ of $A[ib,jb] * B[jb,kb]$. The maximum number of reducers with this strategy is $NIB * NKB$.

Summary

- We have seen in this chapter why large data and what is large data processing.
- Cloud computing and MapReduce are discussed.
- MapReduce and the importance of the underlying distributed file system are covered.
- MapReduce algorithms are used for processing relational data:
 - Selection: Map over tuples, emit only tuples that meet criteria.
 - Projection: Map over tuples, emit new tuples with appropriate attributes.
 - Group by, sorting, partitioning are handled automatically by shuffle/sort in MapReduce.
- Selection, projection and other computations (e.g., aggregation) are performed either in mapper or reducer.
- Multiple strategies for relational joins.
- **Combiner:** Instead of emitting intermediate output for every input key-value pair, the mapper aggregates partial results across multiple input records and only emits intermediate key-value pairs after some amount of local aggregation is performed.
- **MapReduce:** Matrix multiplication block diagram and pseudocode are discussed.

Review Questions

1. Why do organizations store large volumes of data?
2. What is shuffling in MapReduce?
3. How does NameNode tackle DataNode failures?
4. What is InputFormat in Hadoop?
5. What is the purpose of RecordReader in Hadoop?
6. What is InputSplit in MapReduce?
7. What are combiners? When should one use a combiner in my MapReduce Job?

Laboratory Exercise

In this section we will cover the following:

1. Create a file into the local file system and export it to the HDFS file system.
2. To work with Pig in both the modes: (a) Hadoop local mode and (b)HDFS or MapReduce mode.
3. Define schema by using “Load” command and validating it.
4. Relational operators.

A. Create a file into the local file system and export it to the HDFS file system. To do so, follow the commands given below:

\$ cat> students

```
101,sushil,68.21F,male  
102,deepak,70.33F,male  
103,siddhesh,80.21F,male  
104,vikas,70.36F,male  
105,sumit,62.00F,male  
106,ankita,68.00F,female  
107,supriya,72.21F,female  
108,neha,91.22F,female  
109,mayuresh,85.66F,male  
110,pravin,79.88F,male
```

Press **ctrl+z** to save and exit.

Load it to the HDFS file system by typing following command:

```
[root@quickstart ~]# hadoop fs -mkdir /user/cloudera/pig
```

```
[root@quickstart ~]# hadoop fs -mkdir /user/cloudera/pig/example  
[root@quickstart ~]# hadoop fs -copyFromLocal students /user/cloudera/pig/example  
[root@quickstart ~]# hadoop fs -ls /user/cloudera/pig/example  
Found 1 items  
-rw-r--r-- 1 root cloudera 187 2015-07-29 03:18 /user/cloudera/pig/example/students
```

- B. To enter into the grunt shell in HDFS mode type the below command and press enter. Basically pig works in two modes:

1. **Local mode:** To work in local mode use the below command:

```
$ pig -x local
```

Note: Here local means Hadoop local.

2. **HDFS or MapReduce mode:** To work in HDFS mode the below command can be used

```
$ pig
```

For example,

```
[root@quickstart ~]# pig  
grunt>
```

Note: The above command will take you to the **grunt shell** as shown above. Here we can do various operations on data and save the result to the local file system or HDFS file system.

- C. Once we are done with all of the above commands, the first thing that we need to do is define schema by using “Load” command and validating it as follows:

```
grunt> student = load '/user/cloudera/pig/example/students' USING PigStorage(',') as (roll  
no:int,name:chararray,percentage:float,sex:chararray);
```

Here basically dump operator is used to display the result or the content of pig storage file. Syntax is:

```
dump [file_name] ;
```

For example,

```
grunt> DUMP student;  
(101,sushil,68.21,male)  
(102,deepak,70.33,male)  
(103,siddhesh,80.21,male)  
(104,vikas,70.36,male)  
(105,sumit,62.0,male)  
(106,ankita,68.0,female)  
(107,supriya,72.21,female)
```

```
(108,neha,91.22,female)  
(109,mayuresh,85.66,male)  
(110,pravin,79.88,male)
```

```
grunt>student_name = foreach student generate name;  
grunt> dump student_name;
```

```
(sushil)  
(deepak)  
(siddhesh)  
(vikas)  
(sumit)  
(ankita)  
(supriya)  
(neha)  
(mayuresh)  
(pravin)
```

Before going to the relation operators, let us load one more bag named dept to the HDFS file system. Use the above procedure to do so.

```
(10,sushil,engineer,50000)  
(10,nikhil,engineer,45000)  
(10,supriya,engineer,50000)  
(10,siddhesh,engineer,44000)  
(20,manish,clerk,20000)  
(20,mahesh,clerk,25000)  
(30,minal,scientist,60000)  
(30,krishna,scientist,80000)  
(30,govind,scientist,60000)  
(50,rahul,chemist,80000)  
(40,neha,biochemist,90000)
```

```
grunt>dept = load '/user/cloudera/pig/example/dept' USING PigStorage(',') as  
(id:int,name:chararray,dept_name:chararray,sal:int);
```

```
grunt> dump dept;
```

```
(10,sushil,engineer,50000)  
(10,nikhil,engineer,45000)
```

```
(10,supriya,engineer,50000)
(10,siddhesh,engineer,44000)
(20,manish,clerk,20000)
(20,mahesh,clerk,25000)
(30,minal,scientist,60000)
(30,krishna,scientist,80000)
(30,govind,scientist,60000)
(50,rahul,chemist,80000)
(40,neha,biochemist,90000)
```

D. Relational Operator

1. **CROSS:** The **cross** operator is used to calculate the cross product of two or more relation. Cross operation is expensive and therefore one should avoid using it. For example, suppose we have relation student and dept. Then

grunt> DUMP student;

```
(101,sushil,68.21,male)
(102,deepak,70.33,male)
(103,siddhesh,80.21,male)
(104,vikas,70.36,male)
(105,sumit,62.0,male)
(106,ankita,68.0,female)
(107,supriya,72.21,female)
(108,neha,91.22,female)
(109,mayuresh,85.66,male)
(110,pravin,79.88,male)
```

grunt> dump dept;

```
(10,sushil,engineer,50000)
(10,nikhil,engineer,45000)
(10,supriya,engineer,50000)
(10,siddhesh,engineer,44000)
(20,manish,clerk,20000)
(20,mahesh,clerk,25000)
(30,minal,scientist,60000)
(30,krishna,scientist,80000)
```

(30,govind,scientist,60000)

(50,rahul,chemist,80000)

(40,neha,biochemist,90000)

Cross product of student and dept is:

grunt> x = cross student, dept;

grunt> dump x;

(110,pravin,79.88,male,40,neha,biochemist,90000)

(110,pravin,79.88,male,50,rahul,chemist,80000)

(110,pravin,79.88,male,30,govind,scientist,60000)

(110,pravin,79.88,male,30,krishna,scientist,80000)

(110,pravin,79.88,male,30,minal,scientist,60000)

(110,pravin,79.88,male,20,mahesh,clerk,25000)

(110,pravin,79.88,male,20,manish,clerk,20000)

(110,pravin,79.88,male,10,siddhesh,engineer,44000)

(110,pravin,79.88,male,10,supriya,engineer,50000)

(110,pravin,79.88,male,10,nikhil,engineer,45000)

(110,pravin,79.88,male,10,sushil,engineer,50000)

(109,mayuresh,85.66,male,40,neha,biochemist,90000)

(109,mayuresh,85.66,male,50,rahul,chemist,80000)

(109,mayuresh,85.66,male,30,govind,scientist,60000)

(109,mayuresh,85.66,male,30,krishna,scientist,80000)

(109,mayuresh,85.66,male,30,minal,scientist,60000)

(109,mayuresh,85.66,male,20,mahesh,clerk,25000)

(109,mayuresh,85.66,male,20,manish,clerk,20000)

(109,mayuresh,85.66,male,10,siddhesh,engineer,44000)

(109,mayuresh,85.66,male,10,supriya,engineer,50000)

(109,mayuresh,85.66,male,10,nikhil,engineer,45000)

(109,mayuresh,85.66,male,10,sushil,engineer,50000)

(108,neha,91.22,female,40,neha,biochemist,90000)

(108,neha,91.22,female,50,rahul,chemist,80000)

(108,neha,91.22,female,30,govind,scientist,60000)

(108,neha,91.22,female,30,krishna,scientist,80000)

(108,neha,91.22,female,30,minal,scientist,60000)

(108,neha,91.22,female,20,mahesh,clerk,25000)
(108,neha,91.22,female,20,manish,clerk,20000)
(108,neha,91.22,female,10,siddhesh,engineer,44000)
(108,neha,91.22,female,10,supriya,engineer,50000)
(108,neha,91.22,female,10,nikhil,engineer,45000)
(108,neha,91.22,female,10,sushil,engineer,50000)
(107,supriya,72.21,female,40,neha,biochemist,90000)
(107,supriya,72.21,female,50,rahul,chemist,80000)
(107,supriya,72.21,female,30,govind,scientist,60000)
(107,supriya,72.21,female,30,krishna,scientist,80000)
(107,supriya,72.21,female,30,minal,scientist,60000)
(107,supriya,72.21,female,20,mahesh,clerk,25000)
(107,supriya,72.21,female,20,manish,clerk,20000)
(107,supriya,72.21,female,10,siddhesh,engineer,44000)
(107,supriya,72.21,female,10,supriya,engineer,50000)
(107,supriya,72.21,female,10,nikhil,engineer,45000)
(107,supriya,72.21,female,10,sushil,engineer,50000)
(106,ankita,68.0,female,40,neha,biochemist,90000)
(106,ankita,68.0,female,50,rahul,chemist,80000)
(106,ankita,68.0,female,30,govind,scientist,60000)
(106,ankita,68.0,female,30,krishna,scientist,80000)
(106,ankita,68.0,female,30,minal,scientist,60000)
(106,ankita,68.0,female,20,mahesh,clerk,25000)
(106,ankita,68.0,female,20,manish,clerk,20000)
(106,ankita,68.0,female,10,siddhesh,engineer,44000)
(106,ankita,68.0,female,10,supriya,engineer,50000)
(106,ankita,68.0,female,10,nikhil,engineer,45000)
(106,ankita,68.0,female,10,sushil,engineer,50000)
(105,sumit,62.0,male,40,neha,biochemist,90000)
(105,sumit,62.0,male,50,rahul,chemist,80000)
(105,sumit,62.0,male,30,govind,scientist,60000)
(105,sumit,62.0,male,30,krishna,scientist,80000)
(105,sumit,62.0,male,30,minal,scientist,60000)
(105,sumit,62.0,male,20,mahesh,clerk,25000)

(105,sumit,62.0,male,20,manish,clerk,20000)
(105,sumit,62.0,male,10,siddhesh,engineer,44000)
(105,sumit,62.0,male,10,supriya,engineer,50000)
(105,sumit,62.0,male,10,nikhil,engineer,45000)
(105,sumit,62.0,male,10,sushil,engineer,50000)
(104,vikas,70.36,male,40,neha,biochemist,90000)
(104,vikas,70.36,male,50,rahul,chemist,80000)
(104,vikas,70.36,male,30,govind,scientist,60000)
(104,vikas,70.36,male,30,krishna,scientist,80000)
(104,vikas,70.36,male,30,minal,scientist,60000)
(104,vikas,70.36,male,20,mahesh,clerk,25000)
(104,vikas,70.36,male,20,manish,clerk,20000)
(104,vikas,70.36,male,10,siddhesh,engineer,44000)
(104,vikas,70.36,male,10,supriya,engineer,50000)
(104,vikas,70.36,male,10,nikhil,engineer,45000)
(104,vikas,70.36,male,10,sushil,engineer,50000)
(103,siddhesh,80.21,male,40,neha,biochemist,90000)
(103,siddhesh,80.21,male,50,rahul,chemist,80000)
(103,siddhesh,80.21,male,30,govind,scientist,60000)
(103,siddhesh,80.21,male,30,krishna,scientist,80000)
(103,siddhesh,80.21,male,30,minal,scientist,60000)
(103,siddhesh,80.21,male,20,mahesh,clerk,25000)
(103,siddhesh,80.21,male,20,manish,clerk,20000)
(103,siddhesh,80.21,male,10,siddhesh,engineer,44000)
(103,siddhesh,80.21,male,10,supriya,engineer,50000)
(103,siddhesh,80.21,male,10,nikhil,engineer,45000)
(103,siddhesh,80.21,male,10,sushil,engineer,50000)
(102,deepak,70.33,male,40,neha,biochemist,90000)
(102,deepak,70.33,male,50,rahul,chemist,80000)
(102,deepak,70.33,male,30,govind,scientist,60000)
(102,deepak,70.33,male,30,krishna,scientist,80000)
(102,deepak,70.33,male,30,minal,scientist,60000)
(102,deepak,70.33,male,20,mahesh,clerk,25000)
(102,deepak,70.33,male,20,manish,clerk,20000)

```
(102,deepak,70.33,male,10,siddhesh,engineer,44000)
(102,deepak,70.33,male,10,supriya,engineer,50000)
(102,deepak,70.33,male,10,nikhil,engineer,45000)
(102,deepak,70.33,male,10,sushil,engineer,50000)
(101,sushil,68.21,male,40,neha,biochemist,90000)
(101,sushil,68.21,male,50,rahul,chemist,80000)
(101,sushil,68.21,male,30,govind,scientist,60000)
(101,sushil,68.21,male,30,krishna,scientist,80000)
(101,sushil,68.21,male,30,minal,scientist,60000)
(101,sushil,68.21,male,20,mahesh,clerk,25000)
(101,sushil,68.21,male,20,manish,clerk,20000)
(101,sushil,68.21,male,10,siddhesh,engineer,44000)
(101,sushil,68.21,male,10,supriya,engineer,50000)
(101,sushil,68.21,male,10,nikhil,engineer,45000)
(101,sushil,68.21,male,10,sushil,engineer,50000)
```

2. **DISTINCT:** This operator is used to remove the duplicate tuples in a relation. It does not preserve the original order of the contents (because to eliminate the duplicate tuples, it first sorts the data). For example, suppose that we have relation A.

```
grunt> A = load '/user/cloudera/pig/example/A' USING PigStorage(',') as (a1:int,a2:int,a3:int,a4:int);
```

```
grunt> dump A;
```

```
(12,2,3,4)
(2,4,5,65)
(2,4,5,65)
(23,44,3,2)
(23,44,3,2)
(1,2,3,4)
(4,3,2,1)
(5,6,7,8)
```

```
grunt> z = distinct A;
```

```
grunt> dump z;
```

```
(1,2,3,4)
(2,4,5,65)
```

```
(4,3,2,1)  
(5,6,7,8)  
(12,2,3,4)  
(23,44,3,2)
```

3. **FILTER:** This operator can be used to select the required data based on some condition or it filters the unwanted data based on condition. For example, suppose that we have relation A.

```
grunt> A = load '/user/cloudera/pig/example/A' USING PigStorage(',') as  
(a1:int,a2:int,a3:int,a4:int);
```

```
grunt> dump A;
```

```
(12,2,3,4)  
(2,4,5,65)  
(2,4,5,65)  
(23,44,3,2)  
(23,44,3,2)  
(1,2,3,4)  
(4,3,2,1)  
(5,6,7,8)
```

```
grunt> y = FILTER A BY a2 == 2;
```

```
grunt> dump y;
```

```
(12,2,3,4)  
(1,2,3,4)
```

Suppose that we have relation dept.

```
grunt>dept = load '/user/cloudera/pig/example/dept' USING PigStorage(',') as  
(id:int,name:chararray,dept_name:chararray,sal:int);
```

```
grunt> dump dept;
```

```
(10,sushil,engineer,50000)  
(10,nikhil,engineer,45000)  
(10,supriya,engineer,50000)  
(10,siddhesh,engineer,44000)  
(20,manish,clerk,20000)  
(20,mahesh,clerk,25000)  
(30,minal,scientist,60000)  
(30,krishna,scientist,80000)
```

```
(30,govind,scientist,60000)
```

```
(50,rahul,chemist,80000)
```

```
(40,neha,biochemist,90000)
```

```
grunt> y = FILTER dept BY dept_name == 'engineer';
```

```
grunt> dump y;
```

```
(10,sushil,engineer,50000)
```

```
(10,nikhil,engineer,45000)
```

```
(10,supriya,engineer,50000)
```

```
(10,siddhesh,engineer,44000)
```

4. **FOREACH:** This operator is used to generate data transformation based on columns data.

Syntax

```
alias = FOREACH {gen_blk | nested_gen_blk } [AS schema];
```

where

(a) alias = The name of relation.

(b) gen_blk= FOREACH ... GENERATE used with a relation(outer bag).

{Use this syntax: alias = FOREACH alias GENERATE expression [expression....]}

(c) nested_gen_blk = FOREACH ... GENERATE used with a inner bag. Use this syntax:

```
alias = FOREACH nested_alias {  
    alias = nested_op; [alias = nested_op; ...]  
        GENERATE expression [, expression ...]  
};
```

where:

The nested block is enclosed in opening and closing brackets { ... }.

The GENERATE keyword must be the last statement within the nested block.

(d) AS = Keword.

(e) schema = A schema using the AS keyword.

(f) nested_alias = Name of the inner bag

(g) nested_op = It can be DISTINCT,FILTER,ORDER,SAMPLE and LIMIT.

USAGE

(a) If a relation A is an outer bag then a FOREACH statement cloud look like this.

```
Z = FOREACH A GENERATE a2;
```

(b) If a relation A is an inner bag then a FOREACH statement will look like this.

```
Z = FOREACH B {  
    k = FILTER A BY 'abc'  
    GENERATE COUNT (k.$0);  
}
```

For example,

Projection

Consider the example below. The asterisk(*) is used to combine all of the tuples from the relation dept.

```
grunt> X = FOREACH dept GENERATE *;
```

```
grunt> dump X;
```

```
(10,sushil,engineer,50000)  
(10,nikhil,engineer,45000)  
(10,supriya,engineer,50000)  
(10,siddhesh,engineer,44000)  
(20,manish,clerk,20000)  
(20,mahesh,clerk,25000)  
(30,minal,scientist,60000)  
(30,krishna,scientist,80000)  
(30,govind,scientist,60000)  
(50,rahul,chemist,80000)  
(40,neha,biochemist,90000)
```

Note: DESCRIBE bag name is used to display the schema.

```
grunt> describe dept;
```

o/p:

```
dept: {id: int,name: chararray,dept_name: chararray,sal: int}
```

Let us say, we want to display first two columns of the relation (i.e. id and name).

```
grunt> X = FOREACH dept GENERATE id,name;
```

```
grunt> dump X;
```

```
(10,sushil)  
(10,nikhil)  
(10,supriya)  
(10,siddhesh)  
(20,manish)  
(20,mahesh)
```

```
(30,minal)  
(30,krishna)  
(30,govind)  
(50,rahul)  
(40,neha)
```

Suppose we have three relation A,B,C as shown below.

```
grunt> A = load '/user/cloudera/pig/example/A' USING PigStorage(',') as (a1:int,a2:int,a3:int,a4:int);
```

```
grunt> dump A;
```

```
(12,2,3,4)  
(2,4,5,65)  
(2,4,5,65)  
(23,44,3,2)  
(23,44,3,2)  
(1,2,3,4)  
(4,3,2,1)  
(5,6,7,8)
```

```
grunt> B = load '/user/cloudera/pig/example/B' USING PigStorage(',') as (b1:int,b2:int);
```

```
grunt> dump B;
```

```
(23,4)  
(1,3)  
(2,7)  
(5,3)  
(1,4)  
(6,4)
```

```
grunt> C = COGROUP A BY a1 inner, B BY b1 Inner;
```

```
grunt> dump C;
```

```
(1,{(1,2,3,4)},{(1,4),(1,3)})  
(2,{(2,4,5,65),(2,4,5,65)},{(2,7)})  
(5,{(5,6,7,8)},{(5,3)})  
(23,{(23,44,3,2),(23,44,3,2)},{(23,4)})
```

Example: Nested Projection

If one of the field in input relation is a [tuple,bag] then we can perform projection operation on that filed.

```
grunt> Z = FOREACH C GENERATE group,B.b2;  
grunt> dump Z;  
(1,{(4),(3)})  
(2,{(7)})  
(5,{(3)})  
(23,{(4)})
```

In the below example multiple nested columns are retained.

```
grunt> Z = FOREACH C GENERATE group,A.(a1,a2);  
grunt> dump Z;  
(1,{(1,2)})  
(2,{(2,4),(2,4)})  
(5,{(5,6)})  
(23,{(23,44),(23,44)})
```

5

Finding Similar Items

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Understand the motivation for finding similarities between items in a large dataset.
- Perceive similarity as a form of nearest neighbor search.
- Learn and understand different similarity and distance measures like Euclidean measures, Jaccard measures among others.
- Apply the appropriate distance measure to the given application.

5.1

Introduction

For many different problems in data mining, we need to quantify how close two objects are. Measuring similarity and correlations is a basic building block for activities such as clustering, classification and anomaly detection. So getting familiar and understanding these metrics in depth helps in gaining better insights on how to use them in advance data mining algorithms and analysis. Some applications include the following:

1. **Advertiser keyword suggestions:** When targeting advertisements via keywords, it is useful to expand the manually input set of keywords by other similar keywords. This requires finding all other keywords that share a similar meaning with keyword. The word similar is based on a measure of indicating how alike two keywords are with respect to their meaning. Any two keywords that have a similarity measure beyond a particular threshold are said to be similar keywords and these are suggested to the advertiser so as to maximize the reach of their advertisements.
2. **Collaborative filtering:** Collaborative filtering applications require knowing which users have similar interests. For this reason, given a person or user, it is required to find all other objects more similar than a particular threshold. This information can then be used to drive sales.
3. **Web search:** Given a user query similarity measures are used to increase the scope of the query to get more extended and relevant results. Further, most search engines will add clusters of similar queries to a given query as an effective query expansion strategy.

The above applications are only a representative sample of the thousands of real-life applications that use some form of similarity search as part of their strategy. Most of these applications have been around for several years, but the scale at which we need to solve them keeps steadily increasing.

Further there has been a deluge of new applications mostly web based like social networks, sophisticated spam filters, mail servers, etc. all of which use some notion of finding similar items in their algorithms. For instance, Facebook may want to identify similar class of individuals to target ads, sophisticated mail servers like Gmail would like to cluster similar e-mail messages, the list is endless. All these applications have in common the huge volume and velocity of data they have to deal with. Google currently holds an index of more than 1 trillion webpages. Running a duplicate detection algorithm on such a large amount of data is really a daunting task. The number of tweets that Twitter handles in a single day surpasses 200 million. Further, many of these applications involve domains where the dimensionality of the data far exceeds the number of points. Most popular are the recommendation engines as used by E-Bay or Amazon, where the number of products is so large that dimension of a single data point is in hundreds of thousands.

All the above-mentioned problems use similarity detection as part of their problem-solving strategy. This dictates the need for a formal definition of similarity or distance. This definition will depend on the type of data that we have. There are a few standard similarity and distance measures popularly used.

In this chapter, we introduce the notion of similarity and distance. First, we introduce the general version of this problem called Nearest Neighbor (NN) Search. We discuss a few applications of NN search pertaining to big data.

We then introduce the concept of Jaccard Similarity. We also discuss a few commonly used Distance Measures.

5.2 Nearest Neighbor Search

One notion of similarity that most commonly occurs in Big Data applications is “Given some data, one often wants to find approximate matches from a large dataset”. **Nearest neighbor search (NN search)**, also known as **proximity search, similarity search or closest point search**, is an optimization problem for finding closest (or most similar) points. Formally, the NN search problem is defined as follows: **Given a set S of points in a space M and a query point $q \in M$, find the set of closest points in S to q .**

This definition of NN finds numerous applications in varied domains, such as multimedia, biology, finance, sensor, surveillance, social network, etc. For example, given a query image, find similar images in a photo database; given a user-log profile, find similar users in a user database or social network; given a DNA sequence, find similar DNA sequences; given a stock-trend curve, find similar stocks from stock history data; given an event from sensor data, find similar events from sensor network data log; and so on. Further several large-scale machine learning, data mining and social network problems also involve NN search as one of the most crucial steps. Several classification techniques

like k-NN, pattern-based classification and their variations are some form of the basic NN search. Recommendation engines and collaborative filtering systems rely on finding similar users/objects, which is also a form of NN search. Finally, when data mining problems (classification, clustering, pattern identification, etc.) go to large scale and enter the big data realm, approximate NN search helps to speed up these algorithms. NN search helps us to approximate the computation of distance or similarity measures efficiently.

5.2.1 The NN Search Problem Formulation

The NN search problem, illustrated in Fig. 5.1, can be formulated as follows:

Given a vector dataset and a query vector, how to find the vector(s) in the dataset closest to the query.

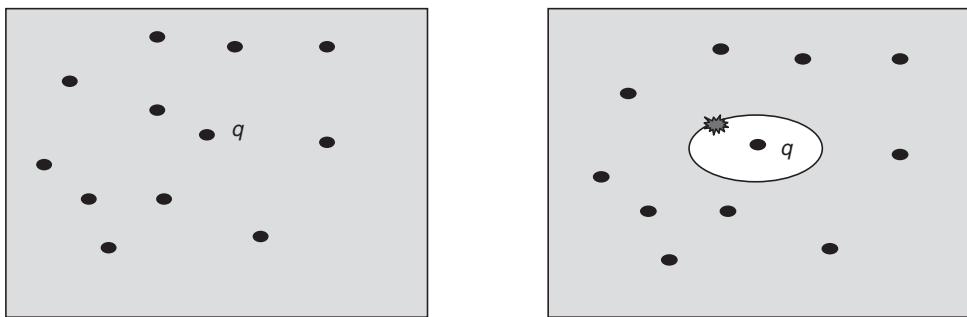


Figure 5.1 Illustrating NN search.

More formally:

Suppose there is a dataset X with n points $X = \{X_i, i = 1, \dots, n\}$.

Given a query point q and a distance metric $D(s, t)$,

find q 's nearest neighbor X_{nn} in X , that is

$$D(X_{nn}, q) \leq D(X_i, q), i = 1, \dots, n$$

An alternate formulation of the NN problem can be defined in the realm of set theory. This attempts to answer the query “Given a set, find similar sets from a large dataset” or “Given a large dataset, extract all similar sets of items”. This basically amounts to finding the size of the intersection of two sets to evaluate similarity. This notion of similarity is called “*Jaccard Similarity*” of sets.

5.2.2 Jaccard Similarity of Sets

A similarity measure $s(A, B)$ indicates the closeness between sets A and B . A good similarity measure has the following properties:

1. It has a large value if the objects A and B are close to each other.
2. It has a small value if they are different from each other.
3. It is (usually) 1 if they are same sets.
4. It is in the range $[0, 1]$.

Consider two sets $A = \{0, 1, 2, 4, 6\}$ and $B = \{0, 2, 3, 5, 7, 9\}$. We now explain the process to evaluate how similar are A and B .

The Jaccard Similarity is defined as

$$JS(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

For the above sets we have

$$JS(A, B) = \frac{|\{0, 2\}|}{|\{0, 1, 2, 3, 5, 6, 7, 9\}|} = \frac{2}{8} = 0.25$$

Given a set A , the cardinality of A , denoted by $|A|$, counts how many elements are in A . The intersection between two sets A and B is denoted by $(A \cap B)$ and reveals all items which are in both sets. The union between two sets A and B is denoted by $(A \cup B)$ and reveals all items which are in either set. We can indeed confirm that Jaccard Similarity satisfies all the properties of a similarity function.

Now we see more examples on this.

Example 1

Compute the Jaccard Similarity of each pair of the following sets:

$$\{1, 2, 3, 4, 5\}, \{1, 6, 7\}, \{2, 4, 6, 8\}$$

Solution

We use the general formula for the Jaccard Similarity of two sets. For the three combinations of pairs above, we have

$$J(\{1, 2, 3, 4, 5\}, \{1, 6, 7\}) = \frac{1}{7}$$

$$J(\{1, 2, 3, 4, 5\}, \{2, 4, 6, 8\}) = \frac{2}{7}$$

$$J(\{1, 6, 7\}, \{2, 4, 6, 8\}) = \frac{1}{6}$$

Example 2

Consider two customers $C1$ and $C2$ with the following purchases:

$$C1 = \{\text{Pen, Bread, Belt, Chocolate}\}$$

$$C2 = \{\text{Chocolate, Printer, Belt, Pen, Paper, Juice, Fruit}\}$$

$$\begin{aligned} JS(C1, C2) &= \frac{|\{\text{Pen, Belt, Chocolate}\}|}{|\{\text{Pen, Bread, Belt, Chocolate, Printer, Paper, Juice, Fruit}\}|} \\ &= \frac{3}{8} = 0.375 \end{aligned}$$

5.3 Applications of Nearest Neighbor Search

The NN search problem arises in numerous fields of application. The following list illustrates some of the common applications:

- 1. Optical Character Recognition (OCR):** “Intelligent” handwriting recognition and most modern OCR software use NN classifiers; for example, the k -NN algorithms are used to compare image features with stored glyph features and choose the nearest match.
- 2. Content-based image retrieval:** These systems typically provide example images and the systems find similar images using NN approach (see Fig. 5.2).

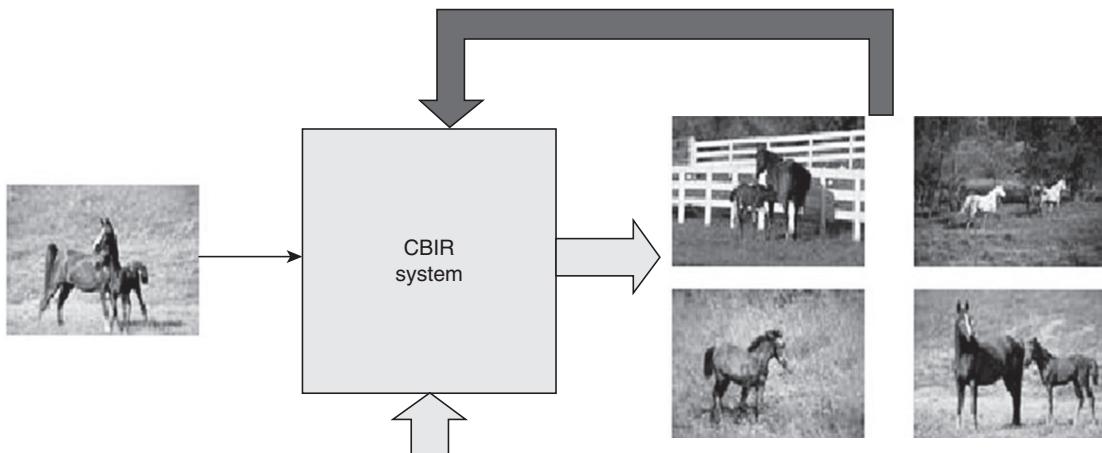


Figure 5.2 Indicating NN search with images.

3. **Collaborative filtering:** In general, collaborative filtering is the process of filtering for information or patterns using a set of collaborating agents, viewpoints, data sources, etc. Applications of collaborative filtering usually involve massive, one important application being Recommendation systems. A collaborative filtering approach is based on the idea that people often get the best recommendations from someone with similar tastes as themselves. Collaborative filtering explores techniques for matching people with similar interests and making recommendations on this basis.
4. **Document Similarity:** A large number of applications like web search engines, news aggregators and the like need to identify textually similar documents from a large corpus of documents like web pages, or a collection of news articles, collection of tweets, etc.

We discuss in detail the two important applications of NN search – document similarity and collaborative filtering – in the following sections.

5.4 Similarity of Documents

Automated analysis and organization of large document repositories is extremely essential and a challenge in many current applications like storage and retrieval of records in a large enterprise, maintaining and retrieving a variety of patient-related data in a large hospital, web search engines, identifying trending topics on Twitter, etc. The sheer volume, variety and speed at which the web or the modern businesses churn out these documents make it an important Big Data problem. One key issue in document management is the quantitative assessment of document similarities. A reliable and efficient similarity measure will provide answers to questions like: How similar are the two text documents? Are two patient histories similar? Which documents match a given query best?, etc.

It is very important to emphasize that the aspect of similarity we are looking for in these applications is character-level similarity, not semantic similarity, which requires us to examine the words in the documents and their uses. That problem is also interesting but is addressed by other techniques. In this text we address the issue textual similarity which is also important in its own right. Many of these involve finding duplicates or near duplicates. Identical duplicate documents are generally very easy to detect, for example, using a simple hash algorithm. However, finding documents that are similar, or near-duplicates, requires more effort.

Some useful applications for text based similarity in the big data scenario include the following:

1. Near-duplicate detection to improve search results quality in search engines.
2. Human Resources applications, such as automated CV to job description matching, or finding similar employees.
3. Patent research, through matching potential patent applications against a corpus of existing patent grant.

4. Document clustering and auto-categorization using seed documents.
5. Security scrubbing – finding documents with very similar content, but with different access control lists.

5.4.1 Plagiarism Detection

Plagiarism detection is the process of locating instances of plagiarism within a work or document. The widespread use of computers and the advent of the Internet have made it easier to plagiarize the work of others. Most cases of plagiarism are found in academia, where documents are typically essays or reports. However, plagiarism can be found in virtually any field, including scientific papers, art designs and source code.

Finding plagiarized documents attempts to use textual similarity measures. Literal copies, aka copy and paste (c&p) plagiarism, or modestly disguised plagiarism cases can be detected with high accuracy. But a smart plagiarizer may extract only some parts of a document . He/She may alter a few words and may alter the order in which the sentences of the original appear. In most cases, the resulting document may still contain 50% or more of the original, which can then be detected. No simple process of comparing documents character by character will detect a sophisticated attempt at plagiarism.

In the following sub-sections we will discuss a few plagiarism detection tools.

5.4.1.1 Turnitin

Turnitin is a web-based system used for detecting and preventing plagiarism and improper citations in someone's works (homeworks, projects, papers, etc.). Once loaded into the system, the content of the document is compared against a vast database. The system detects excerpts, suspicious similarities, uncited references, etc. and produces a similarity report. Turnitin's control database consists of academic databases, academic journals and publications, 200+ million students' homework, 17+ billion web pages.

5.4.1.2 iThenticate

iThenticate is among the most popular professional plagiarism prevention tool. It is designed for the authentication of faculty research, faculty-authored articles or textbooks, grant proposals, supplemental course materials and dissertations and theses. iThenticate compares every submitted paper to a massive database of content from over 90,000 major newspapers, magazines, scholarly journals and books as well as a database of over 14 billion current and archived pages of web content. In addition, iThenticate checks materials from over 50,000 scholarly journals and more than 150 STM publishers. Within moments of submitting a document to iThenticate, a similarity report is produced that shows matches to documents in the iThenticate database. Similarity Reports include an overall similarity index that shows the overall percentage of matched content in the document and a list of all matched sources that make up the similarity index.

Other reports include the following:

1. Side-by-side comparison of the document to matched sources in the database.
2. Largest matches showing where sources and text of the largest content match in the document.
3. Summary report that provides a high-level overview of matched content in the document.

5.4.2 Document Clustering

Computing pair-wise similarity on large document collections is a task common to a variety of problems, such as clustering and cross-document co-reference resolution. A web search engine often returns 1000s of pages in response to a broad query, making it difficult for users to browse or to identify relevant information. For example, Google along with every primary link that is shown on the result page includes a “Similar” link, which provides a link to a set of documents similar to the primary link. Document similarity with respect to the query posed is evaluated and all similar documents are clustered and ranked. The primary link points to the top-ranked page.

Clustering methods can be used to automatically group the retrieved documents into a list of meaningful categories. The automatic generation of taxonomy of Web documents like that provided by Yahoo! is one such popular example.

Hewlett-Packard has many millions of technical support documents in a variety of collections. As a part of content management, such collections are periodically merged and groomed. In the process, it becomes important to identify and weed out support documents that are largely duplicates of newer versions. Doing so improves the quality of the collection, eliminates chaff from search results and improves customer satisfaction. The technical challenge is to use methods that can identify similar documents based on their content alone, without relying on metadata, which may be corrupt or missing.

5.4.3 News Aggregators

Given multiple sources of documents, such as RSS news feeds, several online news agencies etc. news aggregators cluster together similar documents that cover the same material. For instance, several news websites around the world would carry the news of “Germany Lifting the World Cup Football Title in 2014”. We wish to recognize that all these articles are essentially reporting the same underlying story and cluster or aggregate them. However, the core of each newspaper’s page will be the original article. News aggregators, such as Google News, attempt to find all versions of one story and then aggregate them as one article. In order to do this, they must identify when two Web pages are textually similar, although not identical.

5.5

Collaborative Filtering as a Similar-Sets Problem

In this age of the all-pervasive Internet, 24/7 connectivity, we are inundated with choices and options for every activity we perform. What to wear? What movie to view? What stock to buy? What blog and articles do we read? Which holiday destination should be next on our list? And so on. The sizes of these decision domains are frequently massive: Netflix has over 17,000 movies in its selection and Amazon.com has over 4,10,000 titles in its Kindle book store alone. Supporting discovery in information of this magnitude is a significant challenge.

Collaborative filtering methods are based on collecting and analyzing a large amount of information on users’ behaviors, activities or preferences and predicting what users will like based on their similarity to other users. A key advantage of the collaborative filtering approach is that it does

not rely on machine analyzable content and, therefore, it is capable of accurately recommending complex items, such as movies, TV shows, products, etc., without requiring an “understanding” of the item itself.

Collaborative filtering is based on the assumption that people who agreed in the past will agree in the future, and that they will like similar kinds of items as they liked in the past (Fig. 5.3).

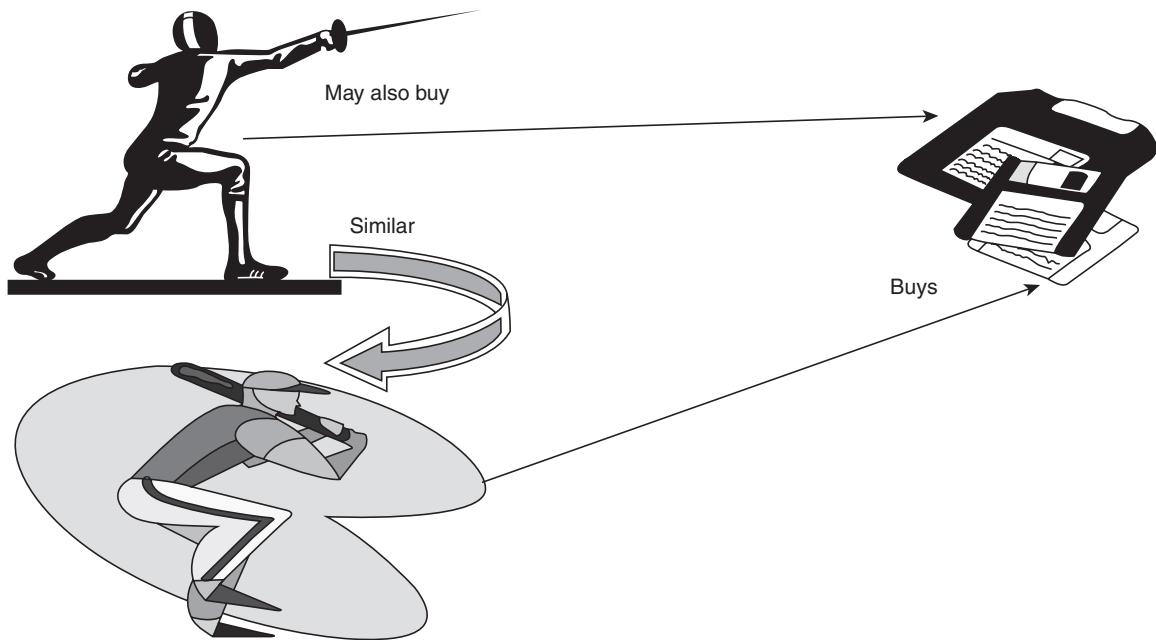


Figure 5.3 Athletes being similar may purchase the same things.

5.5.1 Online Retail

E-commerce recommendation algorithms often operate in a challenging environment. For example:

1. A large on-line retailer might have huge amounts of data, tens of millions of customers and millions of distinct catalog items.
2. Many applications require the results set to be returned in real-time, in no more than half a second, while still producing high-quality recommendations.
3. New customers typically have extremely limited information, based on only a few purchases or product ratings, whereas repeat customers can have a glut of information, based on thousands of purchases and ratings.
4. Customer data is volatile: Each interaction provides valuable customer data, and the algorithm must respond immediately to the new information.

Collaborative filtering generates two well-known types of recommendation algorithms: *item-based recommenders* and *user-based recommenders*. Both styles depend on the concept of a similarity function like Jaccard measure, to measure similarity. User-based recommendation algorithms look at the similarity between users based on their behavior (clicks, purchases, preferences, selections, ratings, etc.), then recommend the same things to the similar users. By contrast, item-based collaborative filtering looks at how users interact with items, such as books, movies, users, news stories, etc., and with that information, recommend items that are more similar. Examples of item based collaborative filtering include Amazon.com, E-Bay, etc. These websites record in their databases every transaction indicating purchases made by customers. The recommendation engine identifies two customers as similar if their sets of purchased items have a high Jaccard Similarity. In the same way, two items that have sets of purchasers with high Jaccard Similarity will be deemed similar. The high volume of data means that even a Jaccard Similarity like 20% might be interesting enough to identify customers with similar tastes. The same observation holds for items; Jaccard similarities need not be very high to be significant (see Fig. 5.4).

Frequently Bought Together

Nikon D3100 SLR (Black) with (AF-S 18-55mm VR Kit Lens)

152 Ratings | 18 Reviews

• 14.2 Megapixels
• CMOS

• Interchangeable lens camera,
SLR
• with 3.0 inch LCD Screen

with 2 year Nikon India Warranty and Free Transit Insurance

Price: Rs.30990 **Rs. 29950**

Discount: Rs. 1000
(Prices are inclusive of all taxes)

FREE Home Delivery

Buy This Now with an EMI Option

Frequently Bought Together With This Camera

- Nikon AF Zoom-Nikkor 70-300mm f/4-5.6G Price: Rs.6185 Rs.5999
- Nikon AF-S DX Nikkor 55-300 mm f/4.5-5.6G Price: Rs.20155 Rs.19550
- Nikon AF-S DX VR Zoom-Nikkor 55-200 mm Price: Rs.14150 Rs.13725
- Nikon AF Nikkor 50mm f/1.8D Lens Price: Rs.6290 Rs.6101

Summary of Camera: Nikon D3100 SLR.

Nikon D3100 is an entry-level DX-format camera, ideal for photographers preparing to refine their skills in DSLR cameras. This camera has a CMOS image sensor of dimensions 23.6 x 15.4 mm, which enables you to capture fabulously images at 14.2 megapixels. The swift EXPEED 2 processing engine of this digital SLR supports an aspect ratio of 3:2. You can record your memories and shoot full HD movies on the Nikon D3100 at 1920 x 1080 resolution (24 fps).

Body

This Nikon DSLR is compact and easy to carry at 445 g (without battery). The rubber panel on the rear of the camera gives you a firm thumb grip, while the ergonomically designed body with rounded edges lets you hold the camera steady. This Nikon digital SLR has a secure Mode dial which features 13 operating modes. The base of this dial has a lever that lets you switch between Single, Continuous, Self-timer and Quiet modes with one simple flick of the finger.

The camera works on a rechargeable Li-Ion EN-EL14 battery capable of clicking 550 images when fully charged.

Figure 5.4 Showing Recommendations (Flipkart.com).

5.6**Recommendation Based on User Ratings**

Applications like MovieLens, NetFlix, TripAdvisor, Yelp, etc. record user ratings for every transaction conducted at their website. Using similarity in the ratings and similarity of customers, new products (movies, holiday destinations, etc.) are recommended to the users.

MovieLens is a movie recommendation website, a free-service provided by GroupLens Research at the University of Minnesota. It says “*You tell us what movies you love and hate. We use that information to generate personalized recommendations for other movies you will like and dislike*” (GroupLens n.d.). MovieLens uses collaborative filtering techniques to generate movie recommendations. It works by matching together users with similar opinions about movies. Each member of the system has a “neighborhood” of other like-minded users. Ratings from these neighbors are used to create personalized recommendations for the target user. We can see movies as similar if they were rented or rated highly by many of the same customers, and see customers as similar if they rented or rated highly many of the same movies.

In MovieLens, there are two classes of entities, *users* and *items*. Users have preferences for certain items and these preferences must be discovered from the data. The data is represented as a *utility matrix*, a value (number of stars) that represents the rating given by that user for that item and is given for each user-item pair. The ratings can take a value between 1 and 5 stars, representing the number of stars that the user gave as a rating for that item.

The matrix is mostly sparse, meaning that most of the entries are unknown. Figure 5.5 shows an example of a utility matrix, depicting four users: Ann(A), Bob (B), Carl(C) and Doug(D). The available movie names are HP1, HP2 and HP3 for the Harry Potter movies, and SW1, SW2 and SW3 for Star Wars movies 1, 2 and 3.

As ratings are 1–5 stars, put a movie in a customer’s set n times if they rated the movie n -stars. Then, use Jaccard Similarity when measuring the similarity of the customers. The Jaccard Similarity for B and C is defined by counting an element n times in the intersection if n is the minimum of the number of times the element appears in B and C. In the union, we count the element the sum of the number of times it appears in B and in C.

	HP1	HP2	HP3	SW1	SW2	SW3
Ann	4			1		
Bob	5	5	4			
Carl				4	5	
Doug		3				3

Figure 5.5 Utility matrix.

Example 3

The Jaccard Similarity of $\{a, a, b, d\}$ and $\{a, a, b, b, b, c\}$ is $3/10$. The intersection counts a twice and b once, so its size is 3. The size of the union of two sets is always the sum of the sizes of the two sets, or 10 in this case. Since the highest possible Jaccard Similarity is $1/2$, the score of 30% is quite good.

It may be difficult to detect the similarity between movies and users because we have little information about movie-item pairs in this sparse-utility matrix. Even if two items belong to the same genre, there are likely to be very few users who bought or rated both. Similarly, even if two users both like a genre or multiple genres, they may not have bought any items to indicate that. One way of dealing with this lack of data is to cluster movies and/or users. For example, cluster similar movies based on their properties. After clustering the movies, we can revise the utility matrix so the columns represent clusters of movies and the entry for User u and Cluster c is the average rating that u gave to the members of Cluster c that u did not individually rate. Note that u may have rated none of the cluster members, in which case the entry for u and c is left blank. Other similarity measures will have to be used to identify similar items. Figure 5.6 shows the revised utility matrix for cluster users and movies

	HP	SW
Ann	4	1
Bob	4.67	
Carl		4.5
Doug	3	3

Figure 5.6 Utility matrix for Clusters.

5.7 Distance Measures

We have seen the importance of finding how similar two items are in the earlier sections. Similarity is a measure of how much alike two data objects are. In a complementary fashion, Similarity in a data-mining context is usually described as a distance measure between the two items. A small distance indicates a high degree of similarity and a large distance indicates a small degree of similarity.

A distance measure indicates the degree of dissimilarity between the two items. Distance is a subjective measure and is highly dependent on the domain and application. Thus, several different types of Distance Measures become relevant in different domains.

We shall discuss a few of the popular Distance Measures and their applications in the following section.

5.7.1 Definition of a Distance Metric

It is a numerical measure of how different two data objects are. It is a function that maps pairs of objects to real values.

1. Is lower when objects are more alike.
2. Minimum distance is 0 when comparing an object with itself.
3. Upper limit varies.

More formally, a distance function d is a distance metric if it is a function from pairs of objects to real numbers such that:

1. $d(x, y) > 0$. **(Non-negativity)**
2. $d(x, y) = 0$ iff $x = y$. **(Identity)**
3. $d(x, y) = d(y, x)$. **(Symmetry)**
4. $d(x, y) < d(x, z) + d(z, y)$. **(Triangle inequality)**

The triangle inequality property guarantees that the distance function is well-behaved. It indicates that the direct connection is the shortest distance between two points. It says, intuitively, that to travel from x to y , we cannot obtain any benefit if we are forced to travel via some particular third point z .

It is useful also for proving properties about the data. For example, suppose we want to find an object that minimizes the sum of distances to all points in a dataset; if we select the best point from the dataset, the sum of distances we get is at most twice that of the optimal point (see Fig. 5.7).

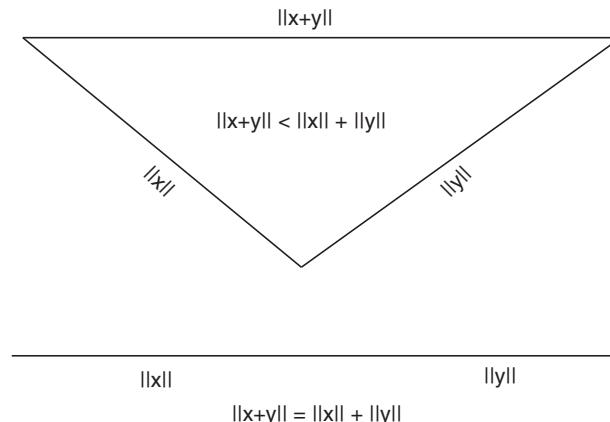


Figure 5.7 Triangle inequality illustration.

5.7.2 Euclidean Distances

The easiest distance measure to compute is called Manhattan Distance or cab-driver distance. In the 2D case, each point is represented by an (x, y) point. So consider two points (x_1, y_1) and (x_2, y_2) . The Manhattan Distance is then calculated by

$$|x_1 - x_2| + |y_1 - y_2|$$

It is called “Manhattan Distance” because it is the distance one would have to travel between points if one were constrained to travel along grid lines, as on the streets of a city such as Manhattan. Manhattan Distance measure is a special case of a distance measure in a “Euclidean Space”. An n -dimensional Euclidean Space is one where each data point is a vector of n real numbers.

The general form of distance measure used for Euclidean spaces is called L_r norm. For any constant r , we can define the L_r -norm to be the distance measure d defined by

$$d([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n]) = \left(\sum_{i=1}^n |x_i - y_i|^r \right)^{1/r}$$

This general form of the distance measure is called the **Minkowski measure**. Manhattan Distance is a special case of the Minkowski measure also called the **$L1$ -norm**. The conventional distance measure in this space, which we shall refer to as the **$L2$ -norm**, is defined as

$$d([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n]) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

This is also called as the **Euclidean Distance**. This measure is derived from the Pythagoras theorem of straight line distance. By taking the positive square root, we see that the distance can never be negative. When the two points are identical, we take the distance to be zero. Because of the square of the differences, the measure is symmetric, that is, $(x_i - y_i)^2 = (y_i - x_i)^2$. The Pythagoras theorem

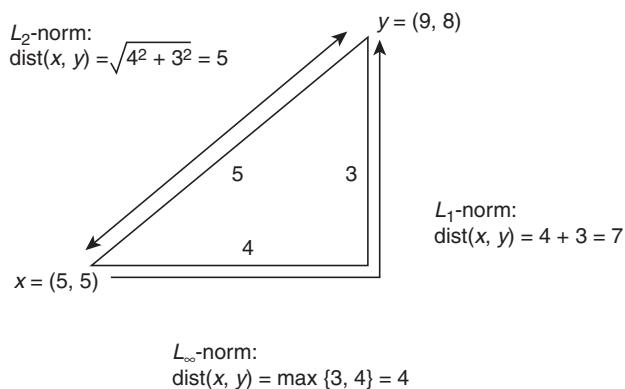


Figure 5.8

itself provides proof of the triangle inequality. Thus, we see that Euclidean Distance is a true-distance measure.

An interesting distance measure is the L^∞ -norm, which is the limit as r approaches infinity of the Lr -norm. As r gets larger, only the dimension with the largest difference matters, so formally, the L^∞ -norm is defined as the maximum of $|x_i - y_i|$ over all dimensions i . An example is shown in Fig. 5.8.

Example 4

Consider the data matrix with four points x_1, x_2, x_3, x_4 with two attributes each in Fig. 5.9. The matrices below indicate the computations of the Manhattan and Euclidean distances.

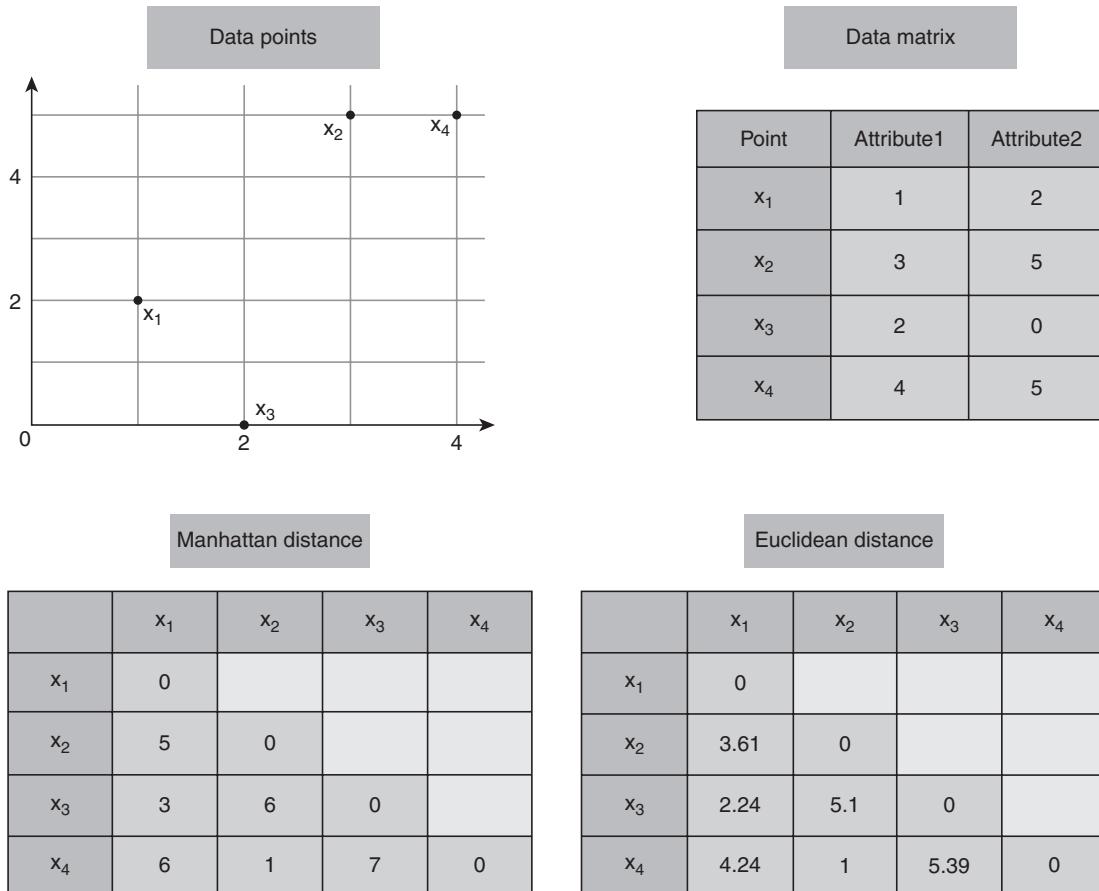


Figure 5.9

5.7.3 Jaccard Distance

The **Jaccard Distance**, which measures dissimilarity between the sample sets, is complementary to the Jaccard coefficient and is obtained by subtracting the Jaccard coefficient from 1, or equivalently, by dividing the difference of the sizes of the union and the intersection of the two sets by the size of the union:

$$d_J(A, B) = 1 - J(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

Jaccard Distance has all the constraints of a distance measure:

1. Since the size of intersection is always less than or equal to the size of the union, Jaccard Distance satisfies the non-negative constraints.
2. The size of the union and the intersection of the two sets can never be same at the same time except the case when both the sets are same. Jaccard Similarity is 1 only when the same sets are used. Further, Jaccard Distance is strictly positive.
3. Union and intersection of two sets are always symmetric; Jaccard Distance satisfies symmetric axiom.
4. Jaccard Similarity always satisfies triangular inequality, and therefore, so does Jaccard Distance.

5.7.4 Cosine Distance

The Cosine Distance between two points is the angle formed between their vectors. The angle always lies in between 0° and 180° regardless of the number of dimensions. The Cosine Distance is measurable in Euclidean spaces of any dimensional spaces where vectors of points in spaces are integer or Boolean components. In this case, points can be considered as directions. Then the cosine distance between the two points is the angle that the vectors to those points make. This angle will be in the range $0^\circ - 180^\circ$, regardless of how many dimensions the space has.

We can calculate the Cosine Distance by first computing the cosine of the angle, and then applying the arc-cosine function to translate to an angle in the $0^\circ - 180^\circ$ range. The cosine of the angle between the two vectors is calculated as the dot product of vectors by dividing with the L_2 -norms of x and y . Figure 5.10 depicts the computation of the Cosine Distance between two documents D1 and D2.

Cosine Distance is a distance measure:

1. Since the values are taken in the range of $(0, 180)$, so there cannot be any negative distances.
2. Angle between the two vectors is 0 only if they are in the same direction.

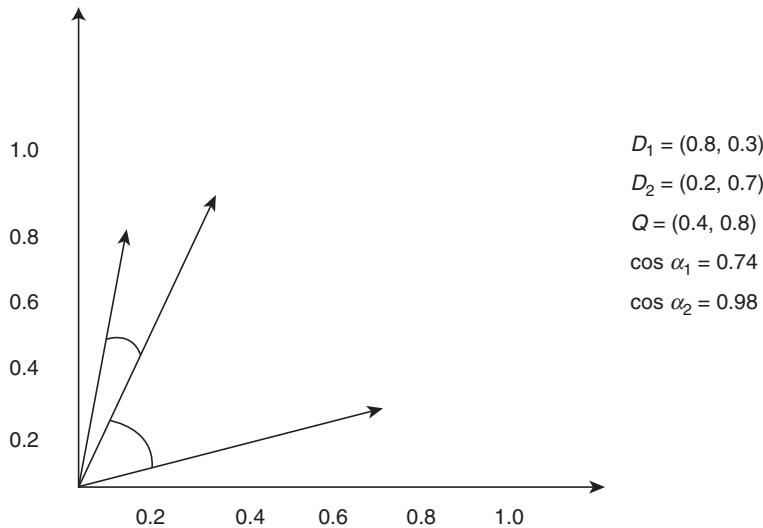


Figure 5.10 Cosine Distance between two documents.

3. Angle between the two vectors satisfies symmetry. The angle between x and y is the same as the angle between y and x .
4. Cosine Distance also satisfies triangle inequality. The triangle inequality is best argued by physical reasoning. One way to rotate from x to y is to rotate to z and then to y . The sum of those two rotations cannot be less than the rotation directly from x to y .

Example 5

Cosine Similarity

Find the similarity between documents 1 and 2.

$$D1 = (5, 0, 3, 0, 2, 0, 0, 2, 0, 0)$$

$$D2 = (3, 0, 2, 0, 1, 1, 0, 1, 0, 1)$$

Solution

$$D1 \cdot D2 = 5 * 3 + 0 * 0 + 3 * 2 + 0 * 0 + 2 * 1 + 0 * 1 + 0 * 1 + 2 * 1 + 0 * 0 + 0 * 1 = 25$$

$$\|D1\| = (5^2 + 0^2 + 3^2 + 0^2 + 2^2 + 0^2 + 0^2 + 2^2 + 0^2 + 0^2)^{0.5} = (42)^{0.5} = 6.481$$

$$\|D2\| = (3^2 + 0^2 + 2^2 + 0^2 + 1^2 + 1^2 + 0^2 + 1^2 + 0^2 + 1^2)^{0.5} = (17)^{0.5} = 4.12$$

$$\cos(D1, D2) = 0.94$$

The angle ϕ that gives cosine value as 0.94 is the cosine distance between the two vectors.

5.7.5 Edit Distance

Edit Distance is used for the string points in vector space. Consider the two strings:

$$x = x_1 x_2 \dots x_n \quad \text{and} \quad y = y_1 y_2 \dots y_m$$

The minimum number of times the insertions and deletions of single characters is done to convert x to y or y to x is the distance between them. For example, the Edit Distance between “Hello” and “Jello” is 1. The edit distance between “good” and “goodbye” is 3. The Edit Distance between any string and itself is 0. Dynamic programming is used to compute the Edit Distance between two strings. The longest common subsequence (LCS) of x and y can also be used to calculate edit distance. An LCS of x and y is a string that is constructed by deleting positions from x and y , and that is as long as any string that can be constructed that way. The Edit Distance can be obtained by subtracting twice the length of their LCS from the sum of lengths of x and y :

$$d(x, y) = |x| + |y| - 2|\text{LCS}(x, y)|$$

Edit Distance is a distance measure because of the following reasons:

1. To convert string x into string y , number of insertions and deletions can never be negative.
2. Edit Distance between two strings is zero only if the two strings are same.
3. Number of insertions and deletions when converting string x to y or y to x is the same.
4. Number of insertions and deletions when converting string x to z plus z to y will always be greater than or equal to that of converting string x directly to y .

Example 6

Let $x = abcde$ and $y = bcduve$. Turn x into y by deleting a ; then insert u and v after d . **Edit-distance = 3.**

Now $\text{LCS}(x, y) = bcde$. So

$$|x| + |y| - 2|\text{LCS}(x, y)| = 5 + 6 - 2 * 4 = 3$$

5.7.6 Hamming Distance

The Hamming Distance is used for the Boolean vectors, that is, which contain only 0 or 1. The number of items in which the two items differ is the Hamming Distance between them. Hamming Distance is definitely a distance measure.

1. The Hamming Distance can never be negative.
2. The Hamming Distance is zero only if the vectors are identical.
3. The Hamming Distance is same irrespective of the order of the vectors and does not depend on which of two vectors we take first. So, it is symmetrical.
4. The triangle inequality is also satisfied because if x is the number of component in which p and r differ, and y is the number of component in which r and q differ, then p and q cannot differ in more than $x + y$ components.

Example 7

Consider two vectors, $p_1 = 10101$; $p_2 = 10011$. Then

$$d(p_1, p_2) = 2$$

because the bit-vectors differ in the 3rd and 4th positions.

Example 8

Consider two vectors, $x = 010101001$; $y = 010011000$. Hamming Distance = 3; there are three binary numbers different between the x and y .

Summary

- **Jaccard Similarity:** The Jaccard Similarity of sets is the ratio of the size of the intersection of the sets to the size of the union. This measure of similarity is suitable for many applications, including textual similarity of documents and similarity of buying habits of customers.
- **Distance Measures:** A Distance Measure is a function on pairs of points in a space that satisfy certain axioms. The distance between two points is 0 if the points are the same, but greater than 0 if the points are different. The distance is symmetric; it does not matter in which order we consider the two points. A distance measure must satisfy the triangle inequality: the distance between the two points is never more than the sum of the distances between those points and some third point.
- **Euclidean Distance:** The most common notion of distance is the Euclidean Distance in an n -dimensional space. This distance,

sometimes called the **L2**-norm, is the square root of the sum of the squares of the differences between the points in each dimension. Another distance suitable for Euclidean spaces, called Manhattan distance or the **L1**-norm, is the sum of the magnitudes of the differences between the points in each dimension.

- **Jaccard Distance:** One minus the Jaccard Similarity is a distance measure, called the Jaccard Distance.
- **Cosine Distance:** The angle between vectors in a vector space is the Cosine Distance measure. We can compute the cosine of that

angle by taking the dot product of the vectors and dividing by the lengths of the vectors.

- **Edit Distance:** This distance measure applies to a space of strings, and is the number of insertions and/or deletions needed to convert one string into the other. The Edit Distance can also be computed as the sum of the lengths of the strings minus twice the length of the longest common subsequence of the strings.
- **Hamming Distance:** This distance measure applies to a space of vectors. The Hamming Distance between two vectors is the number of positions in which the vectors differ.

Exercises

1. Identify five applications not discussed in this chapter for collaborative filtering.
2. What is the difference between user-based collaborative filtering and item-based collaborative filtering?
3. Find Jaccard distances between the following pairs of set:
 - (a) {1, 2, 3, 4} and {2, 3, 5, 7}
 - (b) {3, 2, 1} and {2, 4, 6}
 - (c) {1, 2, 3, 4} and {5, 6, 7, 8}
4. For each of the following vectors, x and y , calculate the cosine distance measure:
 - (a) $x = (1 \ 1 \ 0 \ 0 \ 0)$, $y = (0 \ 0 \ 0 \ 1 \ 1)$
 - (b) $x = (0 \ 1 \ 0 \ 1 \ 1)$, $y = (1 \ 0 \ 1 \ 0 \ 0)$
 - (c) $x = (0 \ 1 \ 2 \ 4 \ 5 \ 3)$, $y = (5 \ 6 \ 7 \ 9 \ 10 \ 8)$
5. What is the range of values that are possible for the cosine measure? If two objects have a cosine measure of 1, are they identical? Explain.
6. Compute the Hamming distance between the following:
 - (a) $x = 0101010001$ and $y = 0100011000$
 - (b) $x = 1111111100$ and $y = 0001111111$
7. Suppose that you are comparing how similar two organisms of different species are in terms of the number of genes they share. Describe which measure, Hamming or Jaccard, you think would be more appropriate for comparing the genetic makeup of two organisms. Explain. (Assume that each animal is represented as a binary vector, where each attribute is 1 if a particular gene is present in the organism and 0 otherwise.)
8. Find edit distance between the following pair of strings:

- (a) “abcdef” and “bcdesg”
- (b) “abcceghi” and “defghsi”
- (c) “aaaaaabbc” and “bbbbbbef”

9. You have studied various similarity and distance measures in this chapter. Give two example applications where the use of a particular distance measure is most appropriate. Explain your answer.

Programming Assignments

1. Consider very large vectors indicating n -dimensional data points. Write programs using the MapReduce framework for computing the following distance measures for any two large vectors. {Assume data to be having more than 1000 dimensions}

- (a) Jaccard Distance
- (b) Cosine Distance

2. Now use the above written codes for distance measures to perform a nearest neighbor search on data points from a standard dataset.

References

1. Junfeng He (2014). Large Scale Nearest Neighbor Search – Theories, Algorithms, & Applications. Columbia University Academic Commons, <http://dx.doi.org/10.7916/D83776PG>.
2. A. Andoni (2009). Nearest Neighbor Search: The Old, the New, and the Impossible. PhD thesis, Massachusetts Institute of Technology.
3. G. Linden, B. Smith, J. York (2003), “Amazon.com Recommendations: Item-to-Item Collaborative Filtering,” *IEEE Internet Computing*, Jan./Feb.
4. Michael D. Ekstrand, John T. Riedl, Joseph A. Konstan (2011). Collaborative Filtering Recommender Systems, Foundations and Trends in Human-Computer Interaction, v.4 n.2, p.81–73, February 2011 [doi>[10.1561/1100000009](https://doi.org/10.1561/1100000009)].
5. T. Brants and R. Stolle (2002). Find similar documents in document collections. In Proceedings of the Third International Conference on Language Resources and Evaluation (LREC-2002), Las Palmas, Spain, June 2002.
6. J. Kasprzak, M. Brandejs and M. Krcaron. Finding Plagiarism by Evaluating Document Similarities, *Proc. SEPLN*, pp. 24–28.
7. G. Forman, K. Eshghi, S. Chiocchetti (2005). Finding similar files in large document repositories, in KDD ‘05: Proceeding of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, pp. 394–400.
8. Anand Rajaraman and Jeffrey David Ullman (2011). Mining of Massive Datasets. Cambridge University Press, New York, NY.

9. Tan P-N, M. Steinbach, V. Kumar. Introduction to Data Mining. Pearson Addison-Wesley.
10. M. Regelson, D. C. Fain (2006). Predicting Click-Through Rate Using Keyword Cluster.
11. V. Abhishek, K. Hosanagar (2007). Keyword Generation for Search Engine Advertising Using Semantic Similarity Between Terms, Proceedings of the Ninth International Conference on Electronic Commerce, pp. 89–94.

6

Mining Data Streams

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Learn and understand the concept of a data stream.
- Become aware of several real-life data stream applications.
- Understand the standard model for a stream-based management system and be able to identify the issues and challenges a stream model presents.
- Understand the importance of sampling and get introduced to several stream sampling techniques.
- Get introduced to the Bloom Filter and its potential applications.
- Learn about several popular stream-based algorithms like Counting Distinct Elements in a Stream, Counting Ones in a Window.
- Learn the Datar–Gionis–Indyk–Motwani algorithm and how it can be used for query processing in a stream.

6.1

Introduction

In the recent years, a new class of data-intensive applications has become widely recognized, that is, applications in which the data is modeled best not as persistent relational databases but rather as transient *data streams*. Data stream real-time analytics are needed to manage the data currently generated, at an ever increasing rate from these applications. Examples of such applications include financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks, call detail records, email, blogging, twitter posts and others.

In the data stream model, individual data items may be relational tuples, for example, network measurements, call records, web page visits, sensor readings and so on. However, their continuous arrival in multiple, rapid, time-varying, possibly unpredictable and unbounded streams appear to yield some fundamentally new research problems.

In all of the applications cited above, it is not feasible to simply load the arriving data into a traditional database management system (DBMS) and operate on it there. Traditional DBMSs are not designed for rapid and continuous loading of individual data items, and they do not directly support

the *continuous queries* that are typical of data stream applications. Further, the data in a data stream is lost forever if it is not processed immediately or stored. Moreover, the data in most data streams arrive so rapidly that it is not feasible to store it all in active storage (i.e., in a conventional database), and then interact with it at the time of our choice.

Thus, algorithms that process them must do so under very strict constraints of space and time. Consequently, data streams pose several challenges for data mining algorithm design. First, algorithms must make use of limited resources (time and memory). Second, they must deal with data whose nature or distribution changes over time.

6.2 Data Stream Management Systems

Traditional relational databases store and retrieve records of data that are static in nature. Further these databases do not perceive a notion of time unless time is added as an attribute to the database during designing the schema itself. While this model was adequate for most of the legacy applications and older repositories of information, many current and emerging applications require support for online analysis of rapidly arriving and changing data streams. This has prompted a deluge of research activity which attempts to build new models to manage streaming data. This has resulted in data stream management systems (DSMS), with an emphasis on continuous query languages and query evaluation.

We first present a generic model for such a DSMS. We then discuss a few typical and current applications of the data stream model.

6.2.1 Data Stream Model

A data stream is a real-time, continuous and ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is not possible to control the order in which the items arrive, nor it is feasible to locally store a stream in its entirety in any memory device. Further, a query over streams will actually run continuously over a period of time and incrementally return new results as new data arrives. Therefore, these are known as long-running, continuous, standing and persistent queries.

As a result of the above definition, we have the following characteristics that must be exhibited by any generic model that attempts to store and retrieve data streams.

1. The data model and query processor must allow both order-based and time-based operations (e.g., queries over a 10 min moving window or queries of the form which are the most frequently occurring data before a particular event and so on).
2. The inability to store a complete stream indicates that some approximate summary structures must be used. As a result, queries over the summaries may not return exact answers.
3. Streaming query plans must not use any operators that require the entire input before any results are produced. Such operators will block the query processor indefinitely.

4. Any query that requires backtracking over a data stream is infeasible. This is due to the storage and performance constraints imposed by a data stream. Thus any online stream algorithm is restricted to make only one pass over the data.
5. Applications that monitor streams in real-time must react quickly to unusual data values. Thus, long-running queries must be prepared for changes in system conditions any time during their execution lifetime (e.g., they may encounter variable stream rates).
6. Scalability requirements dictate that parallel and shared execution of many continuous queries must be possible.

An abstract architecture for a typical DSMS is depicted in Fig. 6.1. An input monitor may regulate the input rates, perhaps by dropping packets. Data are typically stored in three partitions:

1. Temporary working storage (e.g., for window queries).
2. Summary storage.
3. Static storage for meta-data (e.g., physical location of each source).

Long-running queries are registered in the query repository and placed into groups for shared processing. It is also possible to pose one-time queries over the current state of the stream. The query processor communicates with the input monitor and may re-optimize the query plans in response to changing input rates. Results are streamed to the users or temporarily buffered.

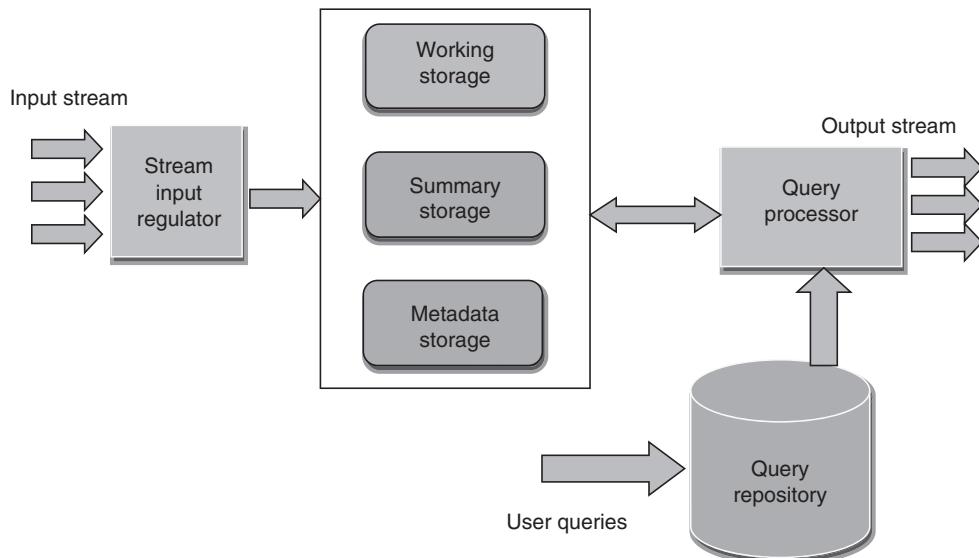


Figure 6.1 Abstract architecture for a typical DSMS.

6.3 Data Stream Mining

Data Stream Mining is the process of extracting useful knowledge from continuous, rapid data streams. Many traditional data mining algorithms can be recast to work with larger datasets, but they cannot address the problem of a continuous supply of data. For example, if a traditional algorithm has learnt and induced a model of the data seen until now, it cannot immediately update the model when new information keeps arriving at continuous intervals. Instead, the entire training process must be repeated with the new examples included.

With big data, this limitation is both undesirable and highly inefficient. Mining big data streams faces three principal challenges: *volume*, *velocity* and *volatility*. Volume and velocity require a high volume of data to be processed in limited time. Starting from the first arriving instance, the amount of available data constantly increases from zero to potentially infinity. This requires incremental approaches that incorporate information as it becomes available, and online processing if not all data can be kept. Volatility, on the other hand, indicates that environment is never stable and has constantly changing patterns. In this scenario, old data is of limited use, even if it could be saved and processed again later. The constantly changing patterns can affect the induced data mining models in multiple ways: *changes in the class label of an existing data variable*, *change in the available feature information*. Both these lead to a phenomenon called *Concept drift*. For example, in a stock market application which labels a particular stock as “hold” or “sell” can change the labels rapidly based on a current stream of input information. Changes in the available feature information can arise when new features become available; for example, a continuous weather forecasting application may now need to consider more attributes because of adding new sensors continuously. Similarly, existing features might need to be excluded due to regulatory requirements, or a feature might change in its scale, if data from a more precise instrument becomes available.

Thus, **Concept drift** is a phenomenon that occurs because of feature changes or changes in behavior of the data itself. Good examples of concept drift include sudden changes in popularity of a movie several days after its release due to good reviews from those who have watched it and also due to changes in price of the movie. This indicates that one important ingredient to mining data streams is *online mining of changes*. This means we are looking to manage data that arrives online, often in real-time, forming a stream which is potentially infinite. Further even if the patterns are discovered in snapshots of data streams the *changes to the patterns* may be more critical and informative. With data streams, people are often interested in mining queries like “*compared to the past few days, what are the distinct features of the current status?*” and “*what are the relatively stable factors over time?*” Clearly, to answer the above queries, we have to examine the changes.

Further mining data streams is challenging in the following two respects. On one hand, random access to fast and large data streams may be impossible. Thus, multi-pass algorithms (i.e., ones that load data items into main memory multiple times) are often infeasible. On the other hand, the exact answers from data streams are often too expensive. The core assumption of data stream processing is that training examples can be briefly inspected a single time only, that is, they arrive in a high speed stream, and then must be discarded to make room for subsequent data. The algorithm processing the stream has no

control over the order of the data seen, and must update its model incrementally as each data element is inspected. Another desirable property is that we must be able to apply the algorithm at any point of time even in between successive arrivals of data elements in the stream.

All these challenges have motivated a whole branch of research, resulting in a new set of algorithms written exclusively for data streams. These algorithms can naturally cope with very large data sizes and can tackle challenging real-time applications not previously tackled by traditional data mining. The most common data stream mining tasks are clustering, classification and frequent pattern mining.

6.4

Examples of Data Stream Applications

We begin by presenting a set of representative data stream applications. This helps us to identify and define a set of query types that a DSMS should support. The reader is referred to the references for many such examples.

6.4.1 Sensor Networks

Sensor networks are a huge source of data occurring in streams. They are used in numerous situations that require constant monitoring of several variables, based on which important decisions are made. In many cases, alerts and alarms may be generated as a response to the information received from a series of sensors. To perform such analysis, aggregation and joins over multiple streams corresponding to the various sensors are required. Some representative queries include the following:

1. Perform a join of several data streams like temperature streams, ocean current streams, etc. at weather stations to give alerts or warnings of disasters like cyclones and tsunami. It can be noted here that such information can change very rapidly based on the vagaries of nature.
2. Constantly monitor a stream of recent power usage statistics reported to a power station, group them by location, user type, etc. to manage power distribution efficiently.

6.4.2 Network Traffic Analysis

Network service providers can constantly get information about Internet traffic, heavily used routes, etc. to identify and predict potential congestions. Streams of network traffic can also be analyzed to identify potential fraudulent activities. For example, consider the case of an intrusion detection system. Here also it may be the case that a situation may drastically change in a limited time. For example, if a particular server on the network becomes a victim of a denial of service attack, that route can become heavily congested within a short period of time.

Example queries may include the following:

1. Check whether a current stream of actions over a time window are similar to a previous identified intrusion on the network.

2. Similarly check if several routes over which traffic is moving has several common intermediate nodes which may potential indicate a congestion on that route.

6.4.3 Financial Applications

Online analysis of stock prices and making hold or sell decisions requires quickly identifying correlations and fast changing trends and to an extent forecasting future valuations as data is constantly arriving from several sources like news, current stock movements, etc. The following are the typical queries:

1. Find all stocks priced between \$50 and \$200, which is showing very large buying in the last one hour based on some federal bank news about tax cuts for a particular industry.
2. Find all stocks trading above their 100 day moving average by more than 10% and also with volume exceeding a million shares.

6.4.4 Transaction Log Analysis

Online mining of web usage logs, telephone call records and automated bank machine transactions are all examples of data streams since they continuously output data and are potentially infinite. The goal is to find interesting customer behavior patterns, identify suspicious spending behavior that could indicate fraud, etc. The following are some examples:

1. Examine current buying pattern of users at a website and potentially plan advertising campaigns and recommendations.
2. Continuously monitor location, average spends etc of credit card customers and identify potential frauds.

6.5 Stream Queries

Queries over continuous data streams have much in common with queries in a traditional DBMS. Two types of queries can be identified as typical over data streams:

1. **One-time queries:** One-time queries are queries that are evaluated once over a point-in-time snapshot of the data set, with the answer returned to the user. For example, a stock price checker may alert the user when a stock price crosses a particular price point.
2. **Continuous queries:** Continuous queries, on the other hand, are evaluated continuously as data streams continue to arrive. The answer to a continuous query is produced over time, always reflecting the stream data seen so far. Continuous query answers may be stored and updated as new data arrives, or they may be produced as data streams themselves.

Typically aggregation queries, such as finding maximum, average, count, etc., are continuous queries where values are stored. For example, the maximum price of a particular stock every hour. We can answer this query by retaining a simple summary: the maximum of all stream elements ever seen up to now. It is not necessary to record the entire stream. When a new stream element arrives, we compare it with the stored maximum, and set the maximum to whichever is larger. We can then answer the query by producing the current value of the maximum. Similarly, if we want the average price of a stock over all time, we have only to record two values: the number of readings ever sent in the stream and the sum of those readings. We can adjust these values easily each time a new reading arrives, and we can produce their quotient as the answer to the query.

Another class of queries may be join queries that are monotonic and may produce rapid, unbounded answers, dictating the stream approach.

We can also make another distinction in stream queries, that between *pre-defined queries* and *ad-hoc queries*. A pre-defined query is one that is supplied to the DSMS before any relevant data has arrived. Pre-defined queries are most commonly continuous queries.

On the other hand, ad-hoc is issued online after the data streams have already begun. Ad-hoc queries can be either one-time queries or continuous queries. Ad-hoc queries are basically questions asked once about the current state of a stream or streams. If we do not store all streams in their entirety, it is difficult to answer arbitrary queries about streams. If we have some idea what kind of queries will be asked, then we can prepare for them by storing appropriate parts or summaries of streams. Ad-hoc queries complicate the design of a DSMS, both because they are not known in advance for the purposes of query optimization, identification of common sub-expressions across queries, etc., and more importantly because the correct answer to an ad-hoc query may require referencing data elements that have already arrived on the data streams (and potentially have already been discarded).

6.6 Issues in Data Stream Query Processing

Query processing in the data stream model of computation comes with its own unique challenges. In this section, we will outline what we consider to be the most interesting of these challenges, and describe several alternative approaches for resolving them.

6.6.1 Unbounded Memory Requirements

Since data streams are potentially unbounded in size, the amount of storage required to compute an exact answer to a data stream query may also grow without bound. Algorithms that use external memory are not well-suited to data stream applications since they do not support continuous queries and are typically too slow for real-time response. New data is constantly arriving even as the old data is being processed; the amount of computation time per data element must be low, or else the latency of the computation will be too high and the algorithm will not be able to keep pace with the data stream.

For this reason, we are interested in algorithms that are able to confine themselves to main memory without accessing disk.

6.6.2 Approximate Query Answering

When we are limited to a bounded amount of memory, it is not always possible to produce exact answers for the data stream queries; however, high-quality approximate answers are often acceptable in lieu of exact answers. Approximation algorithms for problems defined over data streams have been a fruitful research area in the algorithms community in recent years. This work has led to some general techniques for data reduction and synopsis construction, including: sketches, random sampling, histograms and wavelets. For example, recent work develops histogram-based techniques to provide approximate answers for *correlated aggregate queries* over data streams and Gilbert *et al.* present a general approach for building small space summaries over data streams to provide approximate answers for many classes of aggregate queries.

6.6.3 Sliding Windows

One technique for approximate query answering is to evaluate the query not over the entire past history of the data streams, but rather only over sliding windows of the recent data from the streams. For example, only data from the last week could be considered in producing query answers, with data older than 1 week being discarded. Imposing sliding windows on data streams is a natural method for approximation that has several attractive properties. Most importantly, it emphasizes recent data, which in the majority of real-world applications is more important and relevant than the old data: If one is trying in real-time to make sense of network traffic patterns, or phone call or transaction records, or scientific sensor data, then in general insights based on the recent past will be more informative and useful than insights based on stale data. In fact, for many such applications, sliding windows can be a requirement needed as a part of the desired query semantics explicitly expressed as a part of the user's query. A sliding window can be the most recent n elements of a stream, for some n , or it can be all the elements that arrived within the last t time units, for example, 1 month. If we regard each stream element as a tuple, we can treat the window as a relation and query it with any SQL query. Of course, the stream-management system must keep the window fresh, deleting the oldest elements as new ones come in.

Example 1

Suppose a user wanted to compute the average call length, but considering only the 10 most recent long-distance calls placed by each customer. The query can be formulated as follows:

```
SELECT AVG(S.minutes)
FROM Calls S [PARTITION BY S.customer id
ROWS 10 PRECEDING
WHERE S.type = "Long Distance"]
```

Example 2

A slightly more complicated example will be to return the average length of the last 1000 telephone calls placed by “Gold” customers:

```
SELECT AVG(V.minutes)
FROM (SELECT S.minutes
      FROM Calls S, Customers T
     WHERE S.customer id = T.customer id
       AND T.tier = "Gold")
      V [ROWS 1000 PRECEDING]
```

Notice that in these examples, the stream of calls must be joined to the customers relation before applying the sliding window.

6.6.4 Batch Processing, Sampling and Synopses

Another class of techniques for producing approximate answers is to avoid looking at every data element and restrict query evaluation to some sort of sampling or batch processing technique. In batch processing, rather than producing a continually up-to-date answer, the data elements are buffered as they arrive, and the answer to the query is computed periodically. This is an approximate answer since it represents the exact answer at a point in the recent past rather than the exact answer at the present moment. This approach of approximation through batch processing is attractive because it does not cause any uncertainty about the accuracy of the answer, sacrificing timeliness instead. It is also a good approach when data streams are bursty.

Sampling is based on the principle that it is a futile attempt to make use of all the data when computing an answer, because data arrives faster than it can be processed. Instead, some data points must be skipped altogether, so that the query is evaluated over a sample of the data stream rather than over the entire data stream.

For some classes of data stream queries where no exact data structure with the desired properties exist, one can often design an approximate data structure that maintains a small *synopsis* or *sketch* of the data rather than an exact representation, and therefore, is able to keep computation per data element to a minimum.

6.6.5 Blocking Operators

A *blocking query operator* is a query operator that is unable to produce an answer until it has seen its entire input. Sorting is an example of a blocking operator, as are aggregation operators such as SUM, COUNT, MIN, MAX and AVG. If one thinks about evaluating the continuous stream queries using a traditional tree of query operators, where data streams enter at the leaves and final query answers

are produced at the root, then the incorporation of the blocking operators into the query tree poses problems. Since continuous data streams may be infinite, a blocking operator that has a data stream as one of its inputs will never see its entire input, and therefore, it will never be able to produce any output. Clearly, blocking operators are not very suitable to the data stream computation model. Doing away with the blocking operators altogether is not practical, but dealing with them effectively is one of the challenges of data stream computation.

6.7 Sampling in Data Streams

Sampling is a common practice for selecting a subset of data to be analyzed. Instead of dealing with an entire data stream, we select instances at periodic intervals. Sampling is used to compute statistics (expected values) of the stream. While sampling methods reduce the amount of data to process, and, by consequence, the computational costs, they can also be a source of errors. The main problem is to obtain a representative sample, a subset of data that has approximately the same properties of the original data.

6.7.1 Reservoir Sampling

Many mining algorithms can be applied if only we can draw a representative sample of the data from the stream. Imagine you are given a really large stream of data elements (queries on Google searches in May, products bought at Walmart during the Christmas season, names in a phone book, whatever). Your goal is to *efficiently* return a random sample of 1000 elements **evenly distributed** from the original stream. How would you do it? A simple way is to generate random integers between 0 and $(N - 1)$, then retrieving the elements at those indices and you have your answer. To make this sampling without replacement, one simply needs to note whether or not your sample already pulled that random number and if so, choose a new random number. This can make the algorithm pretty expensive if the sample size is very close to N . Further in the case of a data stream one does not know N , the size of the stream in advance and you cannot index directly into it. You can count it, but that requires making two passes of the data that is not possible.

Thus, the general sampling problem in the case of a data stream is, “how to ensure such a sample is drawn uniformly, given that the stream is continuously growing?”. For example, if we want to draw a sample of 100 items and the stream has length of only 1000, then we want to sample roughly one in ten items. But if a further million items arrive, we must ensure that the probability of any item being sampled is more like one in a million. If we retain the same 100 items, then this is very skewed to the prefix of the stream, which is unlikely to be a representative.

Several solutions are possible to ensure that we continuously maintain a uniform sample from the stream. The idea of reservoir sampling dates back to the eighties and before. Reservoir-based methods were originally proposed in the context of one-pass access of data from magnetic storage devices such as tapes. As in the case of streams, the number of records N is not known in advance and the sampling must be performed dynamically as the records from the tape are read. Let us assume that we wish to obtain an unbiased sample of size n from the data stream. In this algorithm, we maintain a reservoir

of size n from the data stream. The first n points in the data streams are added to the reservoir for initialization. Subsequently, when the $(t+1)$ th point from the data stream is received, it is added to the reservoir with probability $n/(t+1)$. In order to make room for the new point, any of the current points in the reservoir are sampled with equal probability and subsequently removed.

It is easiest to describe if we wish to draw a sample of size 1. Here, we initialize the sample with the first item from the stream. We replace the current sample with the i th item in the stream (when it is seen) by throwing some random bits to simulate a coin whose probability of landing “heads” is $1/i$, and keeping the i th item as the sampled item if we observe heads. It is a simple exercise to prove that, after seeing n items, the probability that any of those is retained as the sample is precisely $1/n$. One can generalize this technique to draw a sample of k items, either with replacement (by performing k independent repetitions of the above algorithm) or without replacement (by picking k items and replacing one with the i th with probability $1/i$).

6.7.2 Biased Reservoir Sampling

In many cases, the stream data may evolve over time, and the corresponding data mining or query results may also change over time. Thus, the results of a query over a more recent window may be quite different from the results of a query over a more distant window. Similarly, the entire history of the data stream may not relevant for use in a repetitive data mining application such as classification. The simple reservoir sampling algorithm can be adapted to a sample from a moving window over data streams. This is useful in many data stream applications where a small amount of recent history is more relevant than the entire previous stream. However, this can sometimes be an extreme solution, since for some applications we may need to sample from varying lengths of the stream history. While recent queries may be more frequent, it is also not possible to completely disregard queries over more distant horizons in the data stream. **Biased reservoir sampling** is a bias function to regulate the sampling from the stream. This bias gives a higher probability of selecting data points from recent parts of the stream as compared to distant past. This bias function is quite effective since it regulates the sampling in a smooth way so that the queries over recent horizons are more accurately resolved.

6.7.3 Concise Sampling

Many a time, the size of the reservoir is sometimes restricted by the available main memory. It is desirable to increase the sample size within the available main memory restrictions. For this purpose, the technique of concise sampling is quite effective. Concise sampling exploits the fact that the number of *distinct* values of an attribute is often significantly smaller than the size of the data stream. In many applications, sampling is performed based on a single attribute in multi-dimensional data. For example, customer data in an e-commerce site sampling may be done based on only customer ids. The number of distinct customer ids is definitely much smaller than “ n ” the size of the entire stream.

The repeated occurrence of the same value can be exploited in order to increase the sample size beyond the relevant space restrictions. We note that when the number of distinct values in the stream is smaller than the main memory limitations, the entire stream can be maintained in main memory, and therefore, sampling may not even be necessary. For current systems in which the memory sizes

may be of the order of several gigabytes, very large sample sizes can be main memory resident, as long as the number of distinct values does not exceed the memory constraints. On the other hand, for more challenging streams with an unusually large number of distinct values, we can use the following approach.

1. The sample is maintained as a set S of <value, count> pairs.
2. For those pairs in which the value of count is one, we do not maintain the count explicitly, but we maintain the value as a *singleton*.
3. The number of elements in this representation is referred to as the footprint and is bounded above by n .
4. We use a *threshold parameter* τ that defines the probability of successive sampling from the stream. The value of τ is initialized to be 1.
5. As the points in the stream arrive, we add them to the current sample with probability $1/\tau$.
6. We note that if the corresponding value count pair is already included in the set S , then we only need to increment the count by 1. Therefore, the footprint size does not increase.
7. On the other hand, if the value of the current point is distinct from all the values encountered so far, or it exists as a singleton then the footprint increases by 1. This is because either a singleton needs to be added, or a singleton gets converted to a value count pair with a count of 2.
8. The increase in footprint size may potentially require the removal of an element from sample S in order to make room for the new insertion.
9. When this situation arises, we pick a new (higher) value of the threshold τ' , and apply this threshold to the footprint in repeated passes.
10. In each pass, we reduce the count of a value with probability (τ/τ') , until at least one value-count pair reverts to a singleton or a singleton is removed.
11. Subsequent points from the stream are sampled with probability $(1/\tau')$.

In practice, τ' may be chosen to be about 10% larger than the value of τ . The choice of different values of τ provides different tradeoffs between the average (true) sample size and the computational requirements of reducing the footprint size.

6.8

Filtering Streams

Many queries on data streams translate to some form of the reduction of the initial amount of data in the stream. In the previous section, one form of reduction sampling has been discussed. Another technique “Filtering” tries to observe an infinite stream of data, look at each of its items and quantify whether the item is of interest and should be stored for further evaluation.

This section discusses various filtering methods that can optimally be used in the scenario of data streams. Of particular interest is the concept of a Bloom filter, a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element belongs to a set. Hashing has been the popular solution to designing algorithms that approximate some value that we would like to maintain. The method of Bloom filters is a way to use a constant number of hash functions to get a vast improvement in the accuracy of the algorithm.

When Bloom filters were first introduced in the 1970s, they were immediately applied to many existing applications where memory was a constraint. After that, the idea lay dormant and no more research was done regarding them until as recently as 1998. Now that memory has become a constraint once again with streaming applications that need to deal with vast quantities of data very second, the idea has caught on once again and there has been much research done extending the idea of the Bloom filter.

6.8.1 An Example

Let us discuss an actual example that illustrates the problem. Consider the popular web browser, Google Chrome. Chrome needs to store a blacklist of dangerous URLs. Each time a user is about to navigate to a new page, if the URL matches any one from the blacklist it needs to be blocked.

The number of such malicious sites will be quite large say around a million. Further, the minimum and maximum length of a URL is 2 characters and 2083 characters, respectively (Internet Explorer). It is not feasible to store a million such URLs in main memory. Thus, we can either use disk accesses to determine whether the current URL is to be allowed or not. Considering the large numbers of users accessing websites using Chrome continuously, this is indeed a data stream with a very high velocity, thus accessing secondary memory for checking whether to block the address or not is definitely not practical. Thus, it is absolutely essential to devise a method that requires no more main memory than we have available, and yet will filter most of the undesired stream elements.

Let us assume we want to use about one megabyte of available main memory to store the blacklist. As Bloom filtering, uses that main memory as a bit array being able to store eight million bits. A hash function “ h ” is devised that maps each URL in the blacklist to one of eight million buckets. That corresponding bit in the bit array is set to 1. All other bits of the array remain 0.

Since there are one million members in the blacklist, approximately 1/8th of the bits will be 1. The exact fraction of bits set to 1 will be slightly less than 1/8th, because it is possible that two URLs could hash to the same bit. When a stream element arrives, we hash its URL. If the bit to which it hashes is 1, then we need to further check whether this URL is safe for browsing or not. But if the URL hashes to a 0, we are certain that the address is not in the blacklist so we can ignore this stream element.

Unfortunately, some non-malicious URLs could still hash to a 1 bit and could be blocked. This could happen with approximately 1/8 of the URLs. Nevertheless, blocking all blacklisted URLs at the cost of a few false alarms is indeed a significant benefit. Further all comparisons are achieved by using only the main memory bit map. Any URL is first checked against a local Bloom filter, and only if the Bloom filter returned a positive result is a full check of the URL performed (and the user

warned, if that too returned a positive result). Several variants of this general Bloom Filter exist with different capabilities.

6.8.2 The Bloom Filter

A **Bloom filter** is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not, thus a Bloom filter has a 100% recall rate. In other words, a query returns either “possibly in set” or “definitely not in set”. Elements can be added to the set, but not removed (though this can be addressed with a “counting” filter). The more elements that are added to the set, the larger the probability of false positives.

The Bloom filter for a set is much smaller than the set itself, which makes it appropriate for storing large amounts of data required when filtering on a data stream in main memory itself.

The advantage of this data structure is that it uses considerably less space than any exact method, but pays for this by introducing a small probability of error. Depending on the space available, this error can be made arbitrarily small.

Let us assume we want to represent n -element sets $S = s_1, s_2, \dots, s_n$ from a very large universe U , with $|U| = u \gg n$.

We want to support insertions and membership queries (as to say “Given $x \in U$ is $x \in S$?”) so that:

1. If the answer is No, then $x \in S$.
2. If the answer is Yes, then x may or may not be in S , but the probability that $x \notin S$ (false positive) is low.

Both insertions and membership queries should be performed in a constant time. A Bloom filter is a bit vector B of m bits, with k independent hash functions h_1, \dots, h_k that map each key in U to the set $R_m = 0, 1, \dots, (m - 1)$. We assume that each hash function h_i maps a uniformly at random chosen key $x \in U$ to each element of R_m with equal probability. Since we assume the hash functions are independent, it follows that the vector $[h_1(x), \dots, h_k(x)]$ is equally likely to be any of the m^k k -tuples of elements from R_m .

Algorithm

- Initially all m bits of B are set to 0.
- Insert x into S . Compute $h_1(x), \dots, h_k(x)$ and set

$$B[h_1(x)] = B[h_2(x)] = \dots = B[h_k(x)] = 1$$

- Query if $x \in S$. Compute $h_1(x), \dots, h_k(x)$.

If $B[h_1(x)] = B[h_2(x)] = \dots = B[h_k(x)] = 1$, then answer Yes, else answer No.

Clearly, if an item is inserted into the filter, it is found when searched for. Hence, there is no false negative error. The only possibility for error is when an item is not inserted into the filter, but each of the locations that the hash functions map to it are all turned on. We will show that this error can be kept small while using considerably less space than any exact method.

6.8.3 Analysis of the Bloom Filter

If a key value is in S , then the element will surely pass through the Bloom filter. However, if the key value is not in S , it might still pass. This is called a false positive. We need to understand how to calculate the probability of a false positive, as a function of n , the bit-array length, m the number of members of S , and k , the number of hash functions. The computation can be done as follows:

1. The probability that one hash does not set a given bit is $(1 - 1/m)$ given the setting in previous section.
2. The probability that it is not set by any of the k hash functions is $(1 - 1/m)^k$.
3. Hence, after all n elements of S have been inserted into the Bloom filter, the probability that a specific bit is still 0 is $f = (1 - 1/m)^{kn} = e^{-kn/m}$. (Note that this uses the assumption that the hash functions are independent and perfectly random.)
4. The probability of a false positive is the probability that a specific set of k bits are 1, which is

$$(1 - (1 - 1/m)^{kn})^k = (1 - e^{-kn/m})^k$$

5. Thus, we can identify three performance metrics for Bloom filters that can be adjusted to tune its performance. First the computation time (corresponds to the number k of hash functions), second the size (corresponds to the number m of bits), and finally probability of error (corresponds to the false positive rate).
6. Suppose we are given the ratio m/n and want to optimize the number k of hash functions to minimize the false positive rate f . Note that more hash functions increase the precision but also the number of 1s in the filter, thus making false positives both less and more likely at the same time.
7. Let $g = \ln(f) - k \ln(1 - e^{-kn/m})^k$. Suppose $p = e^{-kn/m}$. Then, the derivative of g is

$$\frac{dg}{dk} = \ln(1 - p) + \frac{kn}{m} \cdot \frac{p}{1 - p}$$

To find the optimal k , or right number of hash functions to use, we should equate the above derivative to 0. The solution is

$$k = \left[(\ln 2) \cdot \frac{m}{n} \right]$$

So for this optimal value of k , the false positive rate is

$$\left(\frac{1}{2}\right)^k = (0.6185)^{m/n}$$

or

$$k = \frac{m}{n} \log_e 2$$

As m grows in proportion to n , the false positive rate decreases.

Some reasonable values for m , n and k are:

(a) $m/n = 6$, $k = 4$, $\Pr[\text{False positive}] = 0.05$

(b) $m/n = 8$, $k = 6$, $\Pr[\text{False positive}] = 0.02$

Example 3

Suppose that we want to store a set S of $n = 20$ elements, drawn from a universe of $U = 10,000$ possible keys, in a Bloom filter of exactly $N = 100$ cells, and that we care only about the accuracy of the Bloom filter and not its speed. For this problem size, what is the best choice of the number of cells per key? (i.e., what value of k gives the smallest possible probability that a key not in S is a false positive?) What is the probability of a false positive for this choice of k ?

Solution

The probability that a cell is missed is $(1 - 1/100)^{20k}$, so the probability that it is hit is $1 - (1 - 1/100)^{20k}$ and the probability of a false positive is $(1 - (1 - 1/100)^{20k})^k$. Computationally (can be verified by a program) $\min(1 - (1 - 1/100)^{20k})^k$ for k in range (1, 30) gives the answer (0.09286327705662296,3); that is, setting $k = 3$ gives false positive probability approximately 0.0929. ($k = 4$ is slightly worse: its false positive probability is approximately 0.0932.) For values of k greater than or equal to 30, the probability that there exists a missed cell is at most $100(1 - 1/100)^{20k} < 0.25$, and if there is no missed cell then there is definitely a false positive, so the probability of a false positive is at least 0.75, clearly not the minimum possible. So it's safe to terminate the computational search for the best k at $k = 30$.

Bloom filters have some nice properties that make it ideal for data streaming applications. When an item is inserted into the filter, all the locations can be updated “blindly” in parallel. The reason for this is that the only operation performed is to make a bit 1, so there can never be any data race conditions. Also, for an insertion, there need to be a constant (that is, k) number of writes and no reads. Again, when searching for an item, there are at most a constant number of reads. These properties make Bloom filters very useful for high-speed applications, where memory access can be the bottleneck.

6.9**Counting Distinct Elements in a Stream**

The **count-distinct problem**, known in applied mathematics as the **cardinality estimation problem**, is the problem of finding the number of distinct elements in a data stream with repeated elements. This is a well-known problem with numerous applications. The elements might represent IP addresses of packets passing through a router, unique visitors to a website, distinct elements in a large database, motifs in a DNA sequence, etc. In general, if stream elements are chosen from some universal sets, we would like to know how many different elements have appeared in the stream, counting from either the beginning of the stream or some known time in the past.

6.9.1 Count Distinct Problem

Let us first give a formal description of the problem in the following way:

Problem: Given a stream $X = \langle x_1, x_2, \dots, x_m \rangle \in [n]^m$ of values. Let F_0 be the number of distinct elements in X . Find F_0 under the following constraints on data stream algorithms.

Constraints: The constraints are as follows:

1. Elements in stream are presented sequentially and single pass is allowed.
2. Limited space to operate. Avoid swaps from secondary memory. Expected space complexity $O(\log(\min(n, m)))$ or smaller.
3. Estimation guarantees: With error $\varepsilon < 1$ and high probability.

For example consider the stream with instance, $\{a, b, a, c, d, b, d\}$ $F_0 = 4 = |\{a, b, c, d\}|$.

A naïve solution for this problem would be to use a counter $c(i)$ for each value i of the universe U , and would therefore require $O(1)$ processing time per item, but linear space. If the universe is $[1\dots n]$, a bit vector of size n can be used as the synopsis, initialized to 0, where bit i is set to 1 upon observing an item with value i . However, in many cases, the universe is quite large (e.g., the universe of IP addresses is $n = 2^{32}$), making the synopsis much larger than the stream size N . Alternatively, the count F_0 can trivially be computed exactly in one pass through the data stream, by just keeping track of all the unique values observed in the stream. We can maintain a set of all the unique values seen, which takes $n \log_2 n$ bits because each value is $\log_2 n$ bits. However, F_0 can be as large as n , so again the synopsis is quite large. Over two decades ago, Flajolet and Martin presented the first small space distinct-values estimation algorithm called the Flajolet–Martin (FM) algorithm.

6.9.2 The Flajolet–Martin Algorithm

The principle behind the FM algorithm is that, it is possible to estimate the number of distinct elements by hashing the elements of the universal set to a bit-string that is sufficiently long. This means that the length of the bit-string must be such that there are more possible results of the hash function than there are elements of the universal set.

Before we can estimate the number of distinct elements, we first choose an upper boundary of distinct elements m . This boundary gives us the maximum number of distinct elements that we might be able to detect. Choosing m to be too small will influence the precision of our measurement. Choosing m that is far bigger than the number of distinct elements will only use too much memory. Here, the memory that is required is $O[\log(m)]$.

For most applications, 64-bit is a sufficiently large bit-array. The array needs to be initialized to zero. We will then use one or more adequate hashing functions. These hash functions will map the input to a number that is representable by our bit-array. This number will then be analyzed for each record. If the resulting number contained k trailing zeros, we will set the k th bit in the bit array to one. Finally, we can estimate the currently available number of distinct elements by taking the index of the first zero bit in the bit-array. This index is usually denoted by R . We can then estimate the number of unique elements N to be estimated by $N = 2^R$. The algorithm is as follows:

Algorithm

- Pick a hash function h that maps each of the n elements to at least $\log_2 n$ bits.
- For each stream element a , let $r(a)$ be the number of trailing 0's in $h(a)$.
- Record $R = \text{the maximum } r(a) \text{ seen.}$
- Estimate $= 2^R$.

Example 4

$r(a) = \text{position of first 1 counting from the right; say } h(a) = 12, \text{ then } 12 \text{ is } 1100 \text{ in binary, so } r(a) = 2$

Example 5

Suppose the stream is 1, 3, 2, 1, 2, 3, 4, 3, 1, 2, 3, 1 Let $h(x) = 3x + 1 \bmod 5$.

- So the transformed stream (h applied to each item) is 4, 5, 2, 4, 2, 5, 3, 5, 4, 2, 5, 4
- Each of the above element is converted into its binary equivalent as 100, 101, 10, 100, 10, 101, 11, 101, 100, 10, 101, 100
- We compute $r(a)$ of each item in the above stream: 2, 0, 1, 2, 1, 0, 0, 0, 2, 1, 0, 2
- So $R = \text{maximum } r(a)$, which is 2. Output $2^2 = 4$.

A very simple and heuristic intuition as to why Flajolet–Martin works can be explained as follows:

1. $h(a)$ hashes a with equal probability to any of N values
2. Then $h(a)$ is a sequence of $\log_2 N$ bits, where 2^{-r} fraction of all a 's have a tail of r zeros
 - About 50% of a 's hash to ***0
 - About 25% of a 's hash to **00
 - So, if we saw the longest tail of $r = 2$ (i.e., item hash ending *100) then we have probably seen about four distinct items so far
3. So, it takes to hash about 2^r items before we see one with zero-suffix of length r .

More formally we can see that the algorithm works, because the probability that a given hash function $h(a)$, ends in at least r zeros is 2^{-r} . In case of m different elements, the probability that $R \geq r$ (R is max. tail length seen so far) is given by

$$P(R \geq r) = 1 - (1 - 2^{-r})^m$$

Since 2^{-r} is small, $1 - (1 - 2^{-r})^m \approx 1 - e^{-m2^{-r}}$

1. If $2^r \ll m$, $1 - (1 - 2^{-r})^m \approx 1 - (1 - m2^{-r}) \approx m/2^r \approx 0$.
2. If $2^r \ll m$, $1 - (1 - 2^{-r})^m \approx 1 - e^{-m2^{-r}} \approx 1$

Thus, 2^R will almost always be around m .

6.9.3 Variations to the FM Algorithm

There are reasons why the simple FM algorithm won't work with just a single hash function. The expected value of $[2^R]$ is actually infinite. The probability halves when R is increased to $R + 1$, however, the value doubles. In order to get a much smoother estimate, that is also more reliable, we can use many hash functions. Another problem with the FM algorithm in the above form is that the results vary a lot. A common solution is to run the algorithm multiple times with k different hash-functions, and combine the results from the different runs.

One idea is to take the mean of the k results together from each hash-function, obtaining a single estimate of the cardinality. The problem with this is that averaging is very susceptible to outliers (which are likely here).

A different idea is to use the median which is less prone to be influenced by outliers. The problem with this is that the results can only take form as some power of 2. Thus, no matter how many hash functions we use, should the correct value of m be between two powers of 2, say 400, then it will be impossible to obtain a close estimate. A common solution is to combine both the mean and the median:

1. Create k/l hash-functions and split them into k distinct groups (each of size l).
2. Within each group use the median for aggregating together the l results
3. Finally take the mean of the k group estimates as the final estimate.

Sometimes, an outsized 2^R will bias some of the groups and make them too large. However, taking the median of group averages will reduce the influence of this effect almost to nothing. Moreover, if the groups themselves are large enough, then the averages can be essentially any number, which enable us to approach the true value m as long as we use enough hash functions. Groups should be of size at least some small multiple of $\log_2 m$.

6.9.4 Space Requirements

Observe that as we read the stream, it is not necessary to store the elements seen. The only thing we need to keep in main memory is one integer per hash function; this integer records the largest tail length seen so far for that hash function and any stream element. If we are processing only one stream, we could use millions of hash functions, which is far more than we need to get a close estimate. Only if we are trying to process many streams at the same time would main memory constrain the number of hash functions, we could associate with any one stream. In practice, the time it takes to compute hash values for each stream element would be the more significant limitation on the number of hash functions we use.

6.10 Querying on Windows – Counting Ones in a Window

In many data streaming applications, the goal is to make decisions based on the statistics or models gathered over the “recently observed” data elements. For example, one might be interested in gathering statistics about packets processed by a set of routers over the last day. Moreover, we would like to maintain these statistics in a continuous fashion. This gives rise to the *sliding window model*: data elements arrive at every instant; each data element expires after exactly N time steps; and the portion of data that is relevant to gathering statistics or answering queries is the set of the last N elements to arrive. The sliding window refers to the window of active data elements at a given time instant. Figure 6.2 shows a typical sliding window of size 6.

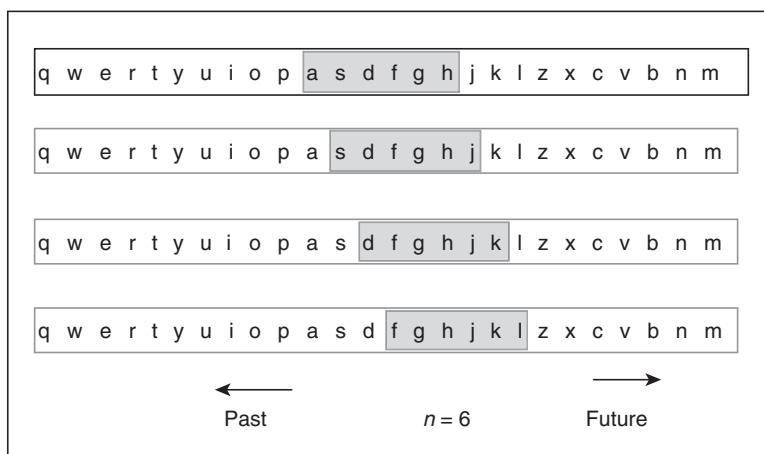


Figure 6.2 Sliding window of size 6.

One basic and simple counting problem whose solution is a pre-requisite for efficient maintenance of a variety of more complex statistical aggregates: Given a stream of bits, maintain a count of the number of 1s in the last N elements seen from the stream. For example, consider an e-retail enterprise that may store for every product \mathbf{X} a 0/1 stream of whether that product was sold in the n th transaction or not. The business may then have queries of the form, how many times has \mathbf{X} in the last k sales?

This problem is termed “Counting Ones in a Window”.

6.10.1 Cost of Exact Counting

It is trivial to see that to able to count exactly the number of 1s in the last k bits for any $k \leq N$, it is necessary to store all N bits of the window. Suppose we have a representation that uses fewer than N bits to represent the N bits in the window. Since there are 2^N sequences of N bits, but fewer than 2^N representations, the pigeon hole principle says that there must be two different bit strings w and x that have the same representation. Thus the query “how many 1s are in the last k bits?” will produce the same answer, whether the window contains w or x , because the algorithm can only see their representations which is the same.

Example 6

If $w = 1001$ and $x = 0101$, then they differ in the third position from the right. Thus any k value lesser than 3, will give same answers.

The real problem is that in most counting applications N is still so large that it cannot be stored on disk, or there are so many streams that windows for all cannot be stored. For example, we are processing 1 billion streams and $N = 1$ billion; thus, we cannot afford to store N bits. This may be true if for instance we are trying to get counts of a particular concept on Twitter to see if its count indicates that is a trending topic currently. As another example, consider a network management application, where a large number of data packets pass through a router every second and we need to compute various statistics for this stream. However, in most applications, it suffices to produce an approximate answer. It is interesting to observe why naive schemes do not suffice for producing approximate answers with low memory requirement. For instance, consider the scheme in which we maintain a simple counter that is incremented upon the arrival of a data element, which is 1. The problem is that an old data element expires at every time instant, but we have no way of knowing whether that was a 0 or 1 and whether we should decrement the counter. Another simple method is random sampling. Just maintaining a sample of the window elements will fail in the case where the 1s are relatively sparse. The sampling method will also fail when the 1s may not arrive in a uniform manner.

6.10.2 The Datar–Gionis–Indyk–Motwani Algorithm

Datar–Gionis–Indyk–Motwani presented an elegant and simple algorithm called DGIM, which uses $O(\log_2 N)$ bits to represent a window of N bits, and enables the estimation of the number of 1s in the window with an error of no more than 50%.

To begin we assume that new data elements are coming from the right and the elements at the left are ones already seen. Note that each data element has an *arrival time*, which increments by one at each arrival, with the leftmost element considered to have arrived at time 1. But, in addition, we employ the notion of a *timestamp*, which corresponds to the position of an *active* data element in the current window. We timestamp the active data elements from right to left, with the most recent element being at position 1.

Clearly, the timestamps change with every new arrival, and we do not wish to make explicit updates. Since we only need to distinguish positions within the window of length N , we shall represent timestamps modulo N , so they can be represented by $\log_2 N$ bits. The timestamp can then be extracted by comparison with the counter value of the current arrival. As mentioned earlier, we concentrate on the 1s in the data stream. If we also store the total number of bits ever seen in the stream (i.e., the most recent timestamp) modulo N , then we can determine from a timestamp modulo N where in the current window the bit with that timestamp is. When we refer to the k th 1, we mean the k th most recent 1 encountered in the data stream. Figure 6.3 illustrates this.

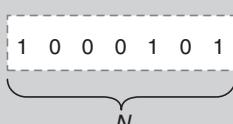
Timestamps	7 6 5 4 3 2 1
Arrival time	20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
Elements	1 0 1 0 1 1 0 0 0 1 0 1 0 1 0 1 1 0 1 1 1 1 1 0 1 

Figure 6.3 Snapshot of DGIM algorithm.

We divide the window into buckets, consisting of:

1. The timestamp of its right (most recent) 1.
2. The number of 1s in the bucket. This number must be a power of 2, and we refer to the number of 1s as the size of the bucket.

For example, in our figure, a bucket with timestamp 1 and size 2 represents a bucket that contains the two most recent 1s with timestamps 1 and 3. Note that the timestamp of a bucket increases as new elements arrive. When the timestamp of a bucket expires that is it reaches $(N + 1)$, we are no longer interested in data elements contained in it, so we drop that bucket and reclaim its memory. If a bucket is still active, we are guaranteed that it contains at least a single 1 that has not expired. Thus, at any instant, there is at most one bucket (the last bucket) containing 1s that may have expired.

To represent a bucket, we need $\log_2 N$ bits to represent the timestamp (modulo N) of its right end. To represent the number of 1s we only need $(\log_2 \log_2 N)$ bits. Since we insist that the number of 1s “ i ” is a power of 2, say 2^j , so we can represent “ i ” by coding j in binary. Since j is at most $\log_2 N$, it requires $\log_2 \log_2 N$ bits. Thus, 0 ($\log N$) bits suffice to represent a bucket.

There are certain constraints that must be satisfied for representing a stream by buckets using the DGIM algorithm.

1. The right end of a bucket always starts with a position with a 1.
2. Number of **1s** must be a power of **2**. That explains the $O(\log \log N)$ bits needed for storing the number of 1s.
3. Either **one** or **two** buckets with the same **power-of-2 number of 1s** exists.
4. **Buckets do not overlap in timestamps.**
5. **Buckets are sorted by size.** Earlier buckets are not smaller than later buckets.
6. Buckets disappear when their end-time is $>N$ time units in the past.

Figure 6.4 indicates an example of a bit stream bucketized using DGIM constraints.

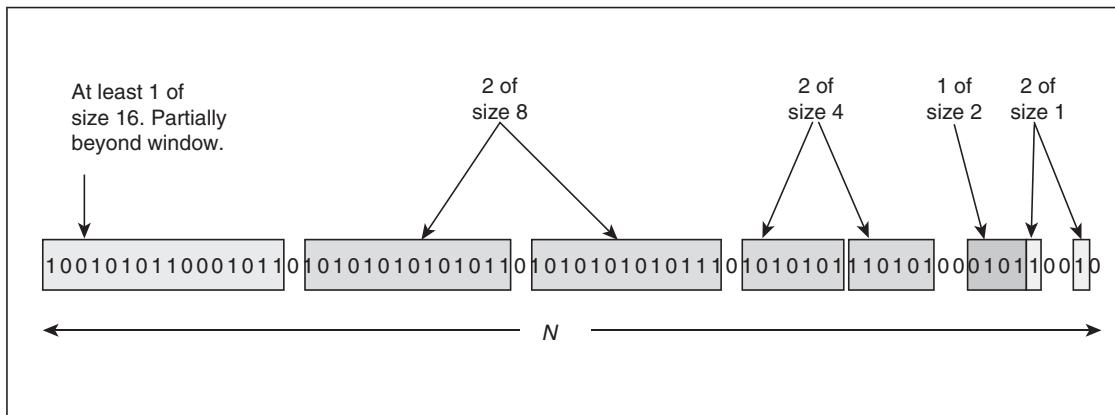


Figure 6.4 Example of a bit stream bucketized using DGIM constraints.

As discussed earlier, each bucket can be represented by $O(\log N)$ bits. For a window of length N , there can be maximum N 1s. Suppose the largest bucket is of size 2^j . Thus j cannot exceed $\log_2 N$. Thus, there are at most two buckets of all sizes from $\log_2 N$ down to 1, and no buckets of larger sizes. So we can conclude that there are $O(\log N)$ buckets. Since each bucket can be represented in $O(\log N)$ bits, the total space required for all the buckets representing a window of size N is $O(\log^2 N)$.

6.10.3 Query Answering in DGIM Algorithm

At any time instant, we may produce an estimate of the number of active 1s as follows. For all but the last bucket, we add the number of 1s that are in them. For the last bucket, let C be the count of the number of 1s in that bucket. The actual number of active 1s in this bucket could be anywhere between 1 and C , and so we estimate it to be $C/2$.

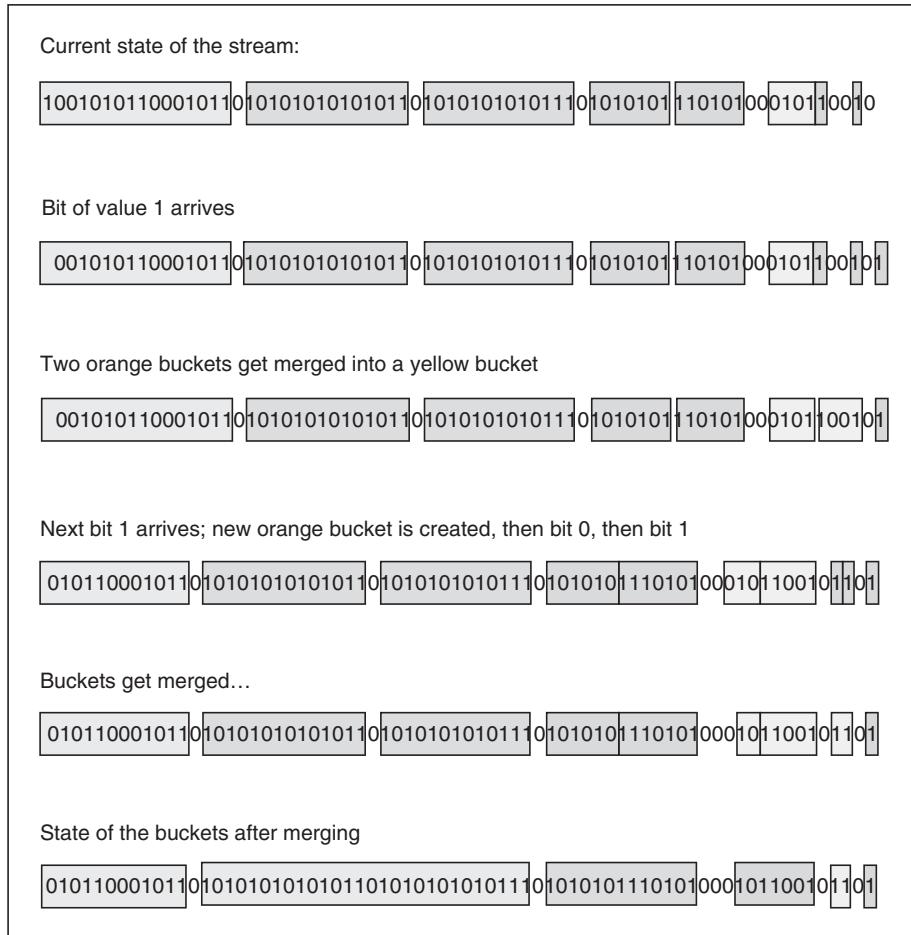


Figure 6.5 Merging Buckets.

For example, for the stream instance shown in Fig. 6.5, the number 1s can be estimated as follows:

1. Estimated: $1 + 1 + 2 + 4 + 4 + 8 + 8 + (1/2)*16 = 36$
2. Exact: $1 + 1 + 2 + 4 + 4 + 8 + 8 + 7 = 35$

In general to estimate the number of 1s in the last k ($1 \leq k \leq N$) bits:

1. Determine bucket b with the earliest timestamp that includes at least some of the N most recent bits.
2. Sum the sizes of all buckets to the right of b . (Recall that “size” means the number of 1s in the bucket).
3. Final estimate: Add half the size of bucket b .

Since we do not know how many 1s of the last bucket are still within the wanted window we are approximating it as half the bucket size.

Example 7

Suppose the stream is that of Fig. 6.5 and $k = 10$. Then the query asks for the number of 1s in the ten rightmost bits, which happen to be 0110010110. Let the current timestamp (time of the rightmost bit) be t . Then the two buckets with one 1, having timestamps $t - 1$ and $t - 2$ are completely included in the answer. The bucket of size 2, with timestamp $t - 4$, is also completely included. However, the rightmost bucket of size 4, with timestamp $t - 8$ is only partly included. We know it is the last bucket to contribute to the answer, because the next bucket to its left has timestamp less than $t - 9$, and thus, is completely out of the window. On the other hand, we know the buckets to its right are completely inside the range of the query because of the existence of a bucket to their left with timestamp $t - 9$ or greater.

Our estimate of the number of 1s in the last ten positions is thus 6. This number is the two buckets of size 1, the bucket of size 2, and half the bucket of size 4 that is partially within range. Of course the correct answer is 5.

Suppose the last bucket has size 2^r , then by assuming 2^{r-1} (i.e., half) of its 1s are still within the window, we make an error of at most 2^{r-1} . Since there is at least one bucket of each of the sizes less than 2^r , the true sum is at least $1 + 2 + 4 + \dots + 2^{r-1} = 2^r - 1$. Thus, error can at most be 50%.

- 1. Case 1:** Estimate is less than exact answer. Worst case: all 1s of B are in range of the query but only half are included. Thus, estimate is at least 50% of the exact answer.
- 2. Case 2:** Estimate is greater than exact answer. Worst case: only the rightmost bit of B (2^r) is in range and only one bucket of each size smaller than B exists.
- 3. Exact:** $1 + 2^{r-1} + 2^{r-2} + \dots + 1 = 2^r$
- 4. Estimated:** $2^{r-1} + 2^{r-1} + 2^{r-2} + \dots + 1 = 2^r + 2^{r-1} - 1$

Thus, the estimate is at most 50% greater than the exact answer always.

6.10.4 Updating Windows in DGIM Algorithm

In a data stream newer bits are arriving at every instant of time. Suppose we have a window of length N properly represented by buckets that satisfy the DGIM constraints. When a new bit comes in, we may need to modify the buckets, so they continue to represent the window and continue to satisfy the DGIM conditions. The following steps describe the Update Algorithm for sliding Windows satisfying DGIM constraints.

1. When a new bit comes in, drop the last (oldest) bucket if its end-time is prior to N time units before the current time
2. **2 cases:** Current bit is **0** or **1**
3. **If the current bit is 0: no other changes are needed**
4. **If the current bit is 1:**
 - Create a new bucket of size **1**, for just this bit. Make the **End timestamp** indicate the **current time**.
 - If there was only one bucket of size 1 before, then nothing more needs to be done. But if there is now **three buckets of size 1**, than **the oldest two must be combined into a new into a bucket of size 2**.
 - If this results in **three buckets of size 2**,**combine the oldest two into a bucket of size 4**
 - **And so on ...**

To combine any two adjacent buckets of the same size, replace them by one bucket of twice the size. The timestamp of the new bucket is the timestamp of the rightmost (later in time) of the two buckets.

6.11 Decaying Windows

Pure sliding windows are not the only way by which the evolution of data streams can be taken into account during the mining process. A second way is to introduce a decay factor into the computation. Specifically, the weight of each transaction is multiplied by a factor of $f < 1$, when a new transaction arrives. The overall effect of such an approach is to create an exponential decay function on the arrivals in the data stream. Such a model is quite effective for evolving data stream, since recent transactions are counted more significantly during the mining process. Specifically, the decay factor is applied only to those itemsets whose counts are affected by the current transaction. However, the decay factor will have to be applied in a modified way by taking into account the last time that the itemset was touched by an update. This approach works because the counts of each itemset reduce by the same decay factor in each iteration, as long as a transaction count is not added to it. Such approach is also applicable to other mining problems where statistics are represented as the sum of decaying values.

We discuss a few applications of decaying windows to find interesting aggregates over data streams.

6.11.1 The Problem of Most-Common Elements

Suppose we have a stream whose elements are the movie tickets purchased all over the world, with the name of the movie as part of the element. We want to keep a summary of the stream that is the most

popular movies “currently.” While the notion of “currently” is imprecise, intuitively, we want to discount the popularity of an older movie that may have sold many tickets, but most of these decades ago. Thus, a newer movie that sold n tickets in each of the last 10 weeks is probably more popular than a movie that sold $2n$ tickets last week but nothing in previous weeks.

One solution would be to imagine a bit stream for each movie. The i th bit has value 1 if the i^{th} ticket is for that movie, and 0 otherwise. Pick a window size N , which is the number of most recent tickets that would be considered in evaluating popularity. Then, use the method of the DGIM algorithm to estimate the number of tickets for each movie, and rank movies by their estimated counts.

This technique might work for movies, because there are only thousands of movies, but it would fail if we were instead recording the popularity of items sold at Amazon, or the rate at which different Twitter-users tweet, because there are too many Amazon products and too many tweeters. Further, it only offers approximate answers.

6.11.2 Describing a Decaying Window

One approach is to re-define the question so that we are not asking for a simple count of 1s in a window. We compute a smooth aggregation of all the 1s ever seen in the stream, but with decaying weights. The further in the past a 1 is found the lesser is the weight given to it. Formally, let a stream currently consist of the elements a_1, a_2, \dots, a_t , where a_1 is the first element to arrive and a_t is the current element. Let c be a small constant, such as 10^{-6} or 10^{-9} . Define the exponentially decaying window for this stream to be the sum

$$\sum_{i=0}^{t-1} a_{t-i} (1-c)^i$$

The effect of this definition is to spread out the weights of the stream elements as far back in time as the stream goes. In contrast, a fixed window with the same sum of the weights, $1/c$, would put equal weight 1 on each of the most recent $1/c$ elements to arrive and weight 0 on all previous elements. This is illustrated in Fig. 6.6.

It is much easier to adjust the sum in an exponentially decaying window than in a sliding window of fixed length. In the sliding window, we have to somehow take into consideration the element that falls out of the window each time a new element arrives. This forces us to keep the exact elements along with the sum, or to use some approximation scheme such as DGIM. But in the case of a decaying window, when a new element a_{t+1} arrives at the stream input, all we need to do is the following:

1. Multiply the current sum by $1 - c$.
2. Add a_{t+1} .

The reason this method works is that each of the previous elements has now moved one position further from the current element, so its weight is multiplied by $1 - c$. Further, the weight on the

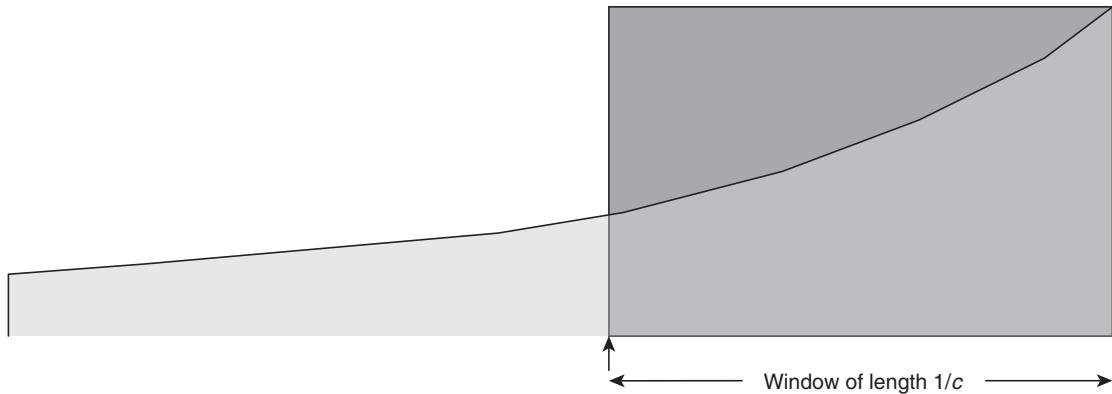


Figure 6.6 Illustrating decaying windows.

current element is $(1 - c)^0 = 1$, so adding $\text{Add } a_{t+1}$ is the correct way to include the new element's contribution.

Now we can try to solve the problem of finding the most popular movies in a stream of ticket sales. We can use an exponentially decaying window with a constant c , say 10^{-9} . We are approximating a sliding window that holds the last one billion ticket sales. For each movie, we can imagine a separate stream with a 1 each time a ticket for that movie appears in the stream, and a 0 each time a ticket for some other movie arrives. The decaying sum of the 1s; thus, it measures the current popularity of the movie.

To optimize this process, we can avoid performing these counts for the unpopular movies. If the popularity score for a movie goes below 1, its score is dropped from the counting. A good threshold value to use is $(1/2)$.

When a new ticket arrives on the stream, do the following:

1. For each movie whose score is currently maintained multiply its score by $(1 - c)$
2. Suppose the new ticket is for movie M. If there is currently a score for M, add 1 to that score. If there is no score for M, create one and initialize it to 1.
3. If any score is below the threshold $1/2$, drop that score.

A point to be noted is that the sum of all scores is $1/c$. Thus, there cannot be more than $2/c$ movies with score of $1/2$ or more, or else the sum of the scores would exceed $1/c$. Thus, $2/c$ is a limit on the number of movies being counted at any time. Of course in practice, the number of actively counted movies would be much less than $2/c$. If the number of items is very large then other more sophisticated techniques are required.

Summary

- **The Stream Data Model:** This model assumes data arrives at a processing engine at a rate that makes it infeasible to store everything in active storage. This has resulted in DSMS, with an emphasis on continuous query languages, and query evaluation.
- **Data Stream Mining:** This is the process of extracting knowledge structures from continuous, rapid data records. Traditional data mining algorithms do not address the problem of a continuous supply of data. With Big Data, this limitation is undesirable highly inefficient. Mining big data streams faces three principal challenges: *volume*, *velocity* and *volatility*.
- **Data Stream Applications and Querying:** Applications exist from all areas like networking to financial applications. Stream queries can be one time or continuous. They can also be distinguished as pre-defined or ad-hoc.
- **Data Stream Issues:** Since a stream cannot be stored in totality due to unbounded memory requirements we need different strategies to deal with them. One strategy to dealing with streams is to maintain summaries or sketches or synopses of the streams, sufficient to give an approximate answer to expected queries about the data. A second approach is to maintaining a sliding window of the most recently arrived data points.
- **Sampling of Streams:** Sampling is a common practice for selecting a subset of data to be analyzed. Instead of dealing with an entire data stream, we select instances at periodic intervals. Sampling is used to compute statistics (expected values) of the stream. To create a sample of a stream that is usable for a class of queries, we identify a set of key attributes for the stream. Several sampling techniques exist like reservoir sampling, Concise samplone etc.
- **Filtering Streams:** Many queries on data streams translate to some form of the reduction of the initial amount of data in the stream. “Filtering” tries to observe an infinite stream of data, look at each of its items and quantify whether the item is of interest and should be stored for further evaluation. **Bloom filters** allow us to filter streams so elements that belong to a particular set are allowed through, while most non-members are deleted. We use a large bit array, and several hash functions. To test a stream element for membership, we hash the element to a set of bits using each of the hash functions, and only accept the element if all these bits are 1.
- **Counting Distinct Elements:** To estimate the number of different elements appearing in a stream, we can hash elements to integers, interpreted as binary numbers. By using many hash functions and combining these estimates, first by taking averages within groups, and then taking the median of the averages, we get a reliable estimate.
- **Estimating the Number of 1s in a Window:** We can estimate the number of 1s in a window of 0s and 1s by grouping the 1s into buckets. Each bucket has a number of 1s that is a power of 2; there are one or two buckets of each size, and sizes never decrease as we go back in time. By recording only the position and size of the buckets, we can represent the contents of a window of size N with $O(\log_2 N)$ space. We use DGIM algorithm

to estimate the number of 1sw. This estimate can never be off by more than 50% of the true count of 1s.

- **Exponentially Decaying Windows:** Rather than fixing a window size, we can imagine that the window consists of all the elements that ever arrived in the stream, but with the element that arrived t time units ago weighted by e^{-ct} for some time-constant c .

Doing so allows us to maintain certain summaries of an exponentially decaying window easily. For instance, the weighted sum of elements can be re-computed, when a new element arrives, by multiplying the old sum by $1 - c$ and then adding the new element. We can use decaying windows to find most popular and most frequent elements in a stream.

Exercises

1. The chapter discusses a few applications of data stream mining. Identify five more applications where data stream mining could be useful.

2. With respect to data stream querying, give examples of

- One time queries.
- Continuous queries.
- Pre-defined queries.
- Ad hoc queries.

(You can identify such queries for the applications you identified for Exercise 1.)

3. Assume we have a data stream of salary information of several individuals across India. The stream elements have the form: (company, department, employee ID, salary). Companies are unique (say Acme Corp, Axis Ltd, Ezee transports, etc.), but department titles are only unique within a single company – different companies may have the same department “Marketing” in different companies. Similarly, employee IDs are also unique within a company, but different companies can use the same IDs to identify their employees. Suppose we want to answer certain queries approximately from a 25%

sample of the data. For each of the following queries, indicate how you would construct the sample so as to have a good estimate:

- For each company, estimate the average number of employees in a department.
 - Estimate the percentage of employees who earn more than Rs. 150,000 per year.
 - Estimate the percentage of departments where all employees make less than Rs. 60,000 a year.
 - Estimate the percentage of companies with more than 30 employees.
 - For each company, estimate the average salary the employees receive across all departments.
- For each of the following applications, explain whether a Bloom filter would be an appropriate choice. Explain your answer.
 - Checking whether an object is stored in a cache in a distributed system.
 - Storing a list of individuals on a “Cannot-Visit” list at a museum.
 - Testing whether a requested webpage has been listed in a “blocked” list of websites.

- (d) In an e-voting system checking whether an attempt is being made to cast a duplicate vote.
5. A Bloom filter with $m = 1000$ cells is used to store information about $n = 100$ items, using $k = 4$ hash functions. Calculate the false positive probability of this instance. Will the performance improve by increasing the number of hash functions from 4 to 5. Explain your answer.
6. Employ the DGIM algorithm. Shown below is a data stream with $N = 24$ and the current bucket configuration. New elements enter the window at the right. Thus, the oldest bit of the window is the left-most bit shown.
- 1 0 1 0 1 1 0 0 0 1 0 1 1 1 0 1 1 0 0
1 0 1 1 0
- (a) What is the largest possible bucket size for $N = 24$?
 - (b) Show one way of how the above initial stream will be divided into buckets
 - (c) What is the estimate of the number of 1's in the latest $k = 14$ bits of this window?
 - (d) The following bits enter the window, one at a time: 1 0 1 0 1 0 1 1. What is the bucket configuration in the window after this sequence of bits has been processed by DGIM?
 - (e) After having processed the bits from (c), what is now the estimate of the number of 1's in the latest $k = 14$ bits of the window?
7. We wish to use the Flajolet–Martin Algorithm to count the number of distinct elements in a stream. Suppose that there 10 possible elements, 1, 2, ..., 10 that could appear in the stream, but only four of them have actually appeared. To make our estimate of the count of distinct elements, we hash each element to a 4-bit binary number. The element x is hashed to $3x + 7$ (modulo 11). For example, element 8 hashes to $3 * 8 + 7 = 31$, which is 9 modulo 11 (i.e., the remainder of $31/11$ is 9). Thus, the 4-bit string for element 8 is 1001.
- A set of four of the elements 1 through 10 could give an estimate that is exact (i.e., 4 distinct elements have appeared), or too high, or too low. List a set of four elements that gives the exact estimate. Briefly describe why this set gives the exact estimate of having seen 4 distinct elements.
8. Suppose a stream consists of the integers 2, 1, 6, 1, 5, 9, 2, 3, 5. Let the hash functions all be of the form $h(x) = ax + b \bmod 16$ for some a and b . You should treat the result as a 4-bit binary integer. Determine the tail length for each stream element and the resulting estimate of the number of distinct elements if the hash function is:
- (a) $h(x) = 2x + 3 \bmod 16$.
 - (b) $h(x) = 4x + 1 \bmod 16$.
 - (c) $h(x) = 5x \bmod 16$.

Programming Assignments

1. Implement the Flajolet–Martin estimate for counting distinct elements from a large data stream using Map reduce. Use the Bloom filter as a data structure.
2. Use the above algorithm to identify the number of distinct customers form any e-commerce dataset (Amazon dataset from

- SNAP dataset; see links to datasets in the Appendix at the end of the book).
3. Implement the DGIM algorithm for counting ones in a sliding window using MapReduce.
 4. Use the DGIM code on any click stream dataset to count the number of hits on a particular item in a given time window.

References

1. Lukasz Golab and M. Tamer Özsu (2003). Issues in Data Stream Management. *SIGMOD Rec.* 32, 2 (June 2003), 5–14. DOI=10.1145/776985.776986 <http://doi.acm.org/10.1145/776985.776986>.
2. Brian Babcock, Shivnath Babu, Mayur Datar, *et al.* (2002). Models and Issues in Data Stream Systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems* (PODS '02). ACM, New York, NY, USA, 1–16. DOI=10.1145/543613.543615 <http://doi.acm.org/10.1145/543613.543615>.
3. A. Rajaraman and J. D. Ullman (2014). *Mining of Massive Datasets*. Cambridge University Press, New York, NY, 2nd edition.
4. G. Cormode (2007). Fundamentals of Analyzing and Mining Data Streams. WORKSHOP ON DATA STREAM ANALYSIS – San Leucio, Italy – March, 15–16.
5. P. B. Gibbons (2007). Distinct-values Estimation over Data Streams. *Data Stream Management: Processing High-Speed Data Streams*. M. Garofalakis, J. Gehrke, and R. Rastogi (Eds.). Springer, New York, NY.
6. Theodore Johnson, S. Muthukrishnan, Irina Rozenbaum (2005). Sampling Algorithms in a Stream Operator. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (SIGMOD '05). ACM, New York, NY, USA, 1–12. DOI=10.1145/1066157.1066159 <http://doi.acm.org/10.1145/1066157.1066159>.
7. C. C. Aggarwal, P. S. Yu (2006). A survey of synopsis construction in data streams. *Data Streams: Models and Algorithms*. Springer-Verlag.
8. R. Bhoraskar, V. Gabale, P. Kulkarni, D. Kulkarni (2013). Importance-Aware Bloom Filter for Managing Set Membership Queries on Streaming Data. In *Proceedings of Fifth International Conference on Communication Systems and Networks (COMSNETS)*, pp. 1–10.
9. Mayur Datar, Aristides Gionis, Piotr Indyk, Rajeev Motwani (2002). Maintaining Stream Statistics over Sliding Windows. *SIAM J. Comput.* 31, 6 (June 2002), 1794–1813. DOI=10.1137/S0097539701398363 <http://dx.doi.org/10.1137/S0097539701398363>.
10. P. Flajolet, G. N. Martin (1983). Probabilistic Counting. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pp. 76–82.

7

Link Analysis

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Understand traditional search engines based on textual analysis.
- Learn about the existence of “Spam” and understand the way search results can be manipulated due to term spam.
- Learn about Google’s pioneering efforts to combat spam by using Link based techniques.
- Clearly understand the PageRank algorithm and its several variants.
- Learn how to efficiently compute PageRank using MapReduce paradigm.
- Learn and apply the HITS search algorithm.
- Become aware of the existence of Link Spam and learn methods to combat them.

7.1

Introduction

The World Wide Web (WWW) is an innovation unsurpassed by any other in this century. It is characterized by its massive size and an absolute lack of coordination in its development. One of the most important features of the WWW is the fact that the Web Users and the Content Creators come from very diverse backgrounds and have different motivations for using the Web. These characteristics of the Web make Web search a challenge as compared to searching “traditional” documents. This has led to a proliferation of “Web Search Engines”. These search engines have disrupted the way the users have engaged with the Web.

At the other end of the spectrum are the spammers, who manipulate the results of the search engine for possible gain which can be commercial or malicious. As search engines become more and more sophisticated to avoid being victims of spam, spammers also are finding innovative ways of defeating the purpose of these search engines.

One such technique used by modern search engines to avoid spam is to analyze the hyperlinks and the graph structure of the Web for ranking of Web search results. This is called Link Analysis. It is one of many factors considered by Web search engines in computing a composite score for a Web page on any given query.

Google was the pioneer in this field with the use of a PageRank measure for ranking Web pages with respect to a user query. Spammers responded with ways to manipulate PageRank too with what is called Link Spam. Techniques like TrustRank were used for detecting Link Spam. Further, various variants of PageRank are also in use to evaluate the Web pages.

This chapter provides the reader with a comprehensive overview of Link Analysis techniques.

7.2 History of Search Engines and Spam

The huge volume of information on the Web is essentially useless unless information can be discovered and consumed by users. Earlier search engines fell into two broad categories:

1. *Full-text index search engines* such as AltaVista, Lycos which presented the user with a keyword search interface. Given the scale of the Web and its growth rate, creating indexes becomes a herculean task.
2. *Taxonomies based search engines* where Web pages were organized in a hierarchical way based on category labels. Example: Yahoo!. Creating accurate taxonomies requires accurate classification techniques and this becomes impossible given the size of the Web and also its rate of growth.

As the Web became increasingly used in applications like e-selling, opinion forming, information pushing, etc., web search engines began to play a major role in connecting users to information they require. In these situations, in addition to fast searching, the quality of results returned by a search engine also is extremely important. Web page owners thus have a strong incentive to create Web pages that rank highly in a search query. This led to the first generation of **spam**, which means “manipulation of Web page content for the purpose of appearing high up in search results for selected keywords”. Earlier search engines came up with several techniques to detect spam, and spammers responded with a richer set of spam techniques.

Spamdexing is the practice of search engine spamming. It is a combination of Spamming with Indexing. Search Engine Optimization (SEO) is an industry that attempts to make a Website attractive to the major search engines and thus increase their ranking. Most SEO providers resort to Spamdexing, which is the practice of creating Websites that will be illegitimately indexed with a high position in the search engines.

Two popular techniques of Spamdexing include “Cloaking” and use of “Doorway” pages.

1. Cloaking is the technique of returning different pages to search engines than what is being returned to the people. When a person requests the page of a particular URL from the Website, the site's normal page is returned, but when a search engine crawler makes the same request, a special page that has been created for the engine is returned, and the normal page for the URL is hidden from the engine – it is cloaked. This results in the Web page being indexed by the search engine under misleading keywords. For every page in the site that needs to be cloaked, another page is created that will cause this page to be ranked highly in a search engine. If more than one

search engine is being targeted, then a page is created for each engine, based on the criteria used by the different engines to rank pages. Thus, when the user searches for these keywords and views the selected pages, the user is actually seeing a Web page that has a totally different content than that indexed by the search engine. Figure 7.1 illustrates the process of cloaking.

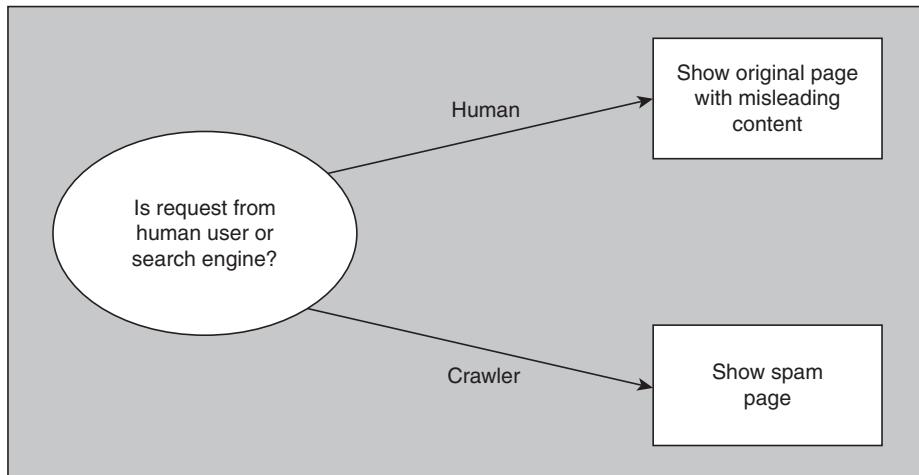


Figure 7.1 The process of cloaking.

Some example situations for cloaking include:

- Rendering a page of HTML text to search engines which guarantee its high ranking. For example, a Web site may be in the business of selling writing instruments. When a search engine browses the site, all it can see would be text indicating history of writing instruments, paper, etc. Thus, the site gets ranked highly for these concepts. The same site then shows a page of images or Flash ads of writing instruments like pens, pencils to users (people) visiting the site.
 - Stuffing relevant extra text or keywords into a page to increase its ranking only when the user-agent requesting the page is a search engine, not a human visitor. Adding a term like “music” to a page several thousand times increases its ranking in the music sub topic, and thus any query about music would lead an user to the pen selling site as first choice.
2. “Doorway” pages are low-quality Web pages created with very little content, but are instead stuffed with very similar keywords and phrases. They are designed to rank highly within the search results, but serve no purpose to visitors looking for information. When a browser requests the doorway page, it is redirected to a page containing content of a more commercial nature. A dummy page with very specific keyword density is created and submitted with a redirect to a commercial site. These pages are generally very ugly and would never pass human scrutiny. The most recent method is to create a single frame and display the entire site through it. Software can create thousands of pages for a single keyword in minutes.

Other techniques used by spammers include meta-tag stuffing, scraper sites, article spinning, etc. The interested reader can go through the references for more information on term spam.

The techniques used by spammers to fool search engines into ranking useless pages higher are called as “Term Spam”. Term spam refers to spam perpetuated because search engines use the visibility of terms or content in a Web page to rank them.

As a concerted effort to defeat spammers who manipulate the text of their Web pages, newer search engines try to exploit the link structure of the Web – a technique known as *link analysis*. The first Web search engine known to apply link analysis on a large scale was Google, although almost all current Web search engines make use of it. But the war between search engines and spammers is far from over as spammers now invest considerable effort in trying to manipulate the link structure too, which is now termed *link spam*.

7.3 PageRank

One of the key concepts for improving Web search has been to analyze the hyperlinks and the graph structure of the Web. Such link analysis is one of many factors considered by Web search engines in computing a composite score for a Web page on any given query.

For the purpose of better search results and especially to make search engines resistant against term spam, the concept of link-based analysis was developed. Here, the Web is treated as one giant graph: The Web page being a node and edges being links pointing to this Web page. Following this concept, the number of inbound links for a Web page gives a measure of its importance. Hence, a Web page is generally more important if many other Web pages link to it. Google, the pioneer in the field of search engines, came up with two innovations based on link analysis to combat term spam:

1. Consider a random surfer who begins at a Web page (a node of the Web graph) and executes a random walk on the Web as follows. At each time step, the surfer proceeds from his current page A to a randomly chosen Web page that A has hyperlinks to. As the surfer proceeds in this random walk from node to node, some nodes will be visited more often than others; intuitively, these are nodes with many links coming in from other frequently visited nodes. As an extension to this idea, consider a set of such random surfers and after a period of time find which Web pages had large number of surfers visiting it. The idea of PageRank is that pages with large number of visits are more important than those with few visits.
2. The ranking of a Web page is not dependent only on terms appearing on that page, but some weightage is also given to the terms used in or near the links to that page. This helps to avoid term spam because even though a spammer may add false terms to one Website, it is difficult to identify and stuff keywords into pages pointing to a particular Web page as that Web page may not be owned by the spammer.

The algorithm based on the above two concepts first initiated by Google is known as PageRank. Since both number and quality are important, spammers just cannot create a set of dummy low-quality Web pages and have them increase the number of in links to a favored Web page.

In Google's own words: *PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the Website is. The underlying assumption is that more important Websites are likely to receive more links from other Websites.*

This section discusses the PageRank algorithm in detail.

7.3.1 PageRank Definition

PageRank is a link analysis function which assigns a numerical weighting to each element of a hyper-linked set of documents, such as the WWW. PageRank helps in “measuring” the relative importance of a document (Web page) within a set of similar entities. The numerical weight that it assigns to any given element E is referred to as the *PageRank of E* and denoted by $\text{PR}(E)$. The PageRank value of a Web page indicates its importance – *higher the value more relevant is this Webpage*.

A hyperlink to a page counts as a vote of support. The PageRank of a page is defined recursively and depends on the number and PageRank metric of all pages that link to it (“incoming links”). A page that is linked to by many pages with high PageRank receives a *high rank itself*.

We can illustrate the simple computation for pageRank using the Web graph model depicted in Fig. 7.2. The figure shows a tiny portion of the Web with five pages 1, 2, 3, 4, and 5. Directed arcs indicate links between the pages. For example, in the figure, the links coming into page 5 are *backlinks* (inlinks) for that page and links going out from page 5 are called *outlinks*. Pages 4 and 1 have a single backlink each, pages 2 and 3 have two backlinks each, and page 5 has three backlinks.

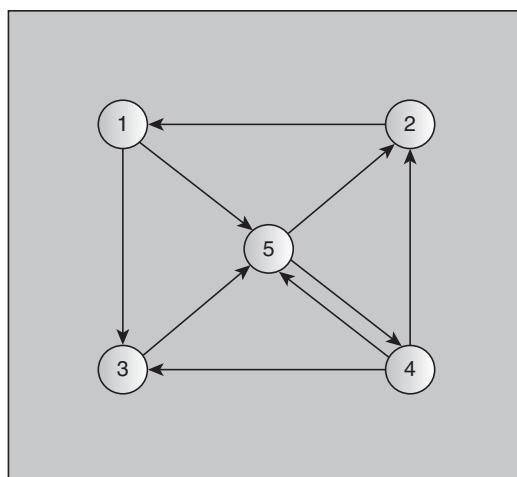


Figure 7.2 A hypothetical web graph.

Let us consider a random surfer who begins at a Web page (a node of the Web graph) and executes a random walk on the Web as follows. At each time step, the surfer proceeds from his current page to a randomly chosen Web page that it has hyperlinks to. So in our figure, the surfer is at a node 1, out of which there are two hyperlinks to nodes 3 and 5; the surfer proceeds at the next time step to one of these two nodes, with equal probabilities 1/2. The surfer has zero probability of reaching 2 and 4.

We can create a “Transition Matrix” “M” of the Web similar to an adjacency matrix representation of a graph, except that instead of using Boolean values to indicate presence of links, we indicate the probability of a random surfer reaching that node from the current node. The matrix M is an $n \times n$ matrix if there are n Web pages. For a Web page pair (P_i, P_j) , the corresponding entry in M (row i column j) is

$$M(i, j) = \frac{1}{k}$$

where k is the number of outlinks from P_j and one of these is to page P_i ; otherwise $M(i, j) = 0$. Thus, for the Web graph of Fig. 7.2, the following will be the matrix M_5 :

$$M_5 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{3} & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & 1 & \frac{1}{3} & 0 \end{bmatrix} \quad (7.1)$$

We see that column 2 represents node 2, and since it has only one outlink to node 1, only first row has a 1 and all others are zeroes. Similarly, node 4 has outlinks to node 2, 3, and 5 and thus has value 1/3 to these nodes and zeroes to node 1 and 4.

7.3.2 PageRank Computation

We know that the PageRank value of a page depends on how important that page is. To compute how important a page is we need to know the probability that a random surfer will land at that page and higher the probability, the more important the page.

To determine the location of a random surfer, we use a column vector v of size n where n is the number of WebPages. The j^{th} component of this vector v is the probability that the surfer is at page j . This probability is nothing but an indication of PageRank value of that page.

- Initially the surfer can be at any of the n pages with probability $1/n$. We denote it as follows:

$$v_0 = \begin{bmatrix} 1/n \\ 1/n \\ \vdots \\ \vdots \\ 1/n \\ 1/n \end{bmatrix}$$

- Consider M , the transition matrix. When we look at the matrix M_5 in Eq. (7.1), we notice two facts: the sum of entries of any column of matrix M is always equal to 1. Further, all entries have values greater or equal to zero. Any matrix possessing the above two properties is called as a matrix of a *Markov chain process*, also called *Markov transition matrix*. At any given instant of time, a process in a Markov chain can be in one of the N states (in a Web set up a state is a node or Web page). Then, the entry m_{ij} in the matrix M gives us the probability that i will be the next node visited by the surfer, provided the surfer is at node j currently. Because of the Markov property, the next node of the surfer only depends on the current node he is visiting. Recall that this is exactly the way we have designed the Transition matrix in Section 7.3.1.
- If vector v shows the probability distribution for the current location, we can use v and M to get the distribution vector for the next state as $x = Mv$. Say currently the surfer is at node j . Then, we have

$$x = M \times v_j = \sum_j m_{ij} \times v_j \quad (7.2)$$

Here v_j is the column vector giving probability that current location is j for every node 1 to n . Thus after first step, the distribution vector will be Mv_0 . After two steps, it will be $M(Mv_0)$. Continuing in this fashion after k steps the distribution vector for the location of the random surfer will be $M^k(Mv_0)$.

- This process cannot continue indefinitely. If a Markov chain is allowed to run for many time steps, the surfer starts to visit certain Web pages (say, a popular stock price indicator site) more often than other pages and slowly the visit frequency converges to fixed, steady-state quantity. Thus, the distribution vector v remains the same across several steps. This final equilibrium state value in v is the PageRank value of every node.
- For a Markov chain to reach equilibrium, two conditions have to be satisfied, the graph must be strongly connected and there must not exist any dead ends, that is, every node in the graph should have at least one outlink. For the WWW this is normally true. When these two conditions are satisfied, for such a Markov chain, there is a unique steady-state probability vector, that is, the principal left eigenvector of the matrix representing the Markov chain.

In our case we have v as the principal eigenvector of matrix M . (An eigenvector of a matrix M is a vector v that satisfies $v = \beta Mv$ for some constant eigenvalue β .) Further, because all columns of the matrix M total to 1, the eigenvalue associated with this principle eigenvector is also 1.

Thus to compute PageRank values of a set of WebPages, we must compute the principal left eigenvector of the matrix M with eigenvalue 1. There are many algorithms available for computing left eigenvectors. But when we are ranking the entire Web, the size of the matrix M could contain a billion rows and columns. So a simple iterative and recursive algorithm called the Power method is used to compute the eigenvalue. It is calculated repetitively until the values in the matrix converge. We can use the following equation to perform iterative computations to calculate the value of PageRank:

$$x_k = M^k(Mv_0) \quad (7.3)$$

After a large number of steps, the values in x_k settle down where difference in values between two different iterations is negligible below a set threshold. At this stage, the values in vector x_k indicate the PageRank values of the different pages. Empirical studies have shown that about 60–80 iterations cause the values in x_k to converge.

Example 1

Let us apply these concepts to the graph of Fig. 7.2 represented by the matrix M_5 as shown before. As our graph has five nodes

$$v_0 = \begin{bmatrix} 1/5 \\ 1/5 \\ 1/5 \\ 1/5 \\ 1/5 \end{bmatrix}$$

If we multiply v_0 by matrix M_5 repeatedly, after about 60 iterations we get converging values as

$$\begin{bmatrix} 1/5 \\ 1/5 \\ 1/5 \\ 1/5 \\ 1/5 \end{bmatrix} \begin{bmatrix} 1/5 & 1/5 & 1/6 & 0.4313 \\ 1/5 & 1/6 & 13/60 & 0.4313 \\ 1/5 & 1/6 & 2/15 & 0.3235 \\ 1/5 & 1/10 & 11/60 & 0.3235 \\ 1/5 & 11/30 & 3/10 & 0.6470 \end{bmatrix} \dots \dots \dots \begin{bmatrix} 0.4313 \\ 0.4313 \\ 0.3235 \\ 0.3235 \\ 0.6470 \end{bmatrix}$$

Thus Page 1 has PageRank 0.4313 as does Page 2. Pages 3 and 4 have PageRank 0.3235 and Page 5 has the highest PageRank of 0.6470.

7.3.3 Structure of the Web

One of the assumptions made for using the concept of Markov processes to compute PageRank is that the entire Web is one giant strongly connected entity. Theoretically a surfer starting at any random Web page can visit any other Web page through a set of links.

But one study conducted in 2000 threw up some surprising results. Researchers from IBM, the AltaVista search engine and Compaq Systems in 2000 conducted a comprehensive study to map the structure of the Web. They analyzed about 200 million Web pages and 1.5 billion hyperlinks. Andrei Broder, AltaVista's Vice President of research at that time and lead author of the study proclaimed that "The old picture – where no matter where you start, very soon you will reach the entire Web – is not quite right."

Older models of the Web had always portrayed its topology as a cluster of sites connected to other clusters and all forming one Strongly Connected Component (SCC). An SCC can be defined as a sub-graph of a graph where a path exists between every pair of nodes in the sub-graph. Further, this is the maximal sub-graph with this property. But the results of the study present a different picture of the Web. The Web, according to the study, consists of three distinct regions:

1. One large portion called as "Core" which is more or less strongly connected so as to form an SCC. Web surfers in the Core can reach any Webpage in the core from any other Webpage in the core. Mostly this is the region of the Web that most surfers visit frequently.
2. A portion of the Web consisted of Web Pages that had links that could lead to the SCC but no path from the SCC led to these pages. This region is called the IN-Component and then pages are called IN pages or "Origination" Pages. New Web Pages or Web Pages forming closed communities belong to this component.
3. Another set of nodes exist that can be reached from the SCC but do not have links that can ultimately lead to a Webpage in the SCC. This is called the "Out" component and the Web Pages "Out" pages or "termination" pages. Many corporate sites, e-commerce sites, etc. expect the SCC to have links to reach them but do not really need links back to the core.

Figure 7.3 shows the original image from the study conducted by Broder *et al.* Because of the visual impact the picture made, they termed this as the "bow-tie picture" of the Web, with the SCC as the central "knot". Figure 7.3 also shows some pages that belong to none of IN, OUT, or SCC. These are further classified into:

1. **Tendrils:** These are pages that do not have any inlinks or outlinks to/from the SCC. Some tendrils consist of pages reachable from the in-component but not SCC and some other tendrils can reach the out-component but not from the SCC.
2. **Tubes:** These are pages that reach from in-component to the out-component without linking to any pages in the SCC.

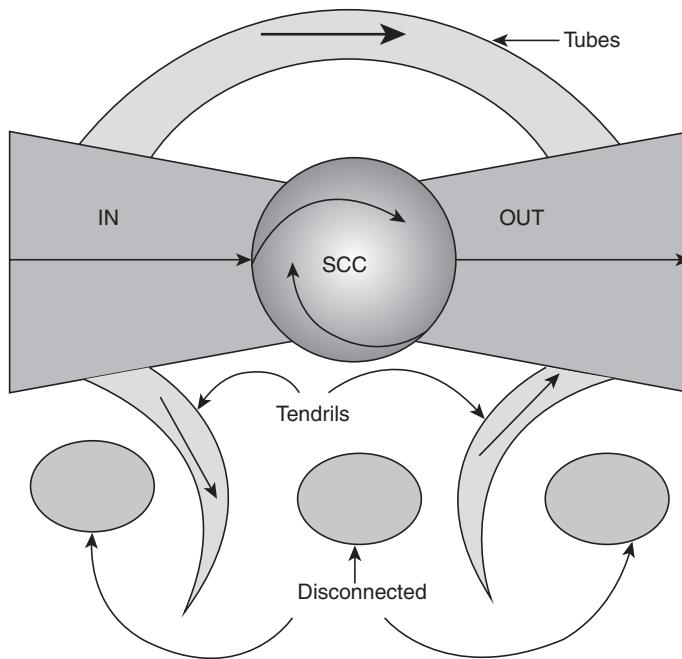


Figure 7.3 Bow-tie structure of the web (Broder *et al.*).

The study further also discussed the size of each region and it was found that perhaps the most surprising finding is the size of each region. Intuitively, one would expect the core to be the largest component of the Web. It is, but it makes up only about one-third of the total. Origination and termination pages both make up about a quarter of the Web, and disconnected pages about one-fifth.

As a result of the bow-tie structure of the Web, assumptions made for the convergence of the Markov process do not hold true causing problems with the way the PageRank is computed. For example, consider the out-component and also the out-tendril of the in-component; if a surfer lands in either of these components he can never leave out, so probability of a surfer visiting the SCC or the in-component is zero from this point. This means eventually pages in SCC and in-component would end up with very low PageRank. This indicates that the PageRank computation must take the structure of the Web into consideration.

There are two scenarios to be taken care of as shown in Fig. 7.4:

1. **Dead ends:** These are pages with no outlinks. Effectively any page that can lead to a dead end means it will lose all its PageRank eventually because once a surfer reaches a page that is a dead end no other page has a probability of being reached.
2. **Spider traps:** These are a set of pages whose outlinks reach pages only from that set. So eventually only these set of pages will have any PageRank.

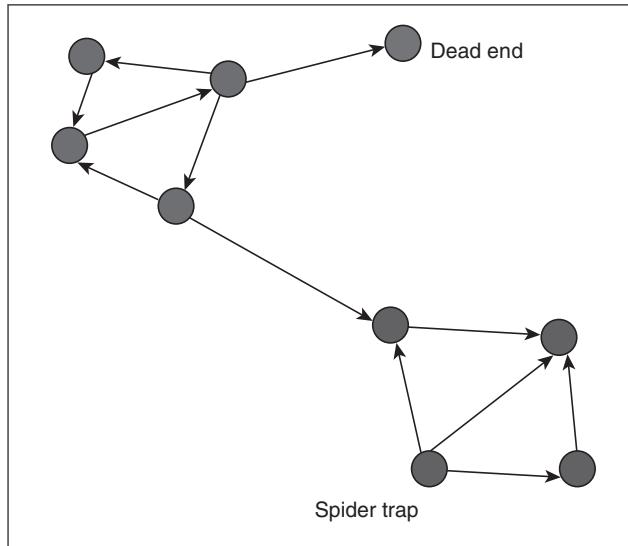


Figure 7.4 Dead end and spider trap.

In both the above scenarios, a method called “taxation” can help. Taxation allows a surfer to leave the Web at any step and start randomly at a new page.

7.3.4 Modified PageRank

One problem we have to solve is the problem of dead ends. When there are dead ends, some columns of the transition matrix M will not sum to 1, they may be 0. For such a matrix where values may sum up to at most 1, the power operation used to generate the ranking vector will result in some or all components reaching 0.

$$M^k v \text{ results in } V \rightarrow 0$$

A simple example can illustrate this. Consider a 3-node network P, Q and R and its associated transition matrix. Since R is dead, eventually all PageRank leaks out leaving all with zero PageRank.

$$\begin{bmatrix} X_P \\ X_Q \\ X_R \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} \begin{bmatrix} 2/6 \\ 1/6 \\ 3/6 \end{bmatrix} \begin{bmatrix} 3/12 \\ 2/12 \\ 7/12 \end{bmatrix} \begin{bmatrix} 5/24 \\ 3/24 \\ 16/24 \end{bmatrix} \dots \dots \dots \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

All the PageRank is trapped in R. Once a random surfer reaches R, he can never leave. Figure 7.5 illustrates this.

We now propose modifications to the basic PageRank algorithm that can avoid the above two scenarios as described in the following subsections.

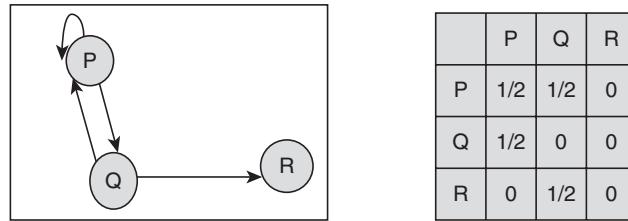


Figure 7.5 A simple Web Graph and its associated transition matrix.

7.3.4.1 Dealing with Dead Ends

We can remove nodes that are dead ends. “Remove” all pages with no outgoing links and remove their in links too. This is a repetitive operation; removing some pages may lead to other pages becoming dead ends. So, recursively we may require to remove more pages ultimately stopping when we land up with an SCC. Name this graph G.

Keep track of the order in which the pages were removed, and in what iteration, because we must eventually restore the graph in reverse order. Now using any method, compute the PageRank of graph G. Now we have to restore the graph by putting back the links and the nodes removed as dead ends. We restore nodes in the reverse order in which they were deleted. The last set of nodes to be deleted will be those whose in links are directly from SCC. All the nodes in the SCC have their PageRank computed.

When we put a dead end page back in, we can compute its PageRank as the sum of PageRanks it receives from each of its inlinks in the SCC. Each inlink will contribute to the dead end page its own PageRank divided by the number of outgoing links for that page.

Example 2

Consider Fig. 7.6.

1. Part (a) shows a portion of a Web graph where A is part of the SCC and B is a dead end. The self-loop of A indicates that A has several links to pages in SCC.
2. In part (b), the dead end B and its links are removed. PageRank of A is computed using any method.
3. In part (c), the dead end last removed, that is B, is put back with its connections. B will use A to get its PageRank. Since A now has two outlinks, its PageRank is divided into 2 and half this rank is propagated to B.
4. In part (d), A has two outlinks and C has three outlinks and both propagate 1/2 and 1/3 of their PageRank values to B. Thus, B gets the final PageRank value as shown. In part (d), A is having a PR value of 2/5 and C has 2/7 leading to B obtaining a PR value of 31/105.

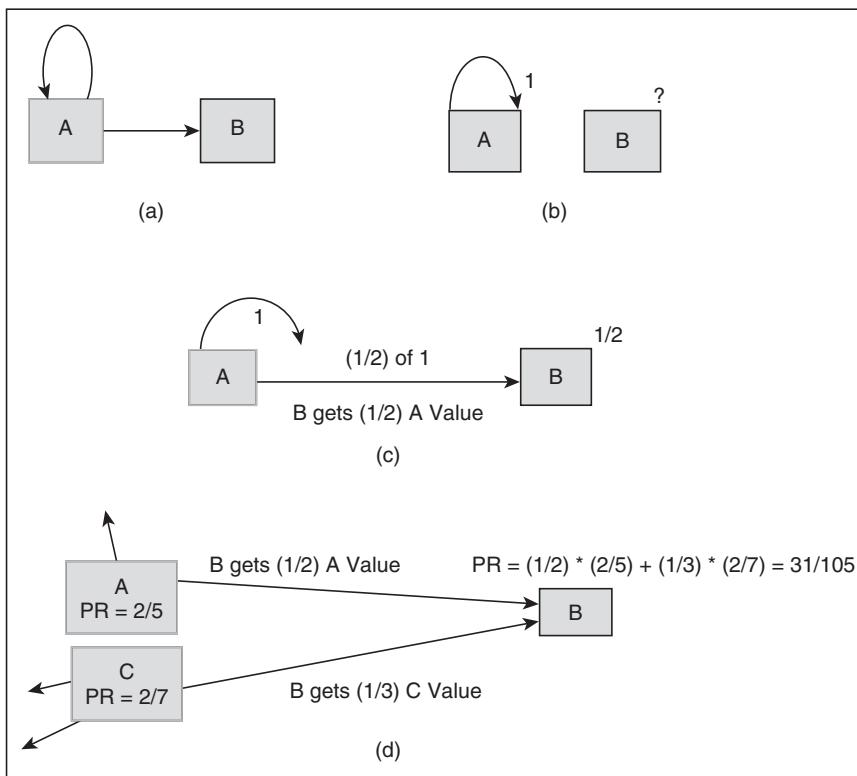


Figure 7.6 Computing PageRank for dead end pages.

7.3.4.2 Avoiding Spider Traps

As we recall, a spider trap is a group of pages with no links out of the group. We have illustrated earlier that in a spider trap all PageRank value will eventually be “trapped” by the group of pages.

To address this issue, we introduce an additional operation for our random surfer: the *teleport* operation. In the teleport operation, the surfer jumps from a node to any other node in the Web graph. Thus, the random surfer, instead of always following a link on the current page, “teleports” to a random page with some probability – Or if the current page has no outgoing links.

In assigning a PageRank score to each node of the Web graph, we use the teleport operation in two ways:

1. When a node has no outlinks, the surfer can invoke the teleport operation with some probability.
2. If a node has outgoing links, the surfer can follow the standard random walk policy of choosing any one of the outlinks with probability $0 < \beta < 1$ and can invoke the teleport operation with probability $1 - \beta$, where β is a fixed parameter chosen in advance.

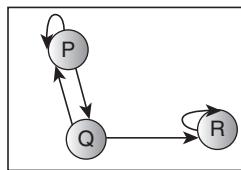
Typical values for β might be 0.8–0.9. So now the modified equation for computing PageRank iteratively from the current PR value will be given by

$$v' = \beta Mv + (1 - \beta)e/n$$

where M is the transition matrix as defined earlier, v is the current PageRank estimate, e is a vector of all ones and n is the number of nodes in the Web graph.

Example 3

Consider the same Web graph example shown earlier in 7.5 (we repeat the graph for clarity). The following shows the new PageRank computations with $\beta = 0.8$.



	P	Q	R
P	1/2	1/2	0
Q	1/2	0	0
R	0	1/2	1

$$(0.8)^* \begin{array}{|c|c|c|c|} \hline & P & Q & R \\ \hline P & 1/2 & 1/2 & 0 \\ \hline Q & 1/2 & 0 & 0 \\ \hline R & 0 & 1/2 & 1 \\ \hline \end{array} + (0.2)^* \begin{array}{|c|c|c|c|} \hline & P & Q & R \\ \hline P & 1/3 & 1/3 & 1/3 \\ \hline Q & 1/3 & 1/3 & 1/3 \\ \hline R & 1/3 & 1/3 & 1/3 \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline & P & Q & R \\ \hline P & 7/15 & 7/15 & 1/15 \\ \hline Q & 7/15 & 1/15 & 1/15 \\ \hline R & 1/15 & 7/15 & 13/15 \\ \hline \end{array}$$

Eventually

$$\begin{bmatrix} X_p \\ X_Q \\ X_R \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1.00 \\ 0.60 \\ 1.40 \end{bmatrix} \begin{bmatrix} 0.84 \\ 0.60 \\ 1.56 \end{bmatrix} \begin{bmatrix} 0.776 \\ 0.536 \\ 1.688 \end{bmatrix} \dots \dots \dots \begin{bmatrix} 7/33 \\ 5/33 \\ 21/33 \end{bmatrix}$$

This indicates that the spider trap has been taken care of. Even though R has the highest PageRank, its effect has been muted as other pages have also received some PageRank.

7.3.5 Using PageRank in a Search Engine

The PageRank value of a page is one of the several factors used by a search engine to rank Web pages and display them in order of ranks in response to an user query. When a user enters a query, also called a search, into Google, the results are returned in the order of their PageRank. The finer details of working of Google's working are proprietary information.

Google utilizes a number of factors to rank search results including standard Information Retrieval measures, proximity, anchor text (text of links pointing to Web pages), and PageRank. According to Sergei and Brin, the designers of PageRank measure and creators of Google, at inception, the ranking of

Web pages by the Google search engine was determined by three factors: Page specific factors, Anchor text of inbound links, PageRank.

Page-specific factors include the body text, for instance, the content of the title tag or the URL of the document. In order to provide search results, Google computes an IR score out of page-specific factors and the anchor text of inbound links of a page. The position of the search term and its weightage within the document are some of the factors used to compute the score. This helps to evaluate the relevance of a document for a particular query. The IR-score is then combined with PageRank to compute an overall importance of that page.

In general, for queries consisting of two or more search terms, there is a far bigger influence of the content-related ranking criteria, whereas the impact of PageRank is more for unspecific single word queries. For example, a query for “Harvard” may return any number of Web pages which mention Harvard on a conventional search engine, but using PageRank, the university home page is listed first.

Currently, it is estimated that Google uses about 250 page-specific properties with updated versions of the PageRank to compute the final ranking of pages with respect to a query.

7.4 Efficient Computation of PageRank

Essentially the implementation of PageRank is very simple. The sheer size of the Web, however, requires much greater care in the use of data structures. We need to perform a matrix and vector multiplication of very large matrices about 50–80 times. This is computationally very expensive.

The algorithm naturally lends itself to a MapReduce (MR) type of scenario. But before we use the MR paradigm, it is also possible to have a more efficient representation of the transition matrix M depicting the Web graph. Further, the use of Combiners with MR helps avoid thrashing.

7.4.1 Efficient Representation of Transition Matrices

Let us consider the structure of a Web graph more precisely. It is typical for a search engine to analyze billions of pages with respect to a query. On an average, a page may have a maximum of 10–15 outlinks. If the page happens to be a doorway page to other pages, a maximum of 25–30 outlinks may exist. When a transition matrix is constructed for computation of PageRank, the matrix will be a billion rows and a billion columns wide. Further, most of the entries in this giant matrix will be zero. Thus, the transition matrix tends to be a very sparse matrix. The storage required to store this matrix is quadratic in the size of the matrix. We can definitely find a more efficient way of storing the transition data by considering only the non-zero values.

One way will be to just list all the non-zero outlinks of a node and their values. This method will need space linear in the number of non-zero entries; 4 byte integers for the node value and 8 byte floating point number for the value of the link. But we can perform one more optimization for a transi-

tion matrix. Each value in the transition matrix is just a fraction that is one divided by the number of outlinks of the node. So, it is sufficient to store along with each node, the number of outlinks, and the destination nodes of these outlinks. This forms an adjacency list like structure as depicted in Fig. 7.6. This figure shows efficient representation of the transition matrix of the Web graph in Fig. 7.2.

$M_5 =$	$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/3 & 1/2 \\ 1/2 & 0 & 0 & 1/3 & 0 \\ 0 & 0 & 0 & 0 & 1/2 \\ 1/2 & 0 & 1 & 1/3 & 0 \end{bmatrix}$	<table border="1"> <thead> <tr> <th>Source nodes</th><th>Out degree</th><th>Destination nodes</th></tr> </thead> <tbody> <tr> <td>1</td><td>2</td><td>3, 5</td></tr> <tr> <td>2</td><td>1</td><td>1</td></tr> <tr> <td>3</td><td>1</td><td>5</td></tr> <tr> <td>4</td><td>3</td><td>2, 3, 5</td></tr> <tr> <td>5</td><td>2</td><td>2, 4</td></tr> </tbody> </table>	Source nodes	Out degree	Destination nodes	1	2	3, 5	2	1	1	3	1	5	4	3	2, 3, 5	5	2	2, 4
Source nodes	Out degree	Destination nodes																		
1	2	3, 5																		
2	1	1																		
3	1	5																		
4	3	2, 3, 5																		
5	2	2, 4																		

Figure 7.6 Efficient representation of a transition matrix.

Node 1 has two outward links to nodes 3 and 5. We can deduce that each entry in that column will have value $1/2$. Similarly, node 4 has three outlinks to 2, 3, and 5 and thus column 4 will have value $1/3$ in rows 2, 3, and 5.

7.4.2 PageRank Implementation Using Map Reduce

Recall from Section 7.2.3 that the step used to update PageRank values in one iteration is given by

$$v' = \beta M v + (1 - \beta) e / n$$

Here β normally has a value of around 0.85, n is the number of nodes and e is a vector of 1s. When the number of nodes is small, each Map task can store the vector v fully in main memory and also the new values v' . Thus after this it is only a simple matrix vector multiplication.

The simple algorithm when memory is sufficient for storing all three, v , v' and M is as follows:

Algorithm

- Assume we have enough RAM to fit v' , plus some working memory
 - Store v and matrix M on disk

Basic Algorithm:

- Initialize: $v = [1/N]_N$
- Iterate:
 - Update: Perform a sequential scan of M and v and update v'
 - Write out v' to disk as v for next iteration
 - Every few iterations, compute $|v - v'|$ and stop if it is below threshold

The update function is as follows:

Algorithm

```

Store  $v^{old}(v)$  and matrix  $M$  on disk
Initialize all entries of  $v' = (1 - \beta)/N$ 
For each page  $i$  (of out-degree  $d_i$ ):
  Read into memory:  $i, d_i, dest_1, \dots, dest_{d_i}, v^{old}(i)$ 
  For  $j = 1, \dots, d_i$ 
     $v^{new}(dest_j) += \beta v^{old}(i)/d_i$ 
```

A simple high level pseudo code for MR implementation of PageRank will be as follows:

Algorithm

- **Inputs**
 - Of the form: page → (current_PR_Of_Page, {adjacency list})
- **Map**
 - Page p “propagates” $1/Np$ of its d^* weight(p) ($v(P)$) to the destination pages of its out-edges
- **Reduce**
 - p -th page sums the incoming weights and adds $(1 - d)$ to get its weight'(p) ($v'(p)$)
- **Iterate until convergence**
 - Run some pre-fixed number of times, e.g., 25, 50, etc.
 - Alternatively: Test after each iteration with a second MapReduce job, to determine the maximum change between old and new values and if min. change stop the process.

But given the size of the Web, it is highly likely that vector v may not fit into the main memory. In such cases, the method of striping discussed in the earlier chapter on MapReduce will have to be used. The matrix M is split into vertical stripes v as also into corresponding horizontal stripes. The i^{th} stripe of the matrix multiplies only components from the i^{th} stripe of the vector. Each Map task is now 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. But when we need to compute the PageRank, we have an additional constraint that the result vector should also be partitioned in the same way as the input vector, so the output may become the input for another iteration of the matrix-vector multiplication.

7.4.3 Use of Combiners to Consolidate the Result Vector

Combiners are a general mechanism to reduce the amount of intermediate data. They could be thought of as “mini-reducers”. Adding a combiner in conjunction with the Map task of the PageRank algorithm helps to achieve a more efficient implementation of the PageRank algorithm. One more reason to use an intermediate combiner comes for the fact that for v'_j , the j^{th} component of the result vector v' , at the Map tasks, may generate extra terms due to the teleport factor.

Let us say that the M matrix and v vector are partitioned using striping as discussed in the previous section. Assume that the vector v does not fit in the main memory and so is the case with M . To compute v' we need entire vertical stripe of M and horizontal stripe of v . Since v' is also of the same size as v it will not fit in the memory. Further, any column of M can affect any value in v' . Thus computing v' will result in movement of data to and from main memory leading to thrashing.

One efficient strategy that can be used is to partition the matrix into k^2 blocks, while the vectors are still partitioned into k stripes. Figure 7.7 shows the arrangement for $k = 3$.



Figure 7.7 Matrix partitioned into square blocks.

This method utilizes k^2 Map tasks, one for each square of the matrix M , say M_{ij} , and one stripe of the vector v , v_j . Now we see that that each v_j has to be sent to k different Map tasks; the task handling M_{ij} for each value of i . So, we have to transmit each portion of the v vector to k different locations. Since portion of v is small in size, this operation does not cost too much. Plus using the combiner saves a large amount of data being passed to the Reducer.

The major advantage of this approach is that we can now keep both the j^{th} stripe of v and the i^{th} stripe of v' together in the main memory when we process M_{ij} . We can further optimize the space required by the M matrix by storing only the non-zero entries as discussed in the previous section.

7.5

Topic-Sensitive PageRank

The PageRank algorithm discussed in the earlier section computes a single PageRank vector considering the link structure of the entire Web, to capture the relative “importance” of Web pages. This process is not dependent on any particular search query entered by a random user. Taher H. Haveliwala,

in 2003, proposed a “Personalized” version of PageRank, which he termed Topic-Sensitive PageRank (TSPR). In TSPR instead of computing a single rank vector, a set of rank vectors, one for each (basis) topic, is computed.

In TSPR, we bias the random walker to teleport to a set of topic-specific relevant nodes. The topic is determined by the context of the search query. A set of PageRank vectors, biased using a set of representative topics, helps to capture more accurately the notion of importance with respect to a particular topic. This in turn yields more accurate search results specific to a query. For ordinary keyword search queries, the TSPR scores for pages satisfying the query using the topic of the query keywords are computed.

This algorithm also helps to combat a type of spam called Link Spam which will be discussed in Section 7.5.1.

7.5.1 Motivation for Topic-Sensitive PageRank

Most natural language possess some ambiguity. Take the example of English; different words may mean the same thing and many times same words mean different things in different contexts. For example, a search query using the word “mining” may mean different things depending on whether the user posing the query is a data scientist or a prospector looking to invest in some property where he could set up mines. Similarly, a search engine which is posed a query for “Windows Utilities” can do a better job of returning relevant pages to the user if it is able to infer that the user is a construction specialist in one case and a system programmer in another.

Theoretically, since each user is different, it would be impossible to create a PageRank vector of billions of pages aligning to each user’s interest. The practical way to do this is to compute multiple PageRank scores for each page. Every page is associated with one PageRank value for each topic from a set of possible topics. At query time, these ranks are combined based on the topics of the query to form a composite PageRank score for those pages matching the query.

The basic idea of a TSPR is to determine the topics most closely associated with the query, and use the appropriate topic-sensitive PageRank vectors for ranking the documents satisfying the query.

Algorithm

- For each page compute k different page ranks
 - $K = \text{number of topics}$
 - When computing PageRank w.r.t. to a topic, we bias the random surfer with a probability ϵ to transition to one of the pages of that topic $_k$
- When a query q is issued,
 - Compute similarity between q (+ its context) to each of the topics
 - Take the weighted combination of the topic-specific page ranks of q , weighted by the similarity to different topics

The first important step to computing the TSPR values for a page is to identify a set of “basis” topics which can be used to generate the PageRank vectors. The original TSPR paper suggests the use of 16 top-level categories the *Open Directory Project* as a source of representative basis topics (<http://www.dmoz.org>). These include categories such as Business, Computers, Games, Health, Shopping, Society, Sports, etc. These topics were used to generate 16 PageRank values for every page. If the search engine could determine the area of interest of the user by either knowing the user profile or keeping track of recent pages visited by the user, then the PageRank vector for that topic would be used to rank the pages.

7.5.2 Implementing Topic-Sensitive PageRank

Consider a user who is a doctor and so is interested in a topic “medicine”. To create a TSPR for “medicine” we assume that Web pages on medicine are “near” (linked) to one another in the Web graph. This is not an unreasonable assumption as pages about medicine are more likely to link to other pages related to medicine. Then, a random surfer who frequently finds himself on random medicine pages is likely (in the course of the random walk) to spend most of his time at medicine pages, so that the steady-state distribution of medicines pages is boosted.

The equation for the iteration that yields TSPR is similar to the equation used for simple PageRank. The only difference is now we first identify pages belonging to the topic of interest. We allow a random surfer to teleport to any one of these pages only. This is called the teleport set. For pages that are not in the teleport set, the probability of a random surfer reaching that page is zero. For pages in the teleport set, the probability of a random surfer visiting them next is uniformly divided amongst all members of the teleport set. Let us define S as a set of integers consisting of the row/column numbers for the pages in the teleport set. We redefine the vector e_s to be a vector that has 1 in the components in S and 0 in other components. Then the TSPR for S is the equilibrium value attained by several iterations of

$$v' = \beta M v + (1 - \beta) e_s / |S|$$

As before M is the transition matrix of the Web and $|S|$ is the size of set S which indicates those pages relevant to the current topic. We proceed as described for computing PageRank and only those topic relevant pages get non-zero probability of being visited by a random surfer and so at the stable values these pages will have higher PageRank value as opposed to the other irrelevant pages.

7.5.3 Using Topic-Sensitive PageRank in a Search Engine

Integration of TSPR in a search engine consists of two major phases:

- 1. Pre-processing:** This step is performed offline similar to PageRank. It consists of the following two steps:
 - (a) Fixing the set of topics for which the PageRank vectors are needed. One way is to use the Open Directory. Other online ontologies like Yahoo! and several other manually constructed categories can be used. Google also uses its own query logs to identify important topics.
 - (b) Pick a teleport set for each topic and compute the TSPR vector for that topic.

2. **Query-processing:** This phase is activated online when the user poses a query to the search engine. The following steps are performed:

- (a) “Guess” the relevant set of topics of the query.
- (b) Combine the topic-sensitive scores with respect to the relevant topics to get final ranking score for pages.

Step (a) of query-processing step needs some discussion. Users may either explicitly register their interests, or the search engine may have a method to learn by observing each user’s behavior over time.

A user could also select a relevant topic for a query manually. But in some cases a user is known to have a mixture of interests from multiple topics. For instance, a user may have an interest mixture (or *profile*) that is 75% Business and 40% Computers. How can we compute a *personalized PageRank* for this user?

One way to address this issue is to assume that an individual’s interests can be well-approximated as a linear combination of a small number of topic page distributions. A user with this mixture of interests could teleport as follows: Determine first whether to teleport to the set of known Business pages, or to the set of known Computer pages. This choice is made randomly, choosing Business pages 75% of the time and Computer pages 25% of the time. Once we choose a particular teleport set S ; Assume S to be a Computer page, we then choose a Web page in S with uniform probability to teleport to.

It can be shown that it is not necessary to compute a PageRank vector for every distinct combination of user interests over topics; the personalized PageRank vector for any user can be expressed as a linear combination of the underlying topic-specific PageRanks.

7.6

Link Spam

We have discussed earlier that Spam is the bane of search engines. Spam deteriorates the quality of search results and slowly, over time, reduces the trust of a user in a search engine. This will ultimately result in a tangible cost loss as the user would switch to another Search Engine. Various surveys place the amount of Web Spam as around 20% which means about 20% of the top search results returned by a search engine are suspected cases.

Link-based ranking algorithms were developed to overcome the effects of rampant term spam in content-based ranking algorithms. PageRank was the first endeavor in this direction which was designed to combat term spam. However, spammers are constantly innovating themselves in finding out methods that can boost the PageRank of a page artificially. This form of spam is referred to as link spam.

Link spam can be formally stated as a class of spam techniques that try to increase the link-based score of a target Web page by creating lots of spurious hyperlinks directed towards it. These spurious hyperlinks may originate from a set of Web pages called a Link farm and controlled by the spammer;

these may be created from a set of partner Websites known as link exchange. Sometimes such links could also be placed in some unrelated Websites like blogs or marketplaces.

Search engines can respond to link spam by mining the Web graph for anomalies and propagating a chain of distrust from spurious pages which will effectively lower the PageRank of such pages. In this section, we first present a model for link spam and then discuss techniques to combat them.

7.6.1 Spam Farm

Once Google became the dominant search engine, spammers began to work out ways to fool Google. Spammers concentrated on methods targeting the PageRank algorithm used by Google. The main concept used by PageRank is that for a Web page to be important it must be pointed to by hyperlinks from a large number of other pages and further these pages themselves should be highly ranked. Once spammers became aware of this there have been many attempts to increase the PageRank values of their own pages by either generating a large set of trivial (dummy) pages to point to their target pages, or somehow try to collect links from important pages.

One such method used by spammers is the creation of a “Spam farm” which helps to concentrate PageRank on a single page. For the purpose of setting up a spam farm the spammer has to use the structure of the Web to his/her own advantage. The spammer can divide the Web into three types of pages:

1. **Inaccessible pages** are those that cannot be reached by a spammer. These are pages that cannot be manipulated by a spammer in any way. Thus, these are pages whose PageRank cannot be altered. The only path available to the spammers in these pages is that spammers can have links pointing to these inaccessible pages.
2. **Accessible pages** are those pages owned by entities not directly connected to the spammer. Spammers can still cleverly use these pages to improve PageRank of their spam page. For example, a spammer may be able to post a note on a discussion board and this note could have a link to a spam site. Another example would be to post several tweets on Twitter; these tweets could be dummy tweets with links to the spam site. Of course, there is a limit to the number of links that could be placed in such accessible pages.
3. **Own pages** are pages wholly created and maintained by the spammer. The spammers have full control over the contents of their own pages. These pages could span multiple domains. Spammers try to own as many possible pages and set up complex interconnections between these pages all for the sole purpose of boosting the PageRank values of the spam pages. This group of pages owned and controlled by a spammer is termed “link spam farm”, or more commonly, a “spam farm”.

This simple spam farm model normally has the following characteristics:

1. A spam farm decides one single target page. This is the page for which the spammer wants to boost rankings so that it can be discovered by a search engine.

2. Each spam farm has, within the farm, a fixed number of other pages whose sole purpose is to boost the rank of the target page. This is done by all these pages having links to the target page. This is possible to achieve because the spammer owns all these pages. The size of the spam farm is limited by the resources that the spammer can afford to expend on one target page. This depends on cost of owning the pages, and maintaining it like hardware costs, domain registration expenses, etc.
3. Spammers can also use links from other pages outside the spam farm. These would come from inaccessible pages of the Web. These links are sometimes called “hijacked” links, and the PageRank value added to the spam page through these hijacked links is called “leakage”. The improvement due to leakage is not significant because placing links in inaccessible pages is not very easy as these pages are owned by other entities. Spammers try to choose high PageRank pages like Twitter, etc. to place these hijacked links.

Figure 7.8 presents the organization of a simple spam farm. The figure indicates one target page T in the set of pages forming the spam farm. All own pages have inlinks from the accessible pages, thus they are reachable by a search engine. Since spammers do not control the accessible pages, these inlinks are few in number. Further note that these accessible pages have inlinks to them from the inaccessible set of pages.

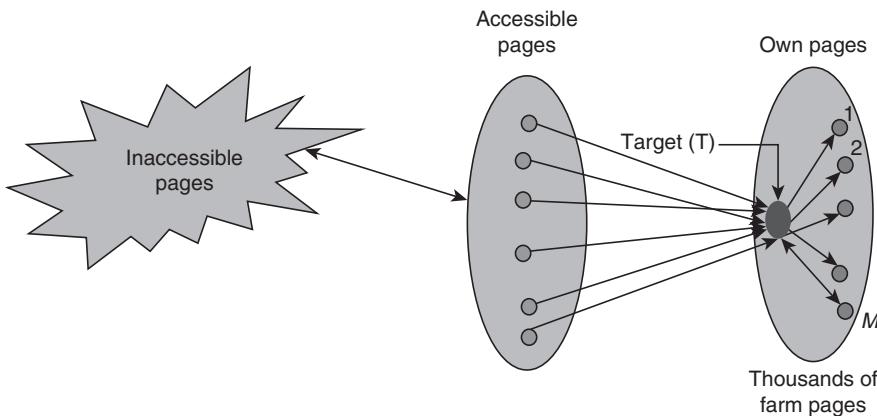


Figure 7.8 Structure of a simple spam farm.

The structure of the spam farm indicates how the PageRank of the target page is maximized:

1. All accessible and own pages have links directly to the target page, thus contributing to increase in PageRank of the target.
2. The target page has links only to other pages in the spam farm. Thus, no PageRank leaks out.
3. Further all the pages in the spam farm point to the target. There are millions of such pages and this cycle of links form target to the other pages and from these other pages back to the target

helps to add significantly to the PageRank. But some PageRank may still leak out due to the availability of the teleport function which may randomly add a link out of the spam farm.

Let us see if we can measure the improvement in PageRank of a single target page based on the above described structure of a spam farm. Let us assume that the teleportation factor β used by the PageRank algorithm is 0.85. The Web is assumed to have n pages. The spam farm consists of m pages supporting a single target t .

We shall denote by x the amount of PageRank contributed by links from the accessible set of pages. Let us denote by y the PageRank of the target page t which we have to evaluate. Let there be m supporting pages in the spam farm. Since t points to the m pages and m pages point to t to get y we need to solve for y in the following equation:

$$y = x + \beta m \left(\frac{\beta y}{m} + \frac{1-\beta}{n} \right) = x + \beta^2 y + \beta(1-\beta) \frac{m}{n}$$

Here $\left(\frac{\beta y}{m} + \frac{1-\beta}{n} \right)$ is the contribution by the m supporting pages linking back to t . We can solve the above equation for y as

$$y = \left(\frac{x}{1-\beta^2} \right) + c \frac{m}{n}$$

where

$$c = \frac{\beta(1-\beta)}{1-\beta^2} = \frac{\beta}{1+\beta}$$

For example when $\beta=0.08$, then

$$\frac{1}{1-\beta^2} = 2.8$$

and

$$c = \frac{1}{1+\beta} = 0.44$$

This indicates that the external contribution to the PageRank has risen by 280% and also 44% of the fraction m/n is from the spam farm. Thus, by making m large we can theoretically boost up the PageRank to any value.

7.6.2 Link Spam Combating Techniques

It is a constant struggle for search engines to identify and eliminate spam from rendering their search results useless. Spam creates mistrust in the user towards the search engine which is unable to differentiate spam sites from real sites.

One way to detect and eliminate link spam would be to identify spam farm like structures and eliminate all those pages from the ranking process. One pattern to search for would be to detect cycles where one page points to numerous other pages which all have links back to only that page. But there are several variants of this pattern that the spammer can resort to, and tracking and detecting all these patterns is infeasible.

Another practical class of methods to track and detect spam would be to make some modifications to the definition and computation of PageRank. The following two techniques are popular:

1. **TrustRank:** This is a variation of TSPR algorithm. The basic idea behind this method is “It is rare for a ‘good’ page to point to a ‘bad’ (spam) page”.

The idea is to start with a seed set of “trustworthy pages”. Either human experts can identify a set of seed pages to start with or a predefined and known set of domains like .edu, .gov. can form the initial seed. We then perform a topic-sensitive PageRank with teleport set = trusted pages formed in the seed. We then propagate a trust value through the links. Each page gets a trust value between 0 and 1. All those pages with trust below a threshold value are marked as spam. The reader is referred to the references for more details of this algorithm.

2. **Spam Mass:** This technique attempts to measure what fraction of a PageRank value could be due to spam. We compute both the normal PageRank (r) and also the TrustRank (t) of a page. TrustRank is computed based on a teleport set of trustworthy pages identified using the methods discussed above. The value $(r-t)/r$ is defined as the “spam mass” of the page. A spam mass value close to 1 indicates a possible spam page. Such pages are then eliminated from the Web search index.

7.7

Hubs and Authorities

In a parallel development along with PageRank, Jon Kleinberg, a professor of Computer Science at Cornell came up with another algorithm to rank pages in relation to a query posed by a user. This algorithm also used the link structure of the Web in order to discover and rank pages relevant for a particular topic. The idea was to associate two scores with each Web page, contributions coming from two different types of pages called “hubs” and “authorities”.

This algorithm called hyperlink-induced topic search (HITS) was first developed to be used with IBM’s effort at a search engine called CLEVER. HITS presently is used by the Asksearch engine (www.Ask.com). Further it is believed that modern information retrieval engines use a combination of PageRank and HITS for query answering.

Similar to the PageRank algorithm, computing the relevance of a page using HITS is an iterative process. But unlike the PageRank algorithm this is not a pre-computed score but a “runtime” algorithm because it is applied only when a user actually submits a query to a search engine. Thus, HITS gives the rank of a page only with relevance to the current query posed by a user.

7.7.1 Hyperlink-Induced Topic Search Concept

The HITS algorithm uses an approach that occurs from an insight into the type of Web pages that can be relevant for a query. Two primary kinds of Web pages are identified as significant for *broad-topic searches*. There are authoritative sources of information on the topic, called “**authorities**”. On the other hand, a set of Web pages just contain a lists of link to authoritative Web pages on a specific topic called “**hubs**”. The idea is that both these types of pages are important when answering a query and thus the relevance of a page should depend on the combination of a “hub score” and an “authority score”.

Authority pages contain valuable information on the subject of a query whereas a hub page contains links to several useful authoritative pages relevant to the query. Figure 7.9 illustrates hubs and authority pages.

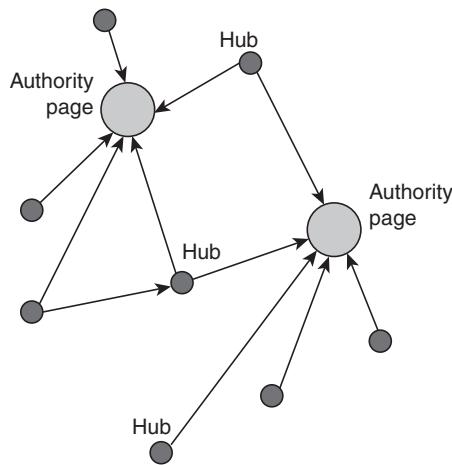


Figure 7.9 Hubs and authorities.

Example 4

Assume a user posts a query “**Big Data Analytics**” several pages which give an overview of big data algorithms, pages dealing with courses dealing with big data all form authoritative pages. But a page like “[kddnuggets.com](#)” or “[AnalyticsVidya.com](#)” which contain links to all relevant resources on Big Data will form hub pages. These hub pages are also important pages since it is possible to reach thousands of authority pages through them.

Example 5

Assume the search query is “Python Programming”. Pages that contain information about python tutorial sites like “[www.python.org](#)”, “[www.pythontutor.com](#)”, etc. are authority pages. Pages like “[pythonprogramming.net](#)” contain links to several notes, tutorial, etc. related to Python and these form hub pages.

7.7.2 Hyperlink-Induced Topic Search Algorithm

HITS identifies good authorities and hubs for a query topic by assigning two scores to a page: an authority score and a hub score. These weights are defined recursively.

1. A page is a good authoritative page with respect to a given query if it is referenced (i.e., pointed to) by many (good hub) pages that are related to the query.
2. A page is a good hub page with respect to a given query if it points to many good authoritative pages with respect to the query.
3. Good authoritative pages (authorities) and good hub pages (hubs) reinforce each other.

Thus, the task of finding the most relevant pages with respect to a query is the problem of finding dense subgraphs of **good** hubs and **good** authorities. Suppose that we do find such a subset of the Web containing good hub and authority pages, together with the hyperlinks. We can then iteratively compute a hub score and an authority score for every Web page in this subset.

Now to start the algorithm we require an initial set of good pages. Typically, a set of pages most relevant (about 200) to the user's initial query is retrieved from the WWW. This can be done by using simple text-based information retrieval techniques. This initial set is called the "root" set. The root set is then extended one step by including any pages that link to a page in the root set. To make the algorithm feasible, the initial root set finds about 200 relevant pages and the extension operation about 70–100 pages more. This is known as the "base" or the "seed" set and this is used as an input to the HITS algorithm. This set forms the Web graph for the HITS algorithm.

7.7.2.1 Steps of the HITS Algorithm (When Submitted with a Query q)

Let us assume a query q is submitted to a search engine using the HITS algorithm. The following steps are performed by the algorithm to return to the user a set of relevant pages for the query q :

1. Submit q to a regular similarity-based search engine. Let S be the set of top n pages returned by the search engine. (S is the root set and n is about 200.)
2. Expand S into a large set T (base set):
 - Add pages that are pointed to by any page in S .
 - Add pages that point to any page in S . (Add around 70–100 pages.)
3. Find the sub-graph SG of the Web graph that is induced by T . (Dense graph of good authorities and hubs.) Figure 7.10 illustrates this.
4. Compute the authority score and hub score of each Web page in T using sub-graph $SG(V, E)$. Given a page p , let
 - (a) $a(p)$ be the authority score of p
 - (b) $h(p)$ be the hub score of p
 - (c) (p, q) be a directed edge in E from p to q

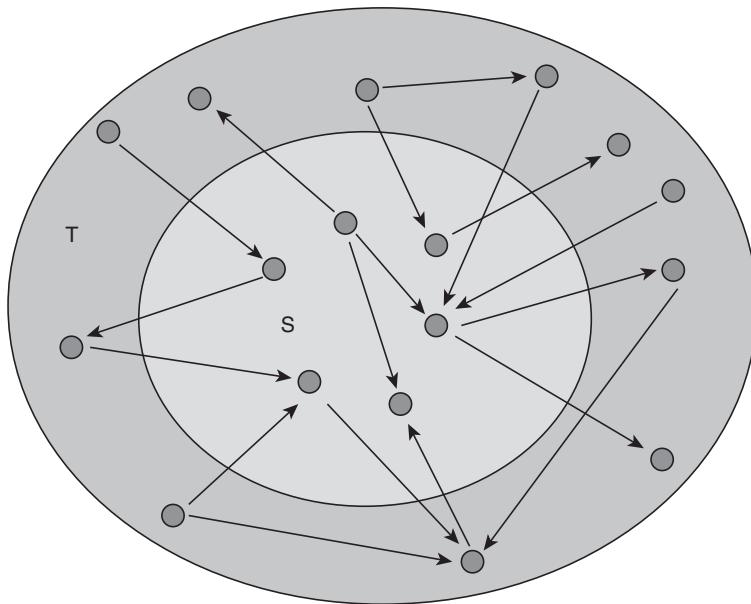


Figure 7.10 Identifying dense sub-graph.

5. Apply two basic operations for updating the scores:

- **Operation I:** Update each $a(p)$ as the sum of all the hub scores of Web pages that point to p .
- **Operation O:** Update each $h(p)$ as the sum of all the authority scores of Web pages pointed to by p .

Figure 7.11 depicts operations I and O.

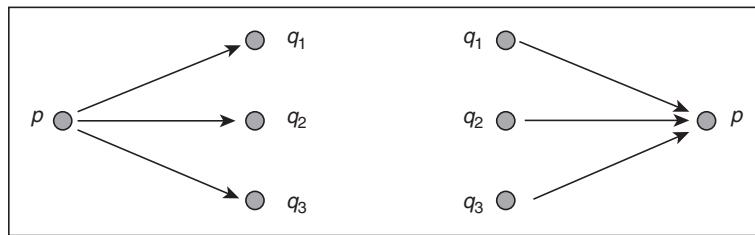


Figure 7.11 Graphs indicating operations I and O.

6. To apply the operations to all pages of the Web graph at once we use matrix representation of operations I and O.
- Let A be the adjacency matrix of SG: $A(p, q)$ is 1 if p has a link to q , else the entry is 0.
 - Let A^T be the transpose of A .

- Let h_i be vector of hub scores after i iterations.
- Let a_i be the vector of authority scores after i iterations.
- Operation I: $a_i = A^T h_{i-1}$
- Operation O: $h_i = A a_i$
- The iterative version of these operations are as shown below:

$$a_i = A^T A a_{i-1}, h_i = A A^T h_{i-1}$$

$$a_i = (A^T A)^i a_0, h_i = (A A^T) h_0$$

7. Since at each iteration, we are adding the hub and authority scores to compute next set of hub and authority scores, values may keep increasing without bounds. Thus, after each iteration of applying Operations I and O, normalize all authority and hub scores so that maximum score will be 1.

$$a(p) = \frac{a(p)}{\sqrt{\sum_{q \in V} [a(q)]^2}}$$

$$h(p) = \frac{h(p)}{\sqrt{\sum_{q \in V} [h(q)]^2}}$$

8. Repeat until the scores for each page converge.
9. Sort pages in descending authority scores.
10. Display the top authority pages.

The following text box summarizes this procedure:

Algorithm

1. Submit q to a search engine to obtain the root set S .
2. Expand S into the base set T .
3. Obtain the induced sub-graph $SG(V, E)$ using T .
4. Initialize $a(p) = h(p) = 1$ for all p in V .
5. For each p in V until the scores converge
 - { apply Operation I;
 - apply Operation O;
 - normalize $a(p)$ and $h(p)$; }
6. Return pages with top authority scores.

Example 6

Consider the Web graph as shown in Fig. 7.12. Initialize all scores to 1.

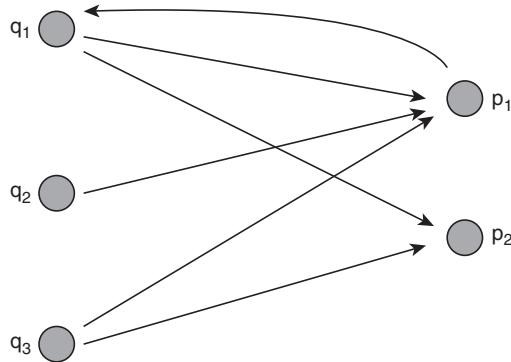


Figure 7.12 Sub-graph.

1st Iteration:**I operation:**

- $a(q_1) = 1, a(q_2) = a(q_3) = 0, a(p_1) = 3, a(p_2) = 2$

O operation:

- $b(q_1) = 5, b(q_2) = 3, b(q_3) = 5, b(p_1) = 1, b(p_2) = 0$

Normalization:

- $a(q_1) = 0.267, a(q_2) = a(q_3) = 0, a(p_1) = 0.802, a(p_2) = 0.535$
- $b(q_1) = 0.645, b(q_2) = 0.387, b(q_3) = 0.645, b(p_1) = 0.129, b(p_2) = 0$

After Two Iterations:

- $a(q_1) = 0.061, a(q_2) = a(q_3) = 0, a(p_1) = 0.791, a(p_2) = 0.609$
- $b(q_1) = 0.656, b(q_2) = 0.371, b(q_3) = 0.656, b(p_1) = 0.029, b(p_2) = 0$

After Five Iterations:

- $a(q_1) = a(q_2) = a(q_3) = 0, a(p_1) = 0.788, a(p_2) = 0.615$
- $b(q_1) = 0.657, b(q_2) = 0.369, b(q_3) = 0.657, b(p_1) = b(p_2) = 0$

Eventually these numbers will converge.

Generally, after a number of iterations, the authority and hub scores do not vary much and can be considered to have “converged”.

HITS algorithm and the **PageRank algorithm** both make use of the link structure of the Web graph to decide the relevance of the pages. The difference is that while the **PageRank** is query independent and works on a large portion of the Web, **HITS** only operates on a small subgraph (the seed S_Q) from the Web graph.

The most obvious strength of HITS is the two separate vectors it returns, which allow the application to decide on which score it is most interested in. The highest ranking pages are then displayed to the user by the query engine.

This sub-graph generated as seed is query dependent; whenever we search with a different query phrase, the seed changes as well. Thus, the major disadvantage of HITS is that the query graph must be regenerated dynamically for each query.

Using a query-based system can also sometimes lead to link spam. Spammers who want their Web page to appear higher in a search query, can make spam farm pages that link to the original site to give it an artificially high authority score.

Summary

- As search engines become more and more sophisticated, to avoid being victims of spam, spammers also are finding innovative ways of defeating the purpose of these search engines. One such technique used by modern search engines to avoid spam is to analyze the hyperlinks and the graph structure of the Web for ranking of Web search results. This is called Link Analysis.
- Early search engines were mostly text based and susceptible to spam attacks. **Spam** means “manipulation of Web page content for the purpose of appearing high up in search results for selected keywords”.
- To attack text-based search engines, spammers resorted to term based spam attacks like cloaking and use of Doorway pages.
- Google, the pioneer in the field of search engines, came up with two innovations based on Link Analysis to combat term spam and called their algorithm PageRank.
- The basic idea behind PageRank is that the ranking of a Web page is not dependent only on terms appearing on that page, but some weightage is also given to the terms used in or near the links to that page. Further pages with large no of visits are more important than those with few visits.
- To compute PageRank, “Random Surfer Model” was used. Calculation of PageRank can be thought of as simulating the behavior of many random surfers, who each start at a random page and at any step move, at random, to one of the pages to which their

current page links. The limiting probability of a surfer being at a given page is the PageRank of that page.

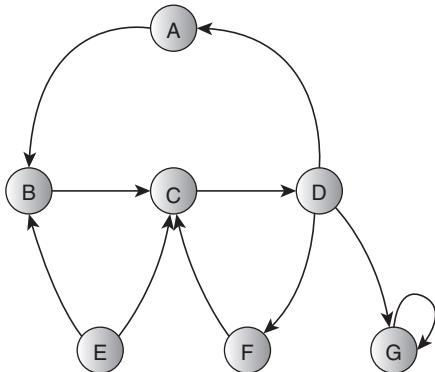
- An iterative matrix based algorithm was proposed to compute the PageRank of a page efficiently.
- The PageRank algorithm could be compromised due to the bow-tie structure of the Web which leads to two types of problems, dead ends and spider traps.
- Using a scheme of random teleportation, PageRank can be modified to take care of dead ends and spider traps.
- To compute the PageRank of pages on the Web efficiently, use of MapReduce is advocated. Further schemes of efficiently storing the transition matrix and the PageRank vector are described.
- In Topic-Sensitive PageRank, we bias the random walker to teleport to a set of topic-specific relevant nodes. The topic is determined by the context of the search query. A set of PageRank vectors, biased using a set of representative topics, helps to capture more accurately the notion of importance with respect to a particular topic. This in turn yields more accurate search results specific to a query.
- Link spam can be formally stated as a class of spam techniques that try to increase the link-based score of a target Web page by creating lots of spurious hyperlinks directed towards it. These spurious hyperlinks may originate from a set of Web pages called a Link farm

and controlled by the spammer. They may be created from a set of partner Web sites known as link exchange. Sometimes such links could also be placed in some unrelated Websites like blogs or marketplaces. These structures are called Spam farms.

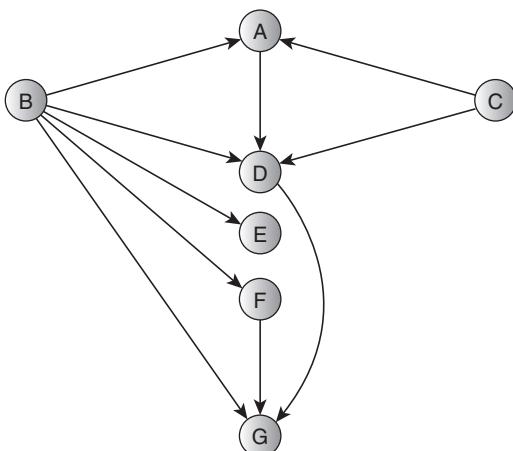
- Search engines can respond to link spam by mining the Web graph for anomalies and propagating a chain of distrust from spurious pages which will effectively lower the PageRank of such pages. TrustRank and Spam mass are two techniques used to combat Link Spam.
- In a parallel development along with PageRank, another algorithm to rank pages in relation to a query posed by a user was proposed. This algorithm also used the link structure of the Web in order to discover and rank pages relevant for a particular topic. The idea was to associate two scores with each Web page, contributions coming from two different types of pages called “hubs” and “authorities”. This algorithm is called hyperlink-induced topic search (HITS). HITS presently is used by the **Ask** search engine (www.Ask.com). Further it is believed that modern information retrieval engines use a combination of PageRank and HITS for query answering.
- Calculation of the hubs and authorities scores for pages depends on solving the recursive equations: “a hub links to many authorities, and an authority is linked to by many hubs”. The solution to these equations is essentially an iterated matrix–vector multiplication, just like PageRank’s.

Exercises

1. Consider the portion of a Web graph shown below.



- (a) Compute the hub and authority scores for all nodes.
 - (b) Does this graph contain spider traps? Dead ends? If so, which nodes?
 - (c) Compute the PageRank of the nodes without teleportation and with teleportation $\beta = 0.8$.
2. Compute hub and authority scores for the following Web-graph:



3. Let the adjacency matrix for a graph of four vertices (n_1 to n_4) be as follows:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Calculate the authority and hub scores for this graph using the HITS algorithm with $k = 6$, and identify the best authority and hub nodes.

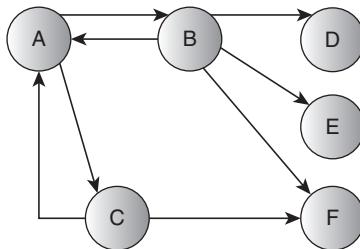
4. Can a Website's PageRank ever increase? What are its chances of decreasing?
5. Given the following HITS scoring vector, normalize x for the next iteration of the algorithm:

$$x = \begin{bmatrix} 3.12 \\ 4.38 \\ 6.93 \\ 3.41 \\ 1.88 \\ 4.53 \end{bmatrix}$$

6. Consider the Web graph given below with six pages (A, B, C, D, E, F) with directed links as follows:

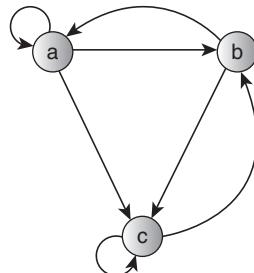
- $A \rightarrow B, C$
- $B \rightarrow A, D, E, F$
- $C \rightarrow A, F$

Assume that the PageRank values for any page m at iteration 0 is $PR(m) = 1$ and teleportation factor for iterations is $\beta = 0.85$. Perform the PageRank algorithm and determine the rank for every page at iteration 2.



7. Consider a Web graph with three nodes 1, 2, and 3. The links are as follows: $1 \rightarrow 2$, $3 \rightarrow 2$, $2 \rightarrow 1$, $2 \rightarrow 3$. Write down the transition probability matrices for the surfer's walk with teleporting, for the following three values of the teleport probability:
- $\beta = 0$
 - $\beta = 0.5$
 - $\beta = 1$

8. Compute the PageRank of each page in the following graph without taxation and with taxation factor of 0.8.



9. For all the Web graphs discussed in the exercises, represent the transition matrices and the PageRank vector using methods discussed in Section 7.3.
10. Compute the TopicSensitive PageRank for the graph of Exercise 2 assuming the teleport set is: (A, C, E, F) only.

Programming Assignments

- Implement the following algorithms on standard datasets available on the web. The input will normally be in the form of sparse matrices representing the webgraph.
 - Simple PageRank Algorithm
 - PageRank algorithm with a teleportation factor to avoid dead-ends and spider traps.
- Describe how you stored the connectivity matrix on disk and how you computed the transition matrix. List the top-10 pages as returned by the algorithms in each case.
- Now rewrite portions of the code to implement the TrustRank algorithm. The user will specify which pages (indices) correspond to trustworthy pages. It might be good to look at the URLs and identify reasonable candidates for trustworthy pages.
- Implement Assignments 1 and 3 using MapReduce.
- Implement HITS algorithm any webgraph using MapReduce.

References

- C.D. Manning, P. Raghavan, H. Schütze (2008). *Introduction to Information Retrieval*.

Website: <http://informationretrieval.org/>; Cambridge University Press.

2. D. Easley, J. Kleinberg (2010). *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Cambridge University Press. Complete preprint on-line at <http://www.cs.cornell.edu/home/kleinber/networks-book/>.
3. Page, Lawrence and Brin, Sergey and Motswani, Rajeev and Winograd, Terry (1999) *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report. Stanford InfoLab.
4. T. Haveliwala. Efficient Computation of PageRank. Tech. rep., Stanford University, 1999.
5. T. Haveliwala (2002). Topic-sensitive Page-Rank. In *Proceedings of the Eleventh International Conference on World Wide Web*, 2002.
6. J. Kleinberg (1998). Authoritative Sources in a Hyperlinked Environment. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*.
7. A. Broder, R. Kumar, F. Maghoul et al. (2000). Graph structure in the Web, *Computer Networks*, 33:1–6, pp. 309–320.

8

Frequent Itemset Mining

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Review your knowledge about frequent itemsets and basic algorithms to identify them.
- Learn about different memory efficient techniques to execute the traditional FIM algorithms.
- Understand how these algorithms are insufficient to handle larger datasets.
- Learn about the algorithm of Park, Chen and Yu, and its variants.
- Understand the sampling-based SON Algorithm and how it can be parallelized using MapReduce.
- Learn some simple stream-based frequent itemset mining methods.

8.1

Introduction

Frequent itemsets play an essential role in many data mining tasks where one tries to find interesting patterns from databases, such as association rules, correlations, sequences, episodes, classifiers, clusters and many more. One of the most popular applications of frequent itemset mining is discovery of association rules. The identification of sets of items, products, symptoms, characteristics and so forth that often occur together in the given database can be seen as one of the most basic tasks in data mining. This chapter discusses a host of algorithms that can be effectively used to mine frequent itemsets from very massive datasets.

This chapter begins with a conceptual description of the “market-basket” model of data. The problem of deriving associations from data was first introduced using the “market-basket” model of data, which is essentially a many-many relationship between two kinds of elements, called “items” and “baskets”. The frequent-itemsets problem is that of finding sets of items that appear in (are related to) many of the same baskets.

The problem of finding frequent itemsets differs from the similarity search discussed in Chapter 5. In the frequent itemset scenario, we attempt to discover sets of items that are found in the same baskets frequently. Further we need the number of such buckets where these items appear together to

be sufficiently large so as to be statistically significant. In similarity search we searched for items that have a large fraction of their baskets in common, even if the absolute number of such baskets is small in number.

Many techniques have been invented to mine databases for frequent events. These techniques work well in practice on smaller datasets, but are not suitable for truly big data. Applying frequent itemset mining to large databases is a challenge. First of all, very large databases do not fit into main memory. For example consider the well-known *Apriori* algorithm, where frequency counting is achieved by reading the dataset over and over again for each size of candidate itemsets. Unfortunately, the memory requirements for handling the complete set of candidate itemsets blows up fast and renders Apriori-based schemes very inefficient to use on large data.

This chapter proposes several changes to the basic *Apriori* algorithm to render it useful for large datasets. These algorithms take into account the size of the main memory available.

Since exact solutions are costly and impractical to find in large data, a class of approximate algorithms is discussed which exploit parallelism, especially the Map–Reduce concept. This chapter also gives a brief overview of finding frequent itemsets in a data stream.

8.2

Market-Basket Model

“Market-Baskets” is an abstraction that models any many-many relationship between two concepts: “items” and “baskets”. Since the term a “market-basket” has its origin in retail applications, it is sometimes called “transactions”. Each basket consists of a set of items (an itemset), and usually we assume that the number of items in a basket is small – much smaller than the total number of items. The number of baskets is usually assumed to be very large, bigger than what can fit in main memory.

Items need not be “contained” in baskets. We are interested in the co-occurrences of items related to a basket, not vice-versa. For this purpose, we define basket data in general terms. Let $I = \{i_1, \dots, i_k\}$ be a set of k elements, called items. Let $B = \{b_1, \dots, b_n\}$ be a set of n subsets of I . We call each $b_i \subset I$ a basket of items. For example, in a retail market-basket application, the set I consists of the items stocked by a retail outlet and each basket is the set of purchases from one register transaction; on the other hand, in a document basket application, the set I contains all dictionary words and proper nouns, while each basket is a single document in the corpus and each basket consists of all words that occur in that document.

8.2.1 Frequent-Itemset Mining

Let $I = \{i_1, \dots, i_k\}$ be a set of items. Let D , the task-relevant data, be a set of database transactions where each transaction T is a set of items such that $T \subseteq I$. Each transaction is associated with an

identifier, called TID. Let A be a set of items. A transaction T is said to contain A if and only if $A \subseteq T$.

A set of items is referred to as an itemset. An itemset that contains k items is a k -itemset. For example, consider a computer store with computer-related items in its inventory. The set {computer, anti-virus software, printer, flash-drive} is a 4-itemset. The occurrence frequency of an itemset is the number of transactions that contain the itemset. This is also known, simply, as the frequency, support count, or count of the itemset. We can call an itemset I a “frequent itemset” only if its support count is sufficiently large. We prescribe a *minimum support* s and any I which has support greater than or equal to s is a frequent itemset.

Example 1

Items = {milk (m), coke (c), pepsi (p), beer (b), juice (j)}

Minimum support $s = 3$

Transactions

1. $T_1 = \{m, c, b\}$
2. $T_2 = \{m, p, j\}$
3. $T_3 = \{m, b\}$
4. $T_4 = \{c, j\}$
5. $T_5 = \{m, p, b\}$
6. $T_6 = \{m, c, b, j\}$
7. $T_7 = \{c, b, j\}$
8. $T_8 = \{b, c\}$

Frequent itemsets: $\{m\}$, $\{c\}$, $\{b\}$, $\{j\}$, $\{m, b\}$, $\{c, b\}$, $\{j, c\}$.

8.2.2 Applications

A supermarket chain may have 10,000 different items in its inventory. Daily millions of customers will push their shopping carts (“market-baskets”) to the checkout section where the cash register records the set of items they purchased and give out a bill. Each bill thus represents one market-basket or one transaction. In this scenario, the identity of the customer is not strictly necessary to get useful information from the data. Retail organizations analyze the market-basket data to learn what typical customers buy together.

Example 2

Consider a retail organization that spans several floors, where soaps are in floor 1 and items like towels and other similar goods are in floor 10. Analysis of the market-basket shows a large number of baskets containing both soaps and towels. This information can be used by the supermarket manager in several ways:

1. Apparently, many people walk from where the soaps are to where the towels are which means they have to move from floor 1, catch the elevator to move to floor 10. The manager could choose to put a small shelf in floor 1 consisting of an assortment of towels and some other bathing accessories that might also be bought along with soaps and towels, for example, shampoos, bath mats etc. Doing so can generate additional “on the spot” sales.
2. The store can run a sale on soaps and at the same time raise the price of towels (without advertising that fact, of course). People will come to the store for the cheap soaps, and many will need towels too. It is not worth the trouble to go to another store for cheaper towels, so they buy that too. The store makes back on towels what it loses on soaps, and gets more customers into the store.

While the relationship between soaps and towels seems somewhat obvious, market-basket analysis may identify several pairs of items that occur together frequently but the connections between them may be less obvious. For example, the analysis could show chocolates being bought with movie CDs. But we need some rules to decide when a fact about co-occurrence of sets of items can be useful. Firstly any useful set (need not be only pairs) of items must be bought by a large number of customers. It is not even necessary that there be any connection between purchases of the items, as long as we know that lots of customers buy them.

Example 3

Online Commerce

An E-Retail store like E-bay or Amazon.com offers several million different items for sale through its websites and also cater to millions of customers. While normal offline stores, such as the supermarket discussed above, can only make productive decisions when combinations of items are purchased by very large numbers of customers, online sellers have the means to tailor their offers even to a single customer. Thus, an interesting question is to find pairs of items that many customers have bought together. Then, if one customer has bought one of these items but not the other, it might be good for Amazon or E-bay to advertise the second item when this customer next logs in. We can treat the purchase data as a market-basket problem, where each “basket” is the set of items that one particular customer has ever bought. But there is another way online sellers can use the same data. This approach, called “collaborative filtering”, finds sets of customers with similar purchase behavior. For example, these businesses look for pairs, or even larger sets, of customers who

have bought many of the same items. Then, if a customer logs in, these sites might pitch an item that a similar customer bought, but this customer has not. Finding similar customers also form one of the market-basket problems. Here, however, the “items” are the customers and the “baskets” are the items for sale. That is, for each item sold online, there is a “basket” consisting of all the customers who bought that item.

Another difference in the two types of businesses, online and offline, is in terms of scale. The meaning of “large number of baskets” differs in both the cases as the following shows:

1. In the brick-and-mortar offline case, we may need thousands of baskets containing the same set of items before we can exploit that information profitably. For online stores, we need many fewer baskets containing a same set of items before we can use the information in the limited context as described above.
2. The brick-and-mortar store does not need too many examples of good sets of items to use; they cannot run sales on millions of items. In contrast, the on-line store needs millions of good pairs to work with – at least one for each customer.

Some other examples of important applications of frequent itemset mining include the following:

1. **Customer transaction analysis:** In this case, the transactions represent sets of items that co-occur in customer buying behavior. In this case, it is desirable to determine frequent patterns of buying behavior, because they can be used for making decision about shelf stocking or recommendations.
2. **Other data mining problems:** Frequent pattern mining can be used to enable other major data mining problems, such as classification, clustering and outlier analysis. This is because the use of frequent patterns is so fundamental in the analytical process for a host of data mining problems.
3. **Web mining:** In this case, the web logs may be processed in order to determine important patterns in the browsing behavior. This information can be used for Website design, recommendations, or even outlier analysis.
4. **Software bug analysis:** Executions of software programs can be represented as graphs with typical patterns. Logical errors in these bugs often show up as specific kinds of patterns that can be mined for further analysis.
5. **Chemical and biological analysis:** Chemical and biological data are often represented as graphs and sequences. A number of methods have been proposed in the literature for using the frequent patterns in such graphs for a wide variety of applications in different scenarios.

8.2.3 Association Rule Mining

The main purpose to discovering frequent itemsets from a large dataset is to discover a set of if-then rules called Association rules. The form of an association rule is $I \rightarrow j$, where I is a set of items (products)

and j is a particular item. The implication of this association rule is that if all of the items in I appear in some basket, then j is “likely” to appear in that basket as well.

A more formal definition of an association rule is as follows:

Let $I = \{I_1, I_2, \dots, I_m\}$ be a set of m distinct attributes, also called *literals*. Let D be a database, where each record (tuple, transaction, basket) T has a unique identifier, and contains a set of items from the set I . An *association rule* is an implication of the form $X \Rightarrow Y$, where $X, Y \subseteq I$ are itemsets and $X \cap Y = \emptyset$. Here, X is called antecedent and Y consequent.

Association rules are of interest only when they result from a frequent itemset. This means that we are only interested in association rules such as $X \Rightarrow Y$, where $X \cup Y$ is a frequent itemset. Thus frequency of occurrence in $X \cup Y$ is at least equal to the minimum support s .

To test the reliability of the inference made in the association rule, we define one more measure called the “Confidence” of the rule. We are interested only in such rules that satisfy a minimum confidence c .

Let $X \rightarrow Y$ be an Association rule. The confidence of the rule is defined as the fraction of the itemsets that support the rule among those that support the antecedent:

$$\text{Confidence } (X \rightarrow Y) := P(Y | X) = \text{support}(X \cup Y) / \text{support}(X)$$

The confidence of a rule indicates the degree of correlation in the dataset between X and Y . Confidence is a measure of a rule’s strength. Minimum support and minimum confidence are two measures of rule interestingness. They respectively reflect the usefulness and certainty of the discovered rules. Typically, association rules are considered interesting if they satisfy both a minimum support threshold and a minimum confidence threshold. Users or domain experts can set such thresholds. Additional analysis can be performed to uncover interesting statistical correlations between associated items.

Example 4

Consider the following baskets:

1. $B_1 = \{m, c, b\}$
2. $B_2 = \{m, p, j\}$
3. $B_3 = \{m, b\}$
4. $B_4 = \{c, j\}$
5. $B_5 = \{m, b, p\}$
6. $B_6 = \{m, c, b, j\}$
7. $B_7 = \{c, b, j\}$
8. $B_8 = \{m, b, c\}$

An association rule $\{m, b\} \rightarrow c$ has Support = Frequency $\{m, b, c\} = \frac{3}{8} = 37.5\%$

$$\text{Confidence} = \frac{\text{Support}\{m, b, c\}}{\text{Support}\{m, b\}} = \frac{3}{5} = 60\%$$

The equation for confidence indicates that the confidence of rule $A \Rightarrow B$ can be easily derived from the support counts of A and $A \cup B$. That is, once the support counts of A , B and $A \cup B$ are found, it is straightforward to derive the corresponding association rules $A \Rightarrow B$ and $B \Rightarrow A$ and check whether they are strong.

Thus, the problem of mining association rules can be reduced to that of mining frequent itemsets. In general, association rule mining can be viewed as a two-step process:

1. Find all frequent itemsets: By definition, each of these itemsets will occur at least as frequently as a predetermined minimum support count, ***min sup***.
2. Generate strong association rules from the frequent itemsets: By definition, these rules must satisfy minimum support and minimum confidence.

Example 5

Consider a small database with four items $I = \{\text{Bread, Butter, Eggs, Milk}\}$ and four transactions as shown in Table 8.1. Table 8.2 shows all itemsets for I . Suppose that the minimum support and minimum confidence of an association rule are 40% and 60%, respectively. There are several potential association rules. For discussion purposes, we only look at those in Table 8.3. At first, we have to find out whether all sets of items in those rules are large. Secondly, we have to verify whether a rule has a confidence of at least 60%. If the above conditions are satisfied for a rule, we can say that there is enough evidence to conclude that the rule holds with a confidence of 60%. Itemsets associated with the aforementioned rules are: $\{\text{Bread, Butter}\}$ and $\{\text{Butter, Eggs}\}$. The support of each individual itemset is at least 40% (see Table 8.2). Therefore, all of these itemsets are large. The confidence of each rule is presented in Table 8.3. It is evident that the first rule ($\text{Bread} \Rightarrow \text{Butter}$) holds. However, the second rule ($\text{Butter} \Rightarrow \text{Eggs}$) does not hold because its confidence is less than 60%.

Table 8.1 Transaction database

<i>Transaction ID</i>	<i>Items</i>
i	Bread, Butter, Eggs
T2	Butter, Eggs, Milk

(Continued)

Table 8.1 (Continued)

<i>Transaction ID</i>	<i>Items</i>
T3	Butter
T4	Bread, Butter

Table 8.2 Support for itemsets in Table 8.1 and large itemsets with a support of 40%

<i>Itemset</i>	<i>Support, s</i>	<i>Large/Small</i>
Bread	50%	Large
Butter	100%	Large
Eggs	50%	Large
Milk	25%	Small
Bread, Butter	50%	Large
Bread, Eggs	25%	Small
Bread, Milk	0%	Small
Butter, Eggs	50%	Large
Butter, Milk	25%	Small
Eggs, Milk	25%	Small
Bread, Butter, Eggs	25%	Small
Bread, Butter, Milk	0%	Small
Bread, Eggs, Milk	0%	Small
Butter, Eggs, Milk	25%	Small
Bread, Butter, Eggs, Milk	0%	Small

Table 8.3 Confidence of some association rules where $\alpha = 60\%$

<i>Rule</i>	<i>Confidence</i>	<i>Rule-hold</i>
Bread \Rightarrow Butter	100%	Yes
Butter \Rightarrow Bread	50%	No
Butter \Rightarrow Eggs	50%	No
Eggs \Rightarrow Butter	100%	Yes

It will normally be the case that there are not too many frequent itemsets and thus not too many candidates for high-support, high-confidence association rules. The reason is that each one found must be acted upon to improve some performance of the system. It could be to improve cross-selling in a retail scenario or improve the quality of results returned by a search algorithm by finding associated keywords from a search query. Thus, it is logical to adjust the support threshold so that we do not get too many frequent itemsets. This assumption helps to design efficient algorithms for finding frequent itemsets.

Most association rule mining algorithms employ a support-confidence framework to prune out trivial and non-interesting rules. However, several times even strong association rules can be un-interesting and misleading. The support-confidence framework then needs to be supplemented with additional interestingness measures based on statistical significance and correlation analysis.

Example 6

Consider the following table that gives the counts of people using saving's account and credit cards from a bank dataset.

<i>Credit Card</i>			
Saving's account	No	Yes	Total
No	50	350	400
Yes	100	500	600

Consider the association rule: $S \geq C$ (People with savings account are likely to have a credit card). Then

$$\text{Support } (S \geq C) = \frac{500}{1000} = 50\%$$

$$\text{Confidence } (S \geq C) = \frac{500}{600} = 83\%$$

$$\text{Expected confidence } (S \geq C) = \frac{350 + 500}{1000} = 85\%$$

Based on the support and confidence, it might be considered a strong rule. However, people without a savings account are even more likely to have a credit card ($=350/400 = 87.5\%$). Savings account and Credit card are, in fact, found to have a negative correlation. Thus, high confidence and support does not imply cause and effect; the two products at times might not even be correlated.

To avoid cases where the association rule actually works to a disadvantage, another interestingness measure is often introduced, called “Lift.” The “Lift” of a rule $X \geq Y$ is the confidence of the rule divided by the expected confidence, assuming that the itemsets are independent.

A lift value greater than 1 indicates that X and Y appear more often together than expected; this means that the occurrence of X has a positive effect on the occurrence of Y or that X is positively correlated with Y .

$$\begin{aligned}\text{Lift}(X \rightarrow Y) &= \frac{\text{Confidence}(X \rightarrow Y)}{\text{Support}(Y)} \\ &= \frac{\text{Support}(X + Y)}{\text{Support}(X) * \text{Support}(Y)}\end{aligned}$$

A lift smaller than 1 indicates that X and Y appear less often together than expected. This means that the occurrence of X has a negative effect on the occurrence of Y or that X is negatively correlated with Y .

A lift value near 1 indicates that X and Y appear almost as often together as expected; this means that the occurrence of X has almost no effect on the occurrence of Y or that X and Y have zero correlation.

In Example 6, savings account and credit card are, in fact, found to have a negative correlation. Thus, high confidence and support do not imply cause and effect; the two products, at times, might not even be correlated. One has to exercise caution in making any recommendations in such cases and look closely at the lift values.

8.3

Algorithm for Finding Frequent Itemsets

In this section we will introduce the seminal algorithm for finding frequent itemsets, the *Apriori Algorithm*. Since the original Apriori algorithm is very effort-intensive both with space and time, several improvements to deal with Big Data are introduced. This section begins with possible representations for the dataset that will optimize the finding of frequent itemsets. Further techniques for implementing the algorithm using mainly the main memory are also discussed. These techniques are of significance when the algorithm is used for big data.

8.3.1 Framework for Frequent-Itemset Mining

Before we proceed with discussing algorithms for finding frequent itemsets, we have to identify how the market-baskets are organized in memory. Generally market-basket data is stored in a file basket-by-basket. It is stored as a set of records in a flat file in the form:

$$(\text{Bid} : item_1, item_2, \dots, item_n)$$

Sometimes, it could also be stored in a relational database as a relation $Baskets(BID, item)$. Since it is reasonable to assume that the data is too large to fit in the main memory, the data could possibly be stored in a distributed file system. Alternately, the data may be stored in a conventional file, with a character code to represent the baskets and their items.

Figure 8.1 depicts one such layout of a market-basket data in a conventional file. The dark thick line indicates basket boundaries. Items are positive integers.

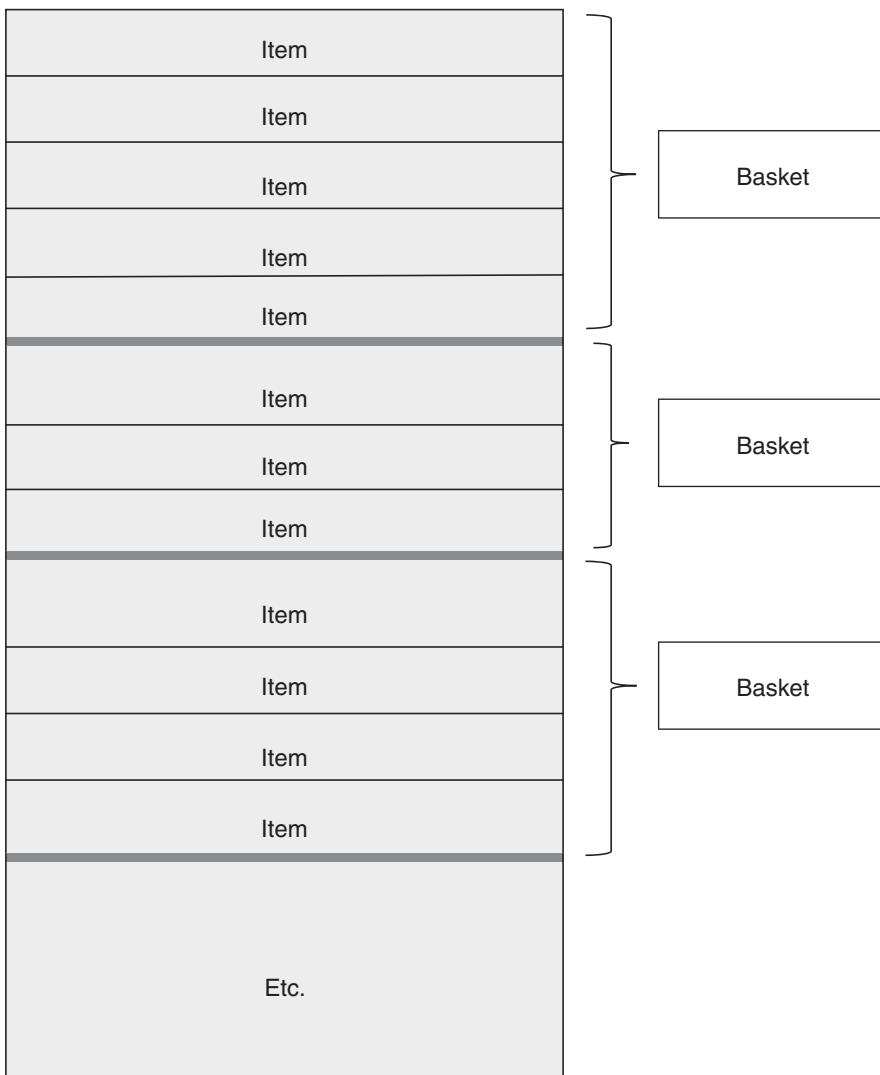


Figure 8.1 Market-baskets in main memory.

Example 7

One example of a market-basket file could look like:

{23, 45, 11001} {13, 48, 92, 145, 222} {...}

Here, the character “{” begins a basket and the character “}” ends it. The items in a basket are represented by integers, and are separated by commas.

Since such a file (Example 7) is typically large, we can use MapReduce or a similar tool to divide the work among many machines. But non-trivial changes need to be made to the frequent itemset counting algorithm to get the exact collection of itemsets that meet a global support threshold. This will be addressed in Section 8.4.3.

For now we shall assume that the data is stored in a conventional file and also that the size of the file of baskets is sufficiently large that it does not fit in the main memory. Thus, the principal cost is the time it takes to read data (baskets) from the disk. Once a disk block full of baskets is read into the main memory, it can be explored, generating all the subsets of size k . It is necessary to point out that it is logical to assume that the average size of a basket is small compared to the total number of all items. Thus, generating all the pairs of items from the market-baskets in the main memory should take less time than the time it takes to read the baskets from disk.

For example, if there are 25 items in a basket, then there are $\binom{25}{2} = 300$ pairs of items in the basket, and these can be generated easily in a pair of nested for-loops. But as the size of the subsets we want to generate gets larger, the time required grows larger; it takes approximately $n^k / k!$ time to generate all the subsets of size k for a basket with n items. So if k is very large then the subset generation time will dominate the time needed to transfer the data from the disk.

However, surveys have indicated that in most applications we need only small frequent itemsets. Further, when we do need the itemsets for a large size k , it is usually possible to eliminate many of the items in each basket as not able to participate in a frequent itemset, so the value of n reduces as k increases.

Thus, the time taken to examine each of the baskets can usually be assumed proportional to the size of the file. We can thus measure the running time of a frequent-itemset algorithm by the number of times each disk block of the data file is read. This, in turn, is characterized by the number of passes through the basket file that they make, and their running time is proportional to the product of the number of passes they make through the basket file and the size of that file.

Since the amount of data is fixed, we focus only on the number of passes taken by the algorithm. This gives us a measure of what the running time of a frequent-itemset algorithm will be.

8.3.2 Itemset Counting using Main Memory

For many frequent-itemset algorithms, main memory is the critical resource. An important computing step in almost all frequent-itemset algorithms is to maintain several different counts in each pass

of the data. For example, we might need to count the number of times that each pair of items occurs in baskets in pass 2. In the next pass along with maintaining the counts of 2-itemsets, we have to now compute frequency of 3-itemsets and so on. Thus, we need main memory space to maintain these counts.

If we do not have enough main memory to store each of the counts at any pass then adding 1 to count of any previous itemset may involve loading the relevant page with the counts from secondary memory. In the worst case, this swapping of pages may occur for several counts, which will result in thrashing. This would make the algorithm several orders of magnitude slower than if we were certain to find each count in main memory. In conclusion, we need counts to be maintained in the main memory. This sets a limit on how many items a frequent-itemset algorithm can ultimately deal with. This number of different things we can count is, thus, limited by the main memory.

The naive way of counting a frequent k -itemset is to read the file once and count in main memory the occurrences of each k -itemset. Let n be the number of items. The number of itemsets of size $1 \leq k \leq n$ is given by

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Example 8

Suppose we need to count all pairs of items (2-itemset) in some step, and there are n items. We thus need space to store $n(n-1)/2$ pairs. **This algorithm will fail if (#items)² exceeds main memory.**

Consider an e-commerce enterprise like Amazon. The number of items can be around 100K or 10B (Web pages). Thus, assuming 10^5 items and counts are 4-byte integers, the number of pairs of items is

$$\frac{10^5(10^5 - 1)}{2} = 5 * 10^9$$

Therefore, $2 * 10^{10}$ (20 GB) of memory is needed. Thus, in general, if integers take 4 bytes, we require approximately $2n^2$ bytes. If our machine has 2 GB, or 231 bytes of main memory, then we require $n \leq 2^{15}$ or approximately $n < 33,000$.

It is important to point out here that it is sufficient to focus on counting pairs, because the probability of an itemset being frequent drops exponentially with size while the number of itemsets grows more slowly with size. This argument is quite logical. The number of items, while possibly very large, is rarely so large we cannot count all the singleton sets in main memory at the same time. For larger sets like triples, quadruples, for frequent-itemset analysis to make sense, the result has to be a small number of sets, or these itemsets will lose their significance. Thus, in practice, the support threshold is set high enough that it is only a rare set that is k -frequent ($k \geq 2$). Thus, we expect to find more frequent pairs than frequent triples, more frequent triples than frequent quadruples, and so on. Thus, we can safely conclude that maximum main memory space is required for counting frequent pairs. We shall, thus, only concentrate on algorithms for counting pairs.

8.3.3 Approaches for Main Memory Counting

Before we can discuss approaches for counting of pairs in the main memory we have to first, discuss how items in the baskets are represented in the memory. As mentioned earlier, it is more space-efficient to represent items by consecutive positive integers from 1 to n , where n is the number of distinct items.

But items will mostly be names or strings of the form “pencil”, “pen”, “crayons”, etc. We will, therefore, need a hash table that translates items as they appear in the file to integers. That is, each time we see an item in the file, we hash it. If it is already in the hash table, we can obtain its integer code from its entry in the table. If the item is not there, we assign it the next available number (from a count of the number of distinct items seen so far) and enter the item and its code into the table.

8.3.3.1 The Triangular-Matrix Method

Coding items as integers saves some space in the main memory. Now we need to store the pair counts efficiently. The main issue is that we should count a pair $\{i, j\}$ in only one place. One way is to order the pair so that $i < j$ and only use the entry $A[i, j]$ in a two-dimensional (2-D) array A . But half the array will be useless in this method. Thus, we need either the upper triangular matrix or the lower triangular matrix. A triangular matrix can be very efficiently stored in a one-dimensional (1-D) **Triangular Array**.

Keep pair counts in lexicographic order:

1. $\{1, 2\}, \{1, 3\}, \dots, \{1, n\}, \{2, 3\}, \{2, 4\}, \dots, \{2, n\}, \dots \{n - 2, n - 1\}, \{n - 2, n\}, \{n - 1, n\}$
2. Pair $\{i, j\}$ is at position $(i - 1)(n - i / 2) + j - 1$

Figure 8.2 shows a sample of a triangular matrix storing a set of pairs.

	1,2	1,3	1,4	1,5
		2,3	2,4	2,5
			3,4	3,5
				4,5

Pair $\{i, j\}$ is at position $(i - 1)(n - i / 2) + j - 1$

- $\{1,2\} = 0 + 2 - 1 = 1$
- $\{1,3\} = 0 + 3 - 1 = 2$
- $\{1,4\} = 0 + 4 - 1 = 3$
- $\{1,5\} = 0 + 5 - 1 = 4$
- $\{2,3\} = (2 - 1) * (5 - 2/2) + 3 - 2 = 5$
- $\{2,4\} = (2 - 1) * (5 - 2/2) + 4 - 2 = 6$
- $\{2,5\} = (2 - 1) * (5 - 2/2) + 5 - 2 = 7$
- $\{3,4\} = (3 - 1) * (5 - 3/2) + 4 - 2 = 8$
- $\{3,5\} = 9$
- $\{4,5\} = 10$

Pair	1,2	1,3	1,4	1,5	2,3	2,4	2,5	3,4	3,5	4,5
Position	1	2	3	4	5	6	7	8	9	10

Figure 8.2 Triangular matrix and its equivalent 1-D representation.

8.3.3.2 The Triples Method

We can consider one more approach to storing counts that may be more useful when the fraction of the possible pairs of items that actually appear in some basket may not be too high.

We can use a hash table with pairs $\{i, j\}$, where $1 \leq i \leq j \leq n$, as keys and the counts as values. This results in triplets of the form $[i, j, c]$, meaning that the count of pair $\{i, j\}$ with $i < j$ is c .

Unlike the triangular matrix approach, in this approach we only store pairs with non-zero count value. If items and counts are represented by 4-byte integers, this approach requires 12 bytes for each pair with non-zero value plus some overhead for the hash table.

We call this approach the triples method of storing counts. Unlike the triangular matrix, the triples method does not require us to store anything if the count for a pair is 0. On the other hand, the triples method requires us to store three integers, rather than one, for every pair that does appear in some basket.

It is easy to see that the hash table approach requires less memory than the triangular matrix approach if less than one-third of possible pairs actually occur.

Let us assume that a supermarket has 100,000 items, which means that there are about 5 billion possible pairs. With the triangular matrix approach, we would need about 20 GB of main memory to hold the pair counts. If all possible pairs actually occur, the hash table approach requires about 60 GB of main memory. If only 1 billion of the possible pairs occur, however, the hash table approach would only require about 12 GB of main memory.

How about frequent triples? Since there are $[n(n-1)(n-2)]/6$ triples, the naive algorithm with the triangular matrix approach would require $[4n(n-1)(n-2)]/6$ bytes of main memory, which is more than 666 TB for the example above. For this reason, we want to avoid counting itemsets that will turn out to be infrequent at the end.

Example 9

Suppose there are 100,000 items and 10,000,000 baskets of 10 items each. Then the integer counts required by triangular matrix method are

$$\binom{1000}{2} = 5 \times 10^9 \text{ (approximately)}$$

On the other hand, the total number of pairs among all the baskets is

$$10^7 \binom{n}{k} = 4.5 \times 10^8$$

Even in the extreme case that every pair of items appeared only once, there could be only 4.5×10^8 pairs with non-zero counts. If we used the triples method to store counts, we would need only three times

that number of integers or 1.35×10^9 integers. Thus, in this case, the triples method will surely take much less space than the triangular matrix.

However, even if there were 10 or a 100 times as many baskets, it would be normal for there to be a sufficiently uneven distribution of items that we might still be better off using the triples method. That is, some pairs would have very high counts, and the number of different pairs that occurred in one or more baskets would be much less than the theoretical maximum number of such pairs.

8.3.4 Monotonicity Property of Itemsets

When n is large, a naive approach to generate and count the supports of all sets over the database cannot be achieved within a reasonable period of time. Typically most enterprises deal with thousands of items and thus 2^n , which is the number of subsets possible, is prohibitively high.

Instead, we could limit ourselves to those sets that occur at least once in the database by generating only those subsets of all transactions in the database. Of course, for large transactions, this number could still be too large. As an optimization, we could generate only those subsets of at most a given maximum size. This technique also suffers from massive memory requirements for even a medium sized database. Most other efficient solutions perform a more directed search through the search space. During such a search, several collections of *candidate sets* are generated and their supports computed until all frequent sets have been generated. Obviously, the size of a collection of candidate sets must not exceed the size of available main memory. Moreover, it is important to generate as few candidate sets as possible, since computing the supports of a collection of sets is a time-consuming procedure. In the best case, only the frequent sets are generated and counted. Unfortunately, this ideal is impossible in general. The main underlying property exploited by most algorithms is that support is monotone decreasing with respect to extension of a set.

Property 1 (Support Monotonicity): Given a database of transactions D over I and two sets $X, Y \subseteq I$. Then,

$$X, Y \subseteq I \Rightarrow \text{support}(Y) \leq \text{support}(X)$$

Hence, if a set is infrequent, all of its supersets must be infrequent, and vice versa, if a set is frequent, all of its subsets must be frequent too. In the literature, this monotonicity property is also called the *downward-closure property*, since the set of frequent sets is downward closed with respect to set inclusion. Similarly, the set of infrequent sets is upward closed.

The downward-closure property of support also allows us to compact the information about frequent itemsets. First, some definitions are given below:

1. An itemset is *closed* if none of its immediate itemset has the same count as the itemset.
2. An itemset is *closed frequent* if it is frequent and closed.
3. An itemset is *maximal frequent* if it is frequent and none of its immediate superset is frequent.

For example, assume we have items = {apple, beer, carrot} and the following baskets:

1. {apple, beer}
2. {apple, beer}
3. {beer, carrot}
4. {apple, beer, carrot}
5. {apple, beer, carrot}

Assume the support threshold $s = 3$.

Table 8.4 Indicating Frequent, Closed and Maximal Itemsets

Itemset	Count	Frequent?	Closed?	Closed Freq?	Max Freq?
{apple}	4	Yes	No	No	No
{beer}	5	Yes	Yes	Yes	No
{carrot}	3	Yes	No	No	No
{apple, beer}	4	Yes	Yes	Yes	Yes
{apple, carrot}	2	No	No	No	No
{beer, carrot}	3	Yes	Yes	Yes	Yes
{apple, beer, carrot}	2	No	Yes	No	No

From Table 8.4, we see that there are five frequent itemsets, of which only three are closed frequent, of which in turn only two are maximal frequent. The set of all maximal frequent itemsets is a subset of the set of all closed frequent itemsets, which in turn is a subset of the set of all frequent itemsets. Thus, maximal frequent itemsets is the most compact representation of frequent itemsets. In practice, however, closed frequent itemsets may be preferred since they also contain not just the frequent itemset information, but also the exact count.

8.3.5 The Apriori Algorithm

In this section we shall concentrate on finding the frequent pairs only. We have previously discussed that counting of pairs is indeed a more memory-intensive task. If we have enough main memory to count all pairs, using either triangular matrix or triples, then it is a simple matter to read the file of baskets in a single pass. For each basket, we use a double loop to generate all the pairs. Each time we

generate a pair, we add 1 to its count. At the end, we examine all pairs to see which have counts that are equal to or greater than the support threshold s ; these are the frequent pairs.

However, this naive approach fails if there are too many pairs of items to count them all in the main memory. The Apriori algorithm which is discussed in this section uses the monotonicity property to reduce the number of pairs that must be counted, at the expense of performing two passes over data, rather than one pass.

The Apriori algorithm for finding frequent pairs is a two-pass algorithm that limits the amount of main memory needed by using the downward-closure property of support to avoid counting pairs that will turn out to be infrequent at the end.

Let s be the minimum support required. Let n be the number of items. In the first pass, we read the baskets and count in main memory the occurrences of each item. We then remove all items whose frequency is lesser than s to get the set of frequent items. This requires memory proportional to n .

In the second pass, we read the baskets again and count in main memory only those pairs where both items are frequent items. This pass will require memory proportional to square of **frequent** items only (for counts) plus a list of the frequent items (so you know what must be counted). Figure 8.3 indicates the main memory in the two passes of the Apriori algorithm.

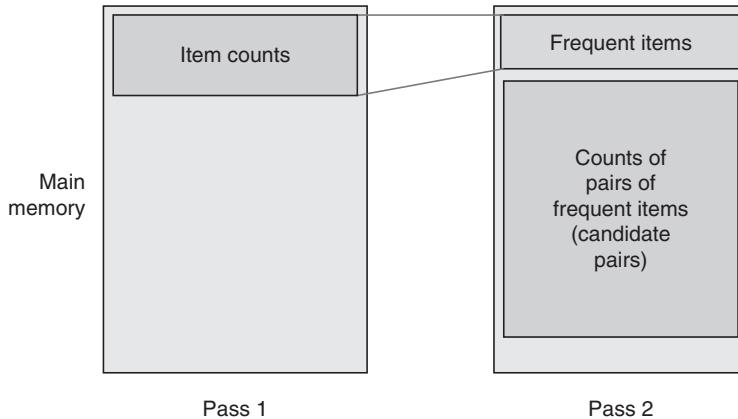


Figure 8.3 Main memory map in Apriori algorithm.

8.3.5.1 The First Pass of Apriori

Two tables are created in this pass. The first table, if necessary, translates item names into integers from 1 to n , as it will be optimal to use numbers than long varying length string names. The other table is an array of counts; the i^{th} array element counts the occurrences of the item numbered i .

Initially, the counts for all the items are 0. As we read baskets, we look at each item in the basket and translate its name into an integer. Next, we use that integer to index into the array of counts, and we add 1 to the integer found there.

After the first pass, we examine the counts of the items to determine which of them are frequent as singletons. It will normally be the case that the number of frequent singletons will definitely be much lesser than the number of items. Thus, we employ a trick for the next pass. For the second pass of Apriori, we create a new numbering from 1 to m for just the frequent items. This table is an array indexed 1 to n , and the entry for i is either 0, if item i is not frequent, or a unique integer in the range 1 to m if item i is frequent. We shall refer to this table as the frequent-items table.

8.3.5.2 The Second Pass of Apriori

During the second pass, we count all the pairs that consist of two frequent items. The space required on the second pass is $2m^2$ bytes, rather than $2n^2$ bytes, if we use the triangular-matrix method for counting. Notice that the renumbering of just the frequent items is necessary if we are to use a triangular matrix of the right size. The complete set of main-memory structures used in the first and second passes of this improved Apriori algorithm is shown in Fig. 8.4.

Let us quantify the benefit of eliminating infrequent items; if only half the items are frequent we need one quarter of the space to count. Likewise, if we use the triples method, we need to count only those pairs of two frequent items that occur in at least one basket.

The mechanics of the second pass are as follows:

1. For each basket, look in the frequent-items table to see which of its items are frequent.
2. In a double loop, generate all pairs of frequent items in that basket.
3. For each such pair, add one to its count in the data structure used to store counts.

Finally, at the end of the second pass, examine the structure of counts to determine which pairs are frequent.

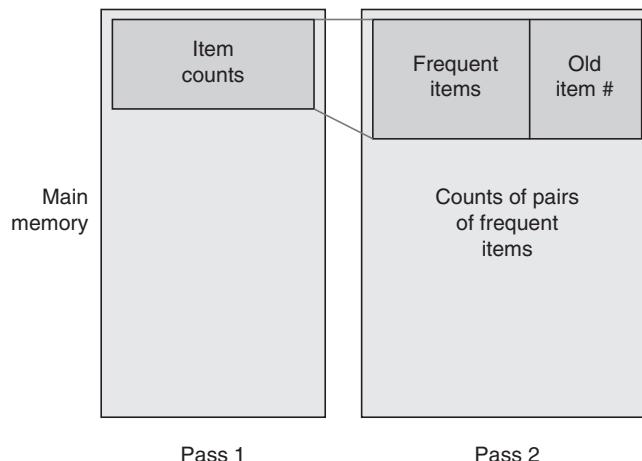


Figure 8.4 Improved Apriori algorithm.

8.3.5.3 Finding All Frequent Itemsets

The same technique as used for finding frequent pairs without counting all pairs can be extended to find larger frequent itemsets without an exhaustive count of all sets. In the Apriori algorithm, one pass is taken for each set-size k . If no frequent itemsets of a certain size are found, then monotonicity tells us there can be no larger frequent itemsets, so we can stop. The pattern of moving from one size k to the next size $k + 1$ can be summarized as follows. For each size k , there are two sets of itemsets:

1. C_k is the set of candidate itemsets of size k – the itemsets that we must count in order to determine whether they are in fact frequent.
 2. L_k is the set of truly frequent itemsets of size k .

The pattern of moving from one set to the next and one size to the next is depicted in Fig. 8.5.

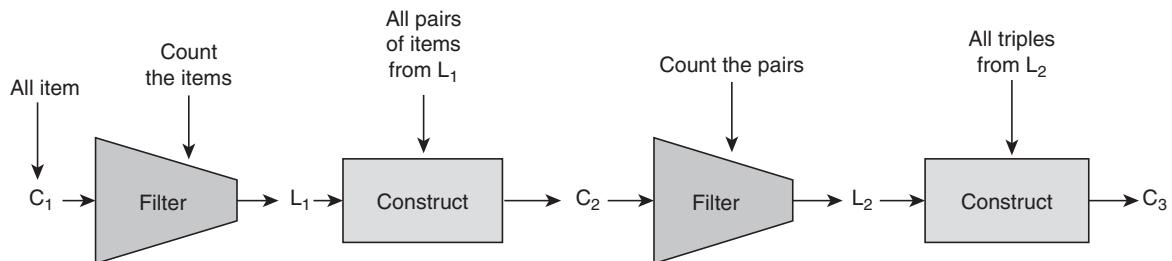


Figure 8.5 Candidate generation: General Apriori algorithm.

Example 10

Let $C_1 = \{ \{b\} \{c\} \{j\} \{m\} \{n\} \{p\} \}$. Then

1. Count the support of itemsets in C_1
 2. Prune non-frequent: $L_1 = \{ b, c, j, m \}$
 3. Generate $C_2 = \{ \{b,c\} \{b,j\} \{b,m\} \{c,j\} \{c,m\} \{j,m\} \}$
 4. Count the support of itemsets in C_2
 5. Prune non-frequent: $L_2 = \{ \{b,m\} \{b,c\} \{c,m\} \{c,j\} \}$
 6. Generate $C_3 = \{ \{b,c,m\} \{b,c,j\} \{b,m,j\} \{c,m,j\} \}$
 7. Count the support of itemsets in C_3
 8. Prune non-frequent: $L_3 = \{ \{b,c,m\} \}$

Example 11

Assume we have items = {*a*, *b*, *c*, *d*, *e*} and the following baskets:

1. {*a*, *b*}
2. {*a*, *b*, *c*}
3. {*a*, *b*, *d*}
4. {*b*, *c*, *d*}
5. {*a*, *b*, *c*, *d*}
6. {*a*, *b*, *d*, *e*}

Let the support threshold $s = 3$. The Apriori algorithm passes as follows:

1.
 - (a) Construct $C_1 = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}\}$.
 - (b) Count the support of itemsets in C_1 .
 - (c) Remove infrequent itemsets to get $L_1 = \{\{a\}, \{b\}, \{c\}, \{d\}\}$.
2.
 - (a) Construct $C_2 = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}\}$.
 - (b) Count the support of itemsets in C_2 .
 - (c) Remove infrequent itemsets to get $L_2 = \{\{a, b\}, \{a, d\}, \{b, c\}, \{b, d\}\}$.
3.
 - (a) Construct $C_3 = \{\{a, b, c\}, \{a, b, d\}, \{b, c, d\}\}$. Note that we can be more careful here with the rule generation. For example, we know $\{b, c, d\}$ cannot be frequent since $\{c, d\}$ is not frequent. That is, $\{b, c, d\}$ should not be in C_3 since $\{c, d\}$ is not in L_2 .
 - (b) Count the support of itemsets in C_3 .
 - (c) Remove infrequent itemsets to get $L_3 = \{\{a, b, d\}\}$.
4. Construct $C_4 = \{\text{empty set}\}$.

8.4 Handling Larger Datasets in Main Memory

We can clearly understand that the initial candidate set generation especially for the large 2-itemsets is the key to improve the performance of the Apriori algorithm. This algorithm is fine as long as the step for the counting of the candidate pairs has enough memory that it can be accomplished without excessive moving of data between disk and main memory.

Another performance related issue is the amount of data that has to be scanned during large itemset discovery. A straightforward implementation would require one pass over the database of all transactions for each iteration. Note that as k increases, not only is there a smaller number of large k itemsets, but

there are also fewer transactions containing any large k itemsets. Reducing the number of transactions to be scanned and trimming the number of items in each transaction can improve the data mining efficiency in later stages.

Several algorithms have been proposed to cut down on the size of candidate set C_2 . In this section, we discuss the Park–Chen–Yu (PCY) algorithm, which takes advantage of the fact that in the first pass of Apriori there is typically lots of main memory not needed for the counting of single items. Then we look at the Multistage algorithm, which uses the PCY trick and also inserts extra passes to further reduce the size of C_2 .

8.4.1 Algorithm of Park–Chen–Yu

In the Apriori algorithm, in each pass we use the set of large itemsets L_i to form the set of candidate large itemsets C_{i+1} by joining L_i with L_i on $(i-1)$ common items for the next pass. We then scan the database and count the support of each item set in C_{i+1} so as to determine L_{i+1} . As a result, in general, the more itemsets in C_i the higher the processing cost for determining L_i will be. An algorithm designed by Park, Chen, and Yu called as DHP (standing for direct hashing and pruning) is so designed that it will reduce the number of itemsets to be explored in C_i in initial iterations significantly. The corresponding processing cost to determine L_i from C_i is, therefore, reduced. This algorithm is also popularly known as PCY algorithm in literature, named so because of its authors.

In essence, the PCY algorithm uses the technique of hashing to filter out unnecessary itemsets for next candidate itemset generation. When the support of candidate k -itemsets is counted by scanning the database, PCY accumulates information about candidate $(k+1)$ -itemsets in advance in such a way that all possible $(k+1)$ -itemsets of each transaction after some pruning are hashed to a hash table. Each bucket in the hash table consists of a number to represent how many itemsets have been hashed to this bucket thus far. We note that based on the resulting hash table a bit vector can be constructed, where the value of one bit is set to 1 if the number in the corresponding entry of the hash table is greater than or equal to s , the minimum support value. This bit vector can be used later to greatly reduce the number of itemsets in C_i .

The PCY algorithm does need any extra space to store the hash tables or the bit vectors. It exploits the observation that there may be much unused space in main memory on the first pass. If there are a million items and gigabytes of main memory, which is quite typical these days, we do not need more than 10% of the main memory for the two tables, one a translation table from item names to small integers and second an array to count those integers.

The PCY algorithm then uses the still available space for an unusual type of hash table. The buckets of this hash table store integers rather than sets of keys. Pairs of items are hashed to buckets of this hash table. As we examine a basket during the first pass, we not only add 1 to the count for each item in the basket, but also generate all possible pairs (2-itemsets). We hash each pair using a hash function, and we add 1 to the bucket into which that pair hashes. Note that the pair itself does not go into the bucket; the pair only affects the single integer in the bucket.

Algorithm

```

FOR (each basket):
    FOR (each item in the basket) :
        add 1 to item's count;
    FOR (each pair of items):
        { hash the pair to a bucket;
        add 1 to the count for that bucket;}
```

At the end of the first pass, each bucket has a count, which is the sum of the counts of all the pairs that hash to that bucket. If the count of a bucket is at least as great as the support threshold s , it is called a frequent bucket. We can say nothing about the pairs that hash to a frequent bucket; they could all be frequent pairs from the information available to us. But if the count of the bucket is less than s (an infrequent bucket), we know no pair that hashes to this bucket can be frequent, even if the pair consists of two frequent items. This fact gives us an advantage on the second pass.

In Pass 2 we only count pairs that hash to frequent buckets.

Algorithm

Count all pairs $\{i, j\}$ that meet the conditions for being a **candidate pair**:

1. Both i and j are frequent items
2. The pair $\{i, j\}$ hashes to a bucket whose bit in the bit vector is 1 (i.e., a **frequent bucket**)

Both the above conditions are necessary for the pair to have a chance of being frequent. Figure 8.6 indicates the memory map.

Depending on the data and the amount of available main memory, there may or may not be a benefit in using the hash table on pass 1. In the worst case, all buckets are frequent, and the PCY algorithm counts exactly the same pairs as Apriori does on the second pass. However, typically most of the buckets will be infrequent. In that case, PCY reduces the memory requirements of the second pass.

Let us consider a typical situation. Suppose we have 1 GB of main memory available for the hash table on the first pass. Let us say the dataset has a billion baskets, each with 10 items. A bucket is an

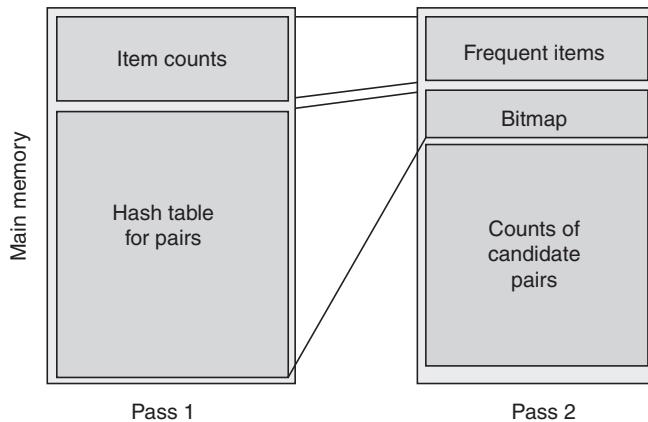


Figure 8.6 Memory map in PCY algorithm.

integer, typically 4 bytes, so we can maintain a quarter of a billion buckets. The number of pairs in all the baskets is

$$10^9 \times \binom{10}{2} = 4.5 \times 10^{10} \text{ pairs}$$

This number is also the sum of the counts in the buckets. Thus, the average count is about 180. If the support threshold s is around 180 or less, we might expect few buckets to be infrequent. However, if s is much larger, say 1000, then it must be that the great majority of the buckets are infrequent. The greatest possible number of frequent buckets is, thus, about 45 million out of the 250 million buckets.

Between the passes of PCY, the hash table is reduced to a bitmap, with one bit for each bucket. The bit is 1 if the bucket is frequent and 0 if it is not. Thus, integers of 4 bytes are replaced by single bits. Thus, the bitmap occupies only 1/32 of the space that would otherwise be available to store counts. However, if most buckets are infrequent, we expect that the number of pairs being counted on the second pass will be much smaller than the total number of pairs of frequent items. Thus, PCY can handle large datasets without thrashing during the second pass, while the Apriori algorithm would have run out of main memory space resulting in thrashing.

There is one more issue that could affect the space requirement of the PCY algorithm in the second pass. In the PCY algorithm, the set of candidate pairs is sufficiently irregular, and hence we cannot use the triangular-matrix method for organizing counts; we must use a table of counts. Thus, it does not make sense to use PCY unless the number of candidate pairs is reduced to at most one-third of all possible pairs. Passes of the PCY algorithm after the second can proceed just as in the Apriori algorithm, if they are needed.

Further, in order for PCY to be an improvement over Apriori, a good fraction of the buckets on the first pass must not be frequent. For if most buckets are frequent, the algorithm does not eliminate many

pairs. Any bucket to which even one frequent pair hashes will itself be frequent. However, buckets to which no frequent pair hashes could still be frequent if the sum of the counts of the pairs that do hash there exceeds the threshold s .

To a first approximation, if the average count of a bucket is less than s , we can expect at least half the buckets not to be frequent, which suggests some benefit from the PCY approach. However, if the average bucket has a count above s , then most buckets will be frequent.

Suppose the total number of occurrences of pairs of items among all the baskets in the dataset is P . Since most of the main memory M can be devoted to buckets, the number of buckets will be approximately $M/4$. The average count of a bucket will then be $4P/M$. In order that there be many buckets that are not frequent, we need

$$\frac{4P}{M} < s \text{ or } M > \frac{4P}{s}$$

An example illustrates some of the steps of the PCY algorithm.

Example 12

Given: Database D ; minimum support = 2 and the following data.

TID	Items
1	1,3,4
2	2,3,5
3	1,2,3,5
4	2,5

Pass 1:

Step 1: Scan D along with counts. Also form possible pairs and hash them to the buckets. For example, {1,3}:2 means pair {1,3} hashes to bucket 2.

Itemset	Sup
{1}	2
{2}	3
{3}	3
{4}	1
{5}	3

T1	{1,3}:2, {1,4}:1, {3,4}:3
T2	{2,3}:1 , {2,5}:3, {3,5}:5
T3	{1,2}:4, {1,3}:2, {1,5}:5, {2,3}:1, {2,5}:3, {3,5}:5
T4	{2,5}:3

Step 2: Using the hash function as discussed in step 1 the bucket looks like the one shown below.

Bucket	1	2	3	4	5
Count	3	2	4	1	3

In between the passes:

We have C_1 : 1, 2, 3, 5 (only these singletons satisfy min. support 2). Further, by looking at the buckets we know that bucket 4 does not satisfy the min. support criteria. So the pair {1,2} that hashes to bucket 4 need not be considered for support counting in the next pass. Thus only the pairs {1,3}, {1,5}, {2,3}, {2,5} and {3,5} survive for next pass.

Example 13

Suppose we perform the PCY algorithm to find frequent pairs, with market-basket data meeting the following specifications:

1. The support threshold s is 10,000.
2. There are one million items, which are represented by the integers 0, 1, ..., 999999.
3. There are 250,000 frequent items, that is, items that occur 10,000 times or more.
4. There are one million pairs that occur 10,000 times or more.
5. There are P pairs that occur exactly once and consist of two frequent items.
6. No other pairs occur at all.
7. Integers are always represented by 4 bytes.
8. When we hash pairs, they distribute among buckets randomly, but as evenly as possible; that is, you may assume that each bucket gets exactly its fair share of the P pairs that occur once.

Suppose there are S bytes of main memory. In order to run the PCY algorithm successfully, we must keep the support count sufficiently large so that most buckets are infrequent. In addition, on the

second pass, there must be enough room to count all the candidate pairs. As a function of S , what is the largest value of P for which we can successfully run the PCY algorithm on this data?

It is left as an exercise to the reader that $S = 300,000,000$; $P = 750,000,000$ are possible values for S and P . This means that this is approximately (i.e., to within 10%) the largest possible value of P for that S .

8.4.2 The Multistage Algorithm

The Multistage algorithm improves upon PCY by using several successive hash tables to reduce the number of candidate pairs further. The tradeoff is that Multistage takes more than two passes to find the frequent pairs.

Instead of counting pairs on the second pass, as we do in Apriori or PCY, we could use the same bucketing technique (with a different hash function) on the second pass. To make the average counts even smaller on the second pass, we do not even have to consider a pair on the second pass unless it would be counted on the second pass of PCY; that is, the pair consists of two frequent items and also hashed to a frequent bucket on the first pass. This idea leads to the three-pass version of the Multistage algorithm for finding frequent pairs.

Figure 8.7 shows the main memory during the various passes of the multistage algorithm. Pass 1 is just like Pass 1 of PCY, and between Passes 1 and 2 we collapse the buckets to bits and select the

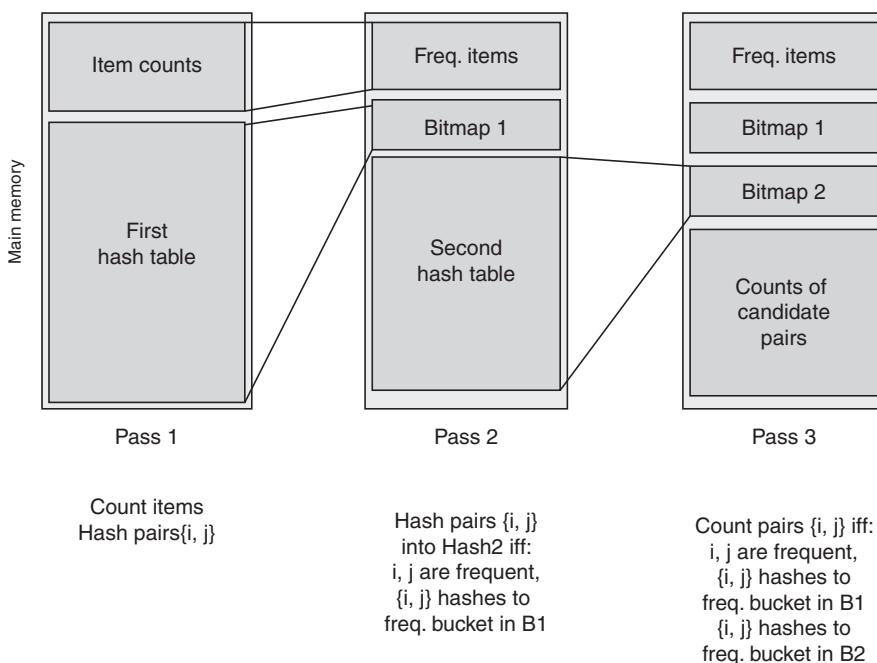


Figure 8.7 Memory map for multistage algorithm.

frequent items, also as in PCY. However, on Pass 2, we again use all available memory to hash pairs into as many buckets as will fit. Since the bitmap from the first hash table takes up 1/32 of the available main memory, the second hash table has almost as many buckets as the first.

Because there is a bitmap to store in the main memory on the second pass, and this bitmap compresses a 4-byte (32-bit) integer into one bit, there will be approximately 31/32 as many buckets on the second pass as on the first. On the second pass, we use a different hash function from that used on Pass 2. We hash a pair $\{i, j\}$ to a bucket and add one to the count there if and only if:

1. Both i and j are frequent items.
2. $\{i, j\}$ hashed to a frequent bucket on the first pass.

This decision is made by just consulting the bitmap. That is, we hash only those pairs we would count on the second pass of the PCY algorithm. Between the second and third passes, we condense the buckets of the second pass into another bitmap, which must be stored in main memory along with the first bitmap and the set of frequent items. On the third pass, we finally count the candidate pairs. In order to be a candidate, the pair $\{i, j\}$ must satisfy all of the following:

1. Both i and j are frequent items.
2. $\{i, j\}$ hashed to a frequent bucket on the first pass. This decision is made by consulting the first bitmap.
3. $\{i, j\}$ hashed to a frequent bucket on the second pass. This decision is made by consulting the second bitmap.

As with PCY, subsequent passes can construct frequent triples or larger itemsets, if desired, using the same method as Apriori. The third condition is the distinction between Multistage and PCY. It often eliminates many pairs that the first two conditions let through. One reason is that on the second pass, not every pair is hashed, so the counts of buckets tend to be smaller than on the first pass, resulting in many more infrequent buckets. Moreover, since the hash functions on the first two passes are different, infrequent pairs that happened to hash to a frequent bucket on the first pass have a good chance of hashing to an infrequent bucket on the second pass.

It might be obvious that it is possible to insert any number of passes between the first and last in the Multistage algorithm. We can have a large number of bucket-filling passes, each using a different hash function. As long as the first pass eliminates some of the pairs because they belong to a non-frequent bucket, the subsequent passes will eliminate a rapidly growing fraction of the pairs, until it is very unlikely that any candidate pair will turn out not to be frequent. However, there is a point of diminishing returns, since each bitmap requires about 1/32 of the memory.

There is a limiting factor that each pass must store the bitmaps from each of the previous passes. Eventually, there is not enough space left in main memory to do the counts. No matter how many passes we use, the truly frequent pairs will always hash to a frequent bucket, so there is no way to avoid counting them.

8.4.3 The Multihash Algorithm

Sometimes we can get most of the benefit of the extra passes of the Multistage algorithm in a single pass. This variation of PCY is called the Multihash algorithm. Instead of using two different hash tables on two successive passes, use two hash functions and two separate hash tables that share main memory on the first pass. This is depicted in Fig. 8.8.

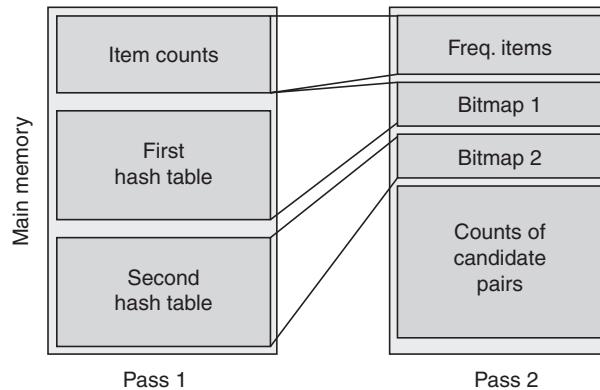


Figure 8.8 Memory map for Multihash algorithm.

The risk in using two hash tables on one pass is that each hash table has half as many buckets as the one large hash table of PCY. As long as the average count of a bucket for PCY is much lower than the support threshold, we can operate two half-sized hash tables and still expect most of the buckets of both hash tables to be infrequent. Thus, in this situation we might well choose the Multihash approach.

For the second pass of Multihash, each hash table is converted to a bitmap, as usual. The conditions for a pair $\{i, j\}$ to be in C_2 , and thus to require a count on the second pass, are the same as for the third pass of Multistage: i and j must both be frequent, and the pair must have hashed to a frequent bucket according to both hash tables.

Suppose, we run PCY on a dataset with the average bucket having a count $s/10$, where s is the support threshold. Then if we use the Multihash approach with two half-sized hash tables, the average count would be $s/5$. As a result, at most one-fifth of the buckets in either hash table could be frequent, and a random infrequent pair has at most probability $(1/5)^2 = 0.04$ of being in a frequent bucket in both hash tables.

In the same way, we can say that the upper bound on the infrequent pair being in a frequent bucket in one PCY hash table is at most $1/10$. That is, we might expect to have to count 2.5 times as many infrequent pairs in PCY as in the version of Multihash suggested above. We would, therefore, expect Multihash to have a smaller memory requirement for the second pass than would PCY. This analysis is just suggestive of the possibility that for some data and support thresholds, we can do better by running several hash functions in main memory at once.

Just as Multistage is not limited to two hash tables, we can divide the available main memory into as many hash tables as we like on the first pass of Multihash. The risk is that should we use too many hash tables, the average count for a bucket will exceed the support threshold. At that point, there may be very few infrequent buckets in any of the hash tables. This would nullify the benefit of using the hash table in the first place.

8.5 Limited Pass Algorithms

The algorithms for frequent itemsets discussed so far use one pass for each size of itemset we investigate. Recall that Apriori algorithm requires k passes to find frequent itemsets of size k . If the main memory is too small to hold the data and the space needed to count frequent itemsets of one size, there does not seem to be any way to avoid k passes to compute the exact collection of frequent itemsets.

In many applications, however, finding each and every frequent itemset is not important. In many cases, it may be impractical to consider too many frequent itemsets. All of them cannot result in some action taken by the enterprise. For instance, if we are looking for items purchased together at a supermarket, we are not going to run a sale or change the layout based on every frequent itemset we find. So it is quite sufficient to find most but not all of the frequent itemsets.

In this section we shall discuss some approximate algorithms that have been proposed to find all or most frequent itemsets using at most two passes. We begin with a simple sampling based algorithm. To get more accurate answers we discuss the Algorithm of Savasere, Omiecinski and Navathe (SON) which use two passes to get the exact answer. Further this algorithm can be easily implemented by MapReduce. Finally, Toivonen's algorithm uses two passes on average, gets an exact answer, but may, rarely, not terminate in finite time.

8.5.1 The Randomized Sampling Algorithm

The main idea for the Sampling algorithm is to select a small sample, one that fits in main memory, of the database of transactions and to determine the frequent itemsets from that sample. If those frequent itemsets form a superset of the frequent itemsets for the entire database, then we can determine the real frequent itemsets by scanning the remainder of the database in order to compute the exact support values for the superset itemsets. A superset of the frequent itemsets can usually be found from the sample by using, for example, the Apriori algorithm, with a lowered minimum support. In some rare cases, some frequent itemsets may be missed and a second scan of the database is needed.

Thus, instead of using the entire file of baskets, we could pick a random subset of the baskets and pretend that it is the entire dataset. We must adjust the support threshold to reflect the smaller number of baskets. For instance, if the support threshold for the full dataset is s and we choose a sample of 1% of the baskets, then we should examine the sample for itemsets that appear in at least $s/100$ of the baskets.

The safest way to pick the sample is to read the entire dataset, and for each basket, select that basket for the sample with some fixed probability p . Suppose there are m baskets in the entire file. At the end, we shall have a sample whose size is very close to $p \times m$ baskets. However, if the dataset is distributed evenly and all items appear in the baskets in random order then we do not even have to read the entire file. We can select the first $p \times m$ baskets for our sample.

However to avoid bias, in general, baskets should not be selected from one part of the file. For example, the transaction data in a supermarket may be organized by the date of sale, which means the first part of the file is older than the rest and the frequent itemsets found by analyzing such data may be less relevant. In such a case, older part of the dataset may not have any mention of the current items on sale and so the frequent itemsets would not reflect the actual situation.

Another situation could occur if the file is part of a distributed file system. In such cases, we can pick some chunks at random to serve as the sample. As another example, consider a file of food items offered at different restaurants. If each chunk comes from different restaurants, then picking chunks at random will give us a sample drawn from only a small subset of the restaurants. If different restaurants differ vastly in the food items they offer, this sample will be highly biased.

Selected baskets are loaded into the main memory. We use part of main memory to store these baskets. The balance of the main memory is used to execute one of the Frequent Itemset algorithms. The algorithm then runs Apriori or one of its improvements in the main memory. Given a support threshold s , an itemset is frequent in the sample dataset if its support is at least $p \times s$ in the sample dataset.

Note that Apriori still requires K passes to find itemsets of size K , but there is no disk access since the baskets are in the main memory. Thus, the algorithm so far requires only a single pass over the data on disk.

As frequent itemsets of each size are discovered, they can be written out to disk; this operation and the initial reading of the sample from disk are the only disk I/O's the algorithm does. Of course, the algorithm will fail if the algorithm cannot be run in the amount of main memory left after storing the sample. If we need more main memory, then an option is to read the sample from disk for each pass. Since the sample is much smaller than the full dataset, we still avoid most of the disk I/O's that the algorithms discussed previously would use.

This simple algorithm cannot be relied upon either to produce all the itemsets that are frequent in the whole dataset, nor will it produce only itemsets that are frequent in the whole.

The random sampling algorithm produces two kinds of errors:

1. *false negatives*: itemsets that are frequent in the full dataset but infrequent in the sample dataset.
2. *false positives*: itemsets that are infrequent in the full dataset but frequent in the sample dataset.

If the sample is large enough, there are unlikely to be serious errors. In such a case, an itemset whose support is much larger than the threshold will almost surely be identified from a random sample, and an itemset whose support is much less than the threshold is extremely unlikely to appear frequent in the sample. However, an itemset whose support in the whole is very close to the threshold is as likely to be frequent in the sample as not.

We can eliminate the false positives by making a second pass over the full dataset and making sure that all the itemsets that are frequent in the sample dataset are frequent in the full dataset. To do this in a single pass, we need to be able to count all frequent itemsets of all sizes at once in the main memory. This is possible since the support threshold s is set to be high enough to make the number of candidate itemsets small.

We cannot eliminate false negatives. However, we can reduce their number by lowering the support threshold during the first pass of the algorithm to, say, $0.9s$. This requires more main memory and increases the number of false positives, but it catches more truly frequent itemsets. The second pass of the algorithm will then eliminate the false positives, leaving us with, hopefully, very few false negatives.

8.5.2 The Algorithm of Savasere, Omiecinski and Navathe

The idea behind Savasere, Omiecinski and Navathe (SON) Algorithm also called as Partition algorithm is summarized below. If we are given a database with a small number of potential large itemsets, say, a few thousand, then the support for all of them can be tested in one scan and the actual large itemsets can be discovered. Clearly, this approach will work only if the given set contains all actual large itemsets.

Partition algorithm accomplishes this in two scans of the database. In one scan, it generates a set of all potentially large itemsets by scanning the database once. This set is a superset of all large itemsets, that is, it may contain false positives. But no false negatives are reported. During the second scan, counters for each of these itemsets are set up and their actual support is measured in one scan of the database.

The SON algorithm divides the database into non-overlapping subsets; these are individually considered as separate databases and all large itemsets for that partition, called local frequent itemsets, are generated in one pass. The Apriori algorithm can then be used efficiently on each partition if it fits entirely in main memory. Partitions are chosen in such a way that each partition can be accommodated in the main memory. As such, a partition is read only once in each pass. The only caveat with the partition method is that the minimum support used for each partition has a slightly different meaning from the original value. The minimum support is based on the size of the partition rather than the size of the database for determining local frequent (large) itemsets. The actual support threshold value is the same as given earlier, but the support is computed only for a partition.

At the end of pass one, we take the union of all frequent itemsets from each partition. This forms the global candidate frequent itemsets for the entire database. When these lists are merged, they may contain some false positives. That is, some of the itemsets that are frequent (large) in

one partition may not qualify in several other partitions and hence may not exceed the minimum support when the original database is considered. Note that there are no false negatives; no large itemsets will be missed.

The global candidate large itemsets identified in pass one are verified in pass two; that is, their actual support is measured for the entire database. At the end of phase two, all global large itemsets are identified. The Partition algorithm lends itself naturally to a parallel or distributed implementation for better efficiency. Further improvements to this algorithm have been suggested in the literature.

The implementation details for the SON algorithm are as follows:

1. The first pass of the SON algorithm is very similar to the first pass of the random sampling algorithm.
2. Instead of randomly sampling the dataset with a fixed probability p , the SON algorithm divides the dataset into chunks of size $1/p$. It then loads the baskets into the main memory and runs the Apriori or one of its improvements in the main memory. Given a support threshold s , an itemset is frequent in a chunk if its support is at least ps in the chunk.

The fact that this algorithm does not generate any false negatives arises from the following theorem and its corollary.

Theorem

If an itemset I is infrequent in every chunk, I is infrequent in the full dataset.

Proof: Since I is infrequent in every chunk, $\text{supp}(I) < ps$ in each chunk. Since there are $1/p$ chunks,

$$\text{supp}(I) < \left(\frac{1}{p}\right)ps$$

in the full dataset.

The contrapositive of the theorem follows:

Corollary: If an itemset I is frequent in the full dataset, I is frequent in at least one chunk.

This guarantees that all truly frequent itemsets are in the set of itemsets that are frequent in at least one chunk. Therefore, if we store all itemsets that are frequent in at least one chunk, we know that we will not have any false negative.

The second pass of the SON algorithm is exactly the same as the second pass of the random sampling algorithm: the algorithm goes over the full dataset and makes sure that all the itemsets that are frequent in at least one chunk are frequent in the full dataset.

8.5.3 The SON Algorithm and MapReduce

The SON algorithm lends itself well to distributed data mining and thus is very suitable to use for Big Data mining. Each of the chunks can be distributed and processed in parallel, and the frequent itemsets from each chunk are then combined to form the candidates.

We can distribute the candidates to many processors, have each processor count the support for each candidate in a subset of the baskets, and finally sum those supports to get the support for each candidate itemset in the whole dataset.

This process can ideally be executed using the MapReduce paradigm. It can be implemented in two sets of MapReduce. We shall summarize this MapReduce–MapReduce sequence below.

1. **First Map Function:** Take the assigned subset of the baskets and find the itemsets frequent in the subset using the simple randomized sampling algorithm of Section 8.5.1. Lower the support threshold from s to ps if each Map task gets fraction p of the total input file. The output is a set of key–value pairs $(F, 1)$, where F is a frequent itemset from the sample. The key–value pair is always 1 and is irrelevant.
2. **First Reduce Function:** Each Reduce task is assigned a set of keys, which are itemsets. The value is ignored, and the Reduce task simply produces those keys (itemsets) that appear one or more times. Thus, the output of the first Reduce function is the candidate itemsets.
3. **Second Map Function:** The Map tasks for the second Map function take all the output from the first Reduce Function (the candidate itemsets) and a portion of the input data file. Each Map task counts the number of occurrences of each of the candidate itemsets among the baskets in the portion of the dataset that it was assigned. The output is a set of key–value pairs (C, v) , where C is one of the candidate sets and v is the support for that itemset among the baskets that were input to this Map task.
4. **Second Reduce Function:** The Reduce tasks take the itemsets they are given as keys and sum the associated values. The result is the total support for each of the itemsets that the Reduce task was assigned to handle. Those itemsets whose sum of values is at least s are frequent in the whole dataset, so the Reduce task outputs these itemsets with their counts. Itemsets that do not have total support of at least s are not transmitted to the output of the Reduce task. The following text box depicts the MapReduce pseudocode for the SON algorithm.

Algorithm

The first map-reduce step:

```
map(key, value):  
    //value is a chunk of the full dataset  
    Count occurrences of itemsets in the chunk.
```

```
for itemset in itemsets:  
    if supp(itemset) >= p*s  
        emit(itemset, null)  
    reduce(key, values):  
        emit(key, null)
```

The second map-reduce step:

```
map(key, value):  
    // value is the candidate itemsets and a chunk  
    // of the full dataset  
    Count occurrences of itemsets in the chunk.  
    for itemset in itemsets:  
        emit(itemset, supp(itemset))  
  
    reduce(key, values):  
        result = 0  
        for value in values:  
            result += value  
        if result >= s:  
            emit(key, result)
```

8.5.4 Toivonen's Algorithm

Toivonen's algorithm makes only one full pass over the database. The idea is to pick a random sample, use it to determine all association rules that probably hold in the whole database, and then to verify the results with the rest of the database. The algorithm thus produces exact association rules in one full pass over the database. In those rare cases where the sampling method does not produce all association rules, the missing rules can be found in a second pass.

The algorithm will give neither false negatives nor positives, but there is a small yet non-zero probability that it will fail to produce any answer at all. In that case, it needs to be repeated until it gives an answer. However, the average number of passes needed before it produces all and only the frequent itemsets is a small constant.

Toivonen's algorithm begins by selecting a small sample of the input dataset, and finding from it the candidate frequent itemsets. It uses the random sampling algorithm, except that it is essential that

the threshold be set to something less than its proportional value. That is, if the support threshold for the whole dataset is s , and the sample size is fraction p , then when looking for frequent itemsets in the sample, use a threshold such as $0.9ps$ or $0.7ps$. The smaller we make the threshold, the more main memory we need for computing all itemsets that are frequent in the sample, but the more likely we are to avoid the situation where the algorithm fails to provide an answer.

Having constructed the collection of frequent itemsets for the sample, we next construct the negative border. This is the collection of itemsets that are not frequent in the sample, but all of their immediate subsets (subsets constructed by deleting exactly one item) are frequent in the sample.

We shall define the concept of “negative border” before we explain the algorithm.

The negative border with respect to a frequent itemset, S , and set of items, I , is the minimal itemsets contained in PowerSet (I) and not in S . The basic idea is that the negative border of a set of frequent itemsets contains the closest itemsets that could also be frequent. Consider the case where a set X is not contained in the frequent itemsets. If all subsets of X are contained in the set of frequent itemsets, then X would be in the negative border. We illustrate this with the following example.

Example 14

Consider the set of items $I = \{A, B, C, D, E\}$ and let the combined frequent itemsets of size 1 to 3 be $S = \{\{A\}, \{B\}, \{C\}, \{D\}, \{AB\}, \{AC\}, \{BC\}, \{AD\}, \{CD\}, \{ABC\}\}$

1. The negative border is $\{\{E\}, \{BD\}, \{ACD\}\}$.
2. The set $\{E\}$ is the only 1-itemset not contained in S .
3. $\{BD\}$ is the only 2-itemset not in S but whose 1-itemset subsets are.
4. $\{ACD\}$ is the only 3-itemset whose 2-itemset subsets are all in S .

The negative border is important since it is necessary to determine the support for those itemsets in the negative border to ensure that no large itemsets are missed from analyzing the sample data. Support for the negative border is determined when the remainder of the database is scanned. If we find that an itemset X in the negative border belongs in the set of all frequent itemsets, then there is a potential for a superset of X to also be frequent. If this happens, then a second pass over the database is needed to make sure that all frequent itemsets are found.

To complete Toivonen’s algorithm, we make a pass through the entire dataset, counting all the itemsets that are frequent in the sample or are in the negative border. There are two possible outcomes.

Outcome 1: No member of the negative border is frequent in the whole dataset. In this case, we already have the correct set of frequent itemsets, which were found in the sample and also were found to be frequent in the whole.

Outcome 2: Some member of the negative border is frequent in the whole. Then it is difficult to decide whether there exists more larger sets that may become frequent when we consider a more larger sample. Thus, the algorithm terminates without a result. The algorithm must be repeated with a new random sample.

It is easy to see that the Toivonen's algorithm never produces a false positive, since it only outputs those itemsets that have been counted and found to be frequent in the whole. To prove that it never produces a false negative, we must show that when no member of the negative border is frequent in the whole, then there can be no itemset whatsoever, that is:

Frequent in the whole, but in neither the negative border nor the collection of frequent itemsets for the sample.

Suppose the above is not true. This means, there is a set S that is frequent in the whole, but not in the negative border and not frequent in the sample. Also, this round of Toivonen's algorithm produced an answer, which would certainly not include S among the frequent itemsets. By monotonicity, all subsets of S are also frequent in the whole.

Let T be a subset of S that is of the smallest possible size among all subsets of S that are not frequent in the sample. Surely T meets one of the conditions for being in the negative border: It is not frequent in the sample. It also meets the other condition for being in the negative border: Each of its immediate subsets is frequent in the sample. For if some immediate subset of T were not frequent in the sample, then there would be a subset of S that is smaller than T and not frequent in the sample, contradicting our selection of T as a subset of S that was not frequent in the sample, yet as small as any such set.

Now we see that T is both in the negative border and frequent in the whole dataset. Consequently, this round of Toivonen's algorithm did not produce an answer.

8.6

Counting Frequent Items in a Stream

We have seen in Chapter 6 that recently there has been much interest in data arriving in the form of continuous and infinite data streams, which arise in several application domains like high-speed networking, financial services, e-commerce and sensor networks.

We have seen that data streams possess distinct computational characteristics, such as unknown or unbounded length, possibly very fast arrival rate, inability to backtrack over previously arrived items (only one sequential pass over the data is permitted), and a lack of system control over the order in which the data arrive. As data streams are of unbounded length, it is intractable to store the entire data into main memory.

Finding frequent itemsets in data streams lends itself to many applications of Big Data. In many such applications, one is normally interested in the frequent itemsets in the recent period of time.

For example, in the telecommunication network fault analysis, it is important to know what are the frequent itemsets happening before the faults on the network during the recent period of time. The mining of frequent itemsets is also essential to a wide range of emerging applications, such as web log and click-stream mining, network traffic analysis, trend analysis, e-business and stock market analysis, sensor networks, etc. With the rapid emergence of these new application domains, it has become challenging to analyze data over fast-arriving and large data streams in order to capture interesting trends, patterns and exceptions.

In a time-varying data stream, the frequency of itemsets may change with time. This will make invalid some old frequent itemsets and also introduce new ones. When changes are detected (e.g., a large fraction of old frequent itemsets are found no longer valid during the test against new arriving data), the frequent itemsets need to be re-computed to reflect the features of new data and the result should be available as quickly as possible.

In a data stream application instead of a file of market-baskets, we have a stream of baskets from which we want to mine the frequent itemsets. A clear distinction between streams and files, when frequent itemsets are considered, is that there is no end to a stream, so eventually an itemset is going to exceed the support threshold as long as it appears repeatedly in the stream. As a result, for streams, we must think of the support threshold s as a fraction of the baskets in which an itemset must appear in order to be considered frequent. Even with this adjustment, we still have several options regarding the portion of the stream over which that fraction is measured.

In the following section, we shall give an overview of several methods that can be used to extract frequent itemsets from a stream.

8.6.1 Sampling Methods for Streams

Let us assume that stream elements are baskets of items. One simple approach to maintaining a current estimate of the frequent itemsets in a stream is to collect some number of baskets and store it as a file. Run any one of the frequent-itemset algorithms discussed earlier in this chapter, meanwhile ignoring the stream elements that keep arriving, or storing them as another file to be analyzed later. When the frequent-itemsets algorithm finishes, we have an estimate of the frequent itemsets in the stream.

We can use this collection of frequent itemsets for whatever application is at hand, but start running another iteration of the chosen frequent-itemset algorithm immediately. The following two methods can be used:

1. Use the file that was collected while the first iteration of the algorithm was running. At the same time, collect yet another file to be used at another iteration of the algorithm when this current iteration finishes.
2. Start collecting another file of baskets now, and run the algorithm when an adequate number of baskets has been collected.

We can continue to count the numbers of occurrences of each of these frequent itemsets, along with the total number of baskets seen in the stream, since the counting started. If any itemset is discovered to occur in a fraction of the baskets that is significantly below the threshold fraction s , then this set can be dropped from the collection of frequent itemsets. When computing the fraction, it is important to include the occurrences from the original file of baskets from which the frequent itemsets were derived. If not, we run the risk that we shall encounter a short period in which a truly frequent itemset does not appear sufficiently frequently and throw it out. We should also allow some way for new frequent itemsets to be added to the current collection.

1. Periodically gather a new segment of the baskets in the stream and use it as the data file for another iteration of the chosen frequent-itemsets algorithm. The new collection of frequent items is formed from the result of this iteration and the frequent itemsets from the previous collection that have survived the possibility of having been deleted for becoming infrequent.
2. Add some random itemsets to the current collection, and count their fraction of occurrences for a while, until one has a good idea of whether or not they are currently frequent.

Rather than choosing the new itemsets completely at random, one might focus on sets with items that appear in many itemsets already known to be frequent. For instance, a good choice can be to pick new itemsets from the negative border of the current set of frequent itemsets.

8.6.2 Frequent Itemsets in Decaying Windows

As discussed in Chapter 6, a decaying window on a stream is formed by picking a small constant c and giving the i^{th} element prior to the most recent element the weight $(1 - c)^i$, or approximately e^{-ci} . We can extend the algorithm discussed in Chapter 6 for computing the frequent items, by redefining the concept of the support threshold. There, we considered, for each item, a stream that had 1 if the item appeared at a certain stream element and 0 if not. We defined the “score” for that item to be the sum of the weights where its stream element was 1. We were constrained to record all items whose score was at least $1/2$. We cannot use a score threshold above 1, because we do not initiate a count for an item until the item appears in the stream, and the first time it appears, its score is only 1 [since 1, or $(1 - c)^0$, is the weight of the current item].

If we wish to adapt this method to streams of baskets, we consider two modifications. Stream elements are baskets rather than individual items, so many items may appear at a given stream element. Treat each of those items as if they were the “current” item and add 1 to their score after multiplying all current scores by $1 - c$. If some items in a basket have no current score, initialize the scores of those items to 1.

The second modification is trickier. We want to find all frequent itemsets, not just singleton itemsets. If we were to initialize a count for an itemset whenever we saw it, we would have too many counts. For example, one basket of 20 items has over a million subsets, and all of these would have to be initiated for one basket. On the other hand, as we mentioned, if we use a requirement

above 1 for initiating the scoring of an itemset, then we would never get any itemsets started, and the method would not work.

A way of dealing with this problem is to start scoring certain itemsets as soon as we see one instance, but be conservative about which itemsets we start. We may borrow from the Apriori trick and only start an itemset I if all its immediate proper subsets are already being scored. The consequence of this restriction is that if I is truly frequent, eventually we shall begin to count it, but we never start an itemset unless it would at least be a candidate in the sense used in the Apriori algorithm.

Example 15

Suppose I is a large itemset, but it appears in the stream periodically, once every $2/c$ baskets. Then its score, and that of its subsets, never falls below $e^{-1/2}$, which is greater than $1/2$. Thus, once a score is created for some subset of I , that subset will continue to be scored forever. The first time I appears, only its singleton subsets will have scores created for them. However, the next time I appears, each of its doubleton subsets will commence scoring, since each of the immediate subsets of those doubletons is already being scored. Likewise, the k th time I appears, its subsets of size $k - 1$ are all being scored, so we initiate scores for each of its subsets of size k . Eventually, we reach the size $|I|$, at which time we start scoring I itself.

Summary

- **Market-basket data model:** This model of data assumes that there are two kinds of entities: items and baskets. There is a many-many relationship between items and baskets. Typically, baskets are related to small sets of items, while items may be related to many baskets.
- **Frequent itemsets:** The support for a set of items is the number of baskets containing all those items. Itemsets with support that is at least some threshold are called frequent itemsets.
- **Association rules:** These are implications that if a basket contains a certain set of items I , then it is likely to contain another particular item j as well. The probability that j is also in a basket containing I is called the

confidence of the rule. The interest of the rule is the amount by which the confidence deviates from the fraction of all baskets that contain j .

- **The Pair-counting bottleneck:** To find frequent itemsets, we need to examine all baskets and count the number of occurrences of sets of a certain size. For typical data, with a goal of producing a small number of itemsets that are the most frequent of all, the part that often takes most of the main memory is the counting of pairs of items. Thus, methods for finding frequent itemsets typically concentrate on how to minimize the main memory needed to count pairs.
- **Triangular matrices and triples:** While one could use a two-dimensional array to count

pairs, doing so wastes half the space. Thus, we use the single dimension representation of the triangular matrix. If fewer than one-third of the possible pairs actually occur in baskets, then it is more space-efficient to store counts of pairs as triples (i, j, c) , where c is the count of the pair $\{i, j\}$ and $i < j$. An index structure such as a hash table allows us to find the triple for (i, j) efficiently.

- **Monotonicity of frequent itemsets:** An important property of itemsets is that if a set of items is frequent, then so are all its subsets. We exploit this property to eliminate the need to count certain itemsets by using its contrapositive: If an itemset is not frequent, then neither are its supersets.
- **The Apriori algorithm for pairs:** We can find all frequent pairs by making two passes over the baskets. On the first pass, we count the items themselves and then determine which items are frequent. On the second pass, we count only the pairs of items both of which are found frequent on the first pass. Monotonicity justifies our ignoring other pairs.
- **The PCY algorithm:** This algorithm improves on Apriori by creating a hash table on the first pass, using all main-memory space that is not needed to count the items. Pairs of items are hashed, and the hashtable buckets are used as integer counts of the number of times a pair has hashed to that bucket. Then, on the second pass, we only have to count pairs of frequent items that hashed to a frequent bucket (one whose count is at least the support threshold) on the first pass.
- **The multistage and multihash algorithm:** These are extensions to the PCY algorithm by inserting additional passes between the

first and second pass of the PCY algorithm to hash pairs to other, independent hash tables. Alternatively, we can modify the first pass of the PCY algorithm to divide available main memory into several hash tables. On the second pass, we only have to count a pair of frequent items if they hashed to frequent buckets in all hash tables.

- **Randomized algorithms:** Instead of making passes through all the data, we may choose a random sample of the baskets, small enough that it is possible to store both the sample and the needed counts of itemsets in the main memory. While this method uses at most one pass through the whole dataset, it is subject to false positives (itemsets that are frequent in the sample but not the whole) and false negatives (itemsets that are frequent in the whole but not the sample).
- **The SON algorithm:** This algorithm divides the entire file of baskets into segments small enough that all frequent itemsets for the segment can be found in main memory. Candidate itemsets are those found frequent for at least one segment. A second pass allows us to count all the candidates and find the exact collection of frequent itemsets. This algorithm is especially appropriate for a MapReduce setting making it ideal for Big Data.
- **Toivonen's algorithm:** This algorithm improves on the random sampling algorithm by avoiding both false positives and negatives. To achieve this, it searches for frequent itemsets in a sample, but with the threshold lowered so there is little chance of missing an itemset that is frequent in the whole. Next, we examine the entire file of baskets, counting not only the itemsets that are frequent in

the sample, but also the negative border. If no member of the negative border is found frequent in the whole, then the answer is exact. But if a member of the negative border is found frequent, then the whole process has to repeat with another sample.

- **Frequent itemsets in streams:** We present overview of a few techniques that can be used to count frequent items in a stream. We also present a few techniques that use the concept of decaying windows for finding more recent frequent itemsets.

Exercises

1. Imagine there are 100 baskets, numbered 1, 2, ..., 100, and 100 items, similarly numbered. Item i is in basket j if and only if i divides j evenly. For example, basket 24 is the set of items $\{1, 2, 3, 4, 6, 8, 12, 24\}$. Describe all the association rules that have 100% confidence.
2. Suppose we have transactions that satisfy the following assumptions:
 - The support threshold s is 10,000.
 - There are one million items, which are represented by the integers 0, 1, ..., 999999.
 - There are N frequent items, that is, items that occur 10,000 times or more.
 - There are one million pairs that occur 10,000 times or more.
 - There are $2M$ pairs that occur exactly once. M of these pairs consist of two frequent items, the other M each have at least one non-frequent item.
 - No other pairs occur at all.
 - Integers are always represented by 4 bytes.

Suppose we run the Apriori algorithm to find frequent pairs and can choose on the second pass between the triangular-matrix method for counting candidate pairs (a triangular array $\text{count}[i][j]$ that holds an integer count

for each pair of items (i, j) where $i < j$) and a hash table of item-item-count triples. In the first case neglect the space needed to translate between original item numbers and numbers for the frequent items, and in the second case neglect the space needed for the hash table. Assume that item numbers and counts are always 4-byte integers.

As a function of N and M , what is the minimum number of bytes of main memory needed to execute the Apriori algorithm on this data?

3. If we use a triangular matrix to count pairs, and n , the number of items, is 20, what pair's count is in a [100]?
4. Let there be I items in a market-basket dataset of B baskets. Suppose that every basket contains exactly K items. As a function of I , B , and K :
 - (a) How much space does the triangular-matrix method take to store the counts of all pairs of items, assuming four bytes per array element?
 - (b) What is the largest possible number of pairs with a non-zero count?
 - (c) Under what circumstances can we be certain that the triples method will use less space than the triangular array?

5. Imagine that there are 1100 items, of which 100 are “big” and 1000 are “little”. A basket is formed by adding each big item with probability 1/10, and each little item with probability 1/100. Assume the number of baskets is large enough that each item-set appears in a fraction of the baskets that equals its probability of being in any given basket. For example, every pair consisting of a big item and a little item appears in 1/1000 of the baskets. Let s be the support threshold, but expressed as a fraction of the total number of baskets rather than as an absolute number. Give, as a function of s ranging from 0 to 1, the number of frequent items on Pass 1 of the Apriori algorithm. Also, give the number of candidate pairs on the second pass.
6. Consider running the PCY algorithm on the data of Exercise 5, with 100,000 buckets on the first pass. Assume that the hash function used distributes the pairs to buckets in a conveniently random fashion. Specifically, the 499,500 little–little pairs are divided as evenly as possible (approximately 5 to a bucket). One of the 100,000 big–little pairs is in each bucket, and the 4950 big–big pairs each go into a different bucket.
- (a) As a function of s , the ratio of the support threshold to the total number of baskets (as in Exercise 5), how many frequent buckets are there on the first pass?
- (b) As a function of s , how many pairs must be counted on the second pass?
7. Here is a collection of 12 baskets. Each contains three of the six items 1 through 6. {1,2,3} {2,3,4} {3,4,5} {4,5,6} {1,3,5} {2,4,6} {1,3,4} {2,4,5} {3,5,6} {1,2,4} {2,3,5} {3,4,6}. Suppose the support threshold is 3. On the first pass of the PCY algorithm we use a hash table with 10 buckets, and the set $\{i, j\}$ is hashed to bucket $i \times j \bmod 10$.
- (a) By any method, compute the support for each item and each pair of items.
- (b) Which pairs hash to which buckets?
- (c) Which buckets are frequent?
- (d) Which pairs are counted on the second pass of the PCY algorithm?
8. Suppose we run the Multistage algorithm on the data of Exercise 7, with the same support threshold of 3. The first pass is the same as in that exercise, and for the second pass, we hash pairs to nine buckets, using the hash function that hashes $\{i, j\}$ to bucket $i + j \bmod 8$. Determine the counts of the buckets on the second pass. Does the second pass reduce the set of candidate pairs?
9. Suppose we run the Multihash algorithm on the data of Exercise 7. We shall use two hash tables with five buckets each. For one, the set $\{i, j\}$ is hashed to bucket $2i + 3j + 4 \bmod 5$, and for the other, the set is hashed to $i + 4j \bmod 5$. Since these hash functions are not symmetric in i and j , order the items so that $i < j$ when evaluating each hash function. Determine the counts of each of the 10 buckets. How large does the support threshold have to be for the Multistage algorithm to eliminate more pairs than the PCY algorithm would, using the hash table and function described in Exercise 7?
10. During a run of Toivonen’s algorithm with set of items {A,B,C,D,E,F,G,H} a sample is found to have the following maximal frequent itemsets: {A,B}, {A,C}, {A,D}, {B,C}, {E}, {F}. Compute the negative border. Then, identify in the list below the set that is NOT in the negative border.

Programming Assignments

1. Implement the following algorithms using MapReduce on any standard dataset.
 - (a) Modified Apriori for counting pairs
 - (b) PCY Algorithm
 - (c) Multihash Algorithm
 - (d) SON Algorithm
 - (e) Toivonen Algorithm

References

1. R. Agrawal, T. Imielinski, A. Swami (1993). Mining Associations between Sets of Items in Massive Databases. *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 207–216.
2. R. Agrawal, R. Srikant (1994). Fast Algorithms for Mining Association Rules. *Intl. Conf. on Very Large Databases*, pp. 487–499.
3. M. Fang, N. Shivakumar, H. Garcia-Molina *et al.* (1998). Computing Iceberg Queries Efficiently. *Intl. Conf. on Very Large Databases*, pp. 299–310.
4. J.S. Park, M.-S. Chen, P.S. Yu (1995). An Effective Hash-Based Algorithm for Mining Association Rules. *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 175–186.
5. Savasere, E. Omiecinski, S.B. Navathe (1995). An Efficient Algorithm for Mining Association Rules in Large Databases. *Intl. Conf. on Very Large Databases*, pp. 432–444.
6. H. Toivonen (1996). Sampling Large Databases for Association Rules. *Intl. Conf. on Very Large Databases*, pp. 134–145.

9

Clustering Approaches

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Understand traditional clustering algorithms like partition-based and hierarchical algorithms.
- Identify the challenges posed by big data to traditional clustering techniques.
- Learn and understand the CURE Algorithm for big data.
- Understand the concept of stream clustering and how to answer stream queries.

9.1

Introduction

The problem of data clustering has been widely explored in the data mining and machine learning literature because of its numerous applications to summarization, learning, segmentation, and target marketing. In many large datasets that are unlabeled, clustering offers a concise model of the data that can be interpreted either as a summary or a generative model. The goal of data clustering, also known as “Cluster Analysis”, is to discover the *natural* grouping(s) of a set of patterns, points, or objects.

The basic definition of the problem of clustering may be stated as follows: *Given a set of data points, partition them into a set of groups which are as similar as possible.*

An operational definition of clustering can be stated as follows: “Given a *representation* of n objects, find K groups based on a measure of *similarity* such that objects within the same group are *alike* but the objects in different groups are not alike”.

Thus, the above definition indicates that before we can talk about clustering we must first define a measure that can be used to indicate similarity between objects. Similar objects will have higher similarity values as compared to dissimilar objects.

There are a variety of algorithms that can be used for clustering. All these algorithms differ in the way they define clusters and on the basis of how they can efficiently identify them. In general, when we perform clustering of data the final output would consist of sets of dense data points where there is minimum distance among points belonging to the same group and a larger separation among different

groups. The choice of which clustering algorithm is appropriate, which similarity/distance measures to use, what are the number of clusters to be obtained, what are the distance thresholds within a cluster and other such parameters depend on the dataset for which clustering is desired and further on the intended application of the clustering results.

In this chapter, we discuss a series of different methods for performing clustering analysis on different types of data. Initially a short overview to cluster analysis is provided. The chapter then focuses on the challenge of clustering high-dimensional data.

9.2

Overview of Clustering Techniques

In this section, we will review the basic concepts of clustering and introduce the different terminologies used. We will give an overview of traditional clustering techniques and discuss the so-called “Curse of Dimensionality” that dictates the design of different set of algorithms for high-dimensional data.

9.2.1 Basic Concepts

Clustering datasets consist of objects (measurements, events) that are normally represented as points (vectors) in a multi-dimensional space. Each dimension represents a distinct attribute (variable, measurement) describing the object. For simplicity, it is usually assumed that values are present for all attributes. The set of objects form an m by n matrix, where there are m rows, one for each object, and n columns, one for each attribute.

Since clustering is the grouping of similar instances/objects, some measures that can determine whether two objects are similar or dissimilar are required. The two main types of measures normally used to estimate this relation are distance measures and similarity measures. Many clustering methods use distance measures to determine the similarity or dissimilarity between any pair of objects. It is useful to denote the distance between two instances x_i and x_j as $d(x_i, x_j)$. We have seen several such distance measures in Chapter 5.

We can recall that a valid distance measure should be symmetric and attain zero value when the two vectors are identical. The distance measure is called a metric distance measure if it also satisfies the following properties:

1. Triangle inequality $d(x_i, x_k) \leq d(x_i, x_j) + d(x_j, x_k) \forall x_i, x_j, x_k \in S$
2. $d(x_i, x_k) = 0 \Rightarrow x_i = x_k \forall x_i, x_k \in S$

However, current clustering situations require more sophisticated measures. They may involve Euclidean spaces of very high dimension or spaces that are not Euclidean at all. In such cases, objects

are sometimes represented by more complicated data structures than the vectors of attributes. Good examples include text documents, images, or graphs. Determining the similarity (or differences) of two objects in such a situation is more complicated, but if a reasonable similarity (dissimilarity) measure exists, then a clustering analysis can still be performed. Such measures which also were discussed in Chapter 5 include the Jaccard distance, Cosine distance, Hamming distance, and Edit distance.

Example 1**Clustering Problem: Books****1. Intuitively: Books are divided into categories, and customers prefer a few categories**

Represent a book by a set of customers who bought it:

Similar books have similar sets of customers, and vice-versa

2. Space of all books:

Think of a space with one dimension for each customer

Values in a dimension may be 0 or 1 only

A book is a point in this space (x_1, x_2, \dots, x_k) , where $x_i = 1$ iff the i^{th} customer bought the CD

For an online bookseller Amazon, this dimension is tens of millions

3. Task: Find clusters of similar CDs**Example 2****Clustering Problem: Documents****Finding topics:**

1. Represent a document by a vector (x_1, x_2, \dots, x_k) , where $x_i = 1$ iff the i^{th} word (in some order) appears in the document
2. Documents with similar sets of words may be about the same topic
3. We have a choice when we think of documents as sets of words:
 - *Sets as vectors:* Measure similarity by the cosine distance
 - *Sets as sets:* Measure similarity by the Jaccard distance
 - *Sets as points:* Measure similarity by Euclidean distance

9.2.2 Clustering Applications

Some common application domains in which the clustering problem arises are as follows:

1. **Intermediate step for other fundamental data mining problems:** Since a clustering can be considered a form of data summarization, it often serves as a key intermediate step for many fundamental data mining problems, such as classification or outlier analysis. A compact summary of the data is often useful for different kinds of application-specific insights.
2. **Collaborative filtering:** In collaborative filtering methods of clustering provides a summarization of like-minded users. The ratings provided by the different users for each other are used in order to perform the collaborative filtering. This can be used to provide recommendations in a variety of applications.
3. **Customer segmentation:** This application is quite similar to collaborative filtering, since it creates groups of similar customers in the data. The major difference from collaborative filtering is that instead of using rating information, arbitrary attributes about the objects may be used for clustering purposes.
4. **Data summarization:** Many clustering methods are closely related to dimensionality reduction methods. Such methods can be considered as a form of data summarization. Data summarization can be helpful in creating compact data representations that are easier to process and interpret in a wide variety of applications.
5. **Dynamic trend detection:** Many forms of dynamic and streaming algorithms can be used to perform trend detection in a wide variety of social networking applications. In such applications, the data is dynamically clustered in a streaming fashion and can be used in order to determine the important patterns of changes. Examples of such streaming data could be multi-dimensional data, text streams, streaming time-series data, and trajectory data. Key trends and events in the data can be discovered with the use of clustering methods.
6. **Multimedia data analysis:** A variety of different kinds of documents, such as images, audio, or video, fall in the general category of multimedia data. The determination of similar segments has numerous applications, such as the determination of similar snippets of music or similar photographs. In many cases, the data may be multi-modal and may contain different types. In such cases, the problem becomes even more challenging.
7. **Biological data analysis:** Biological data has become pervasive in the last few years, because of the success of the human genome effort and the increasing ability to collect different kinds of gene expression data. Biological data is usually structured either as sequences or as networks. Clustering algorithms provide good ideas of the key trends in the data, as well as the unusual sequences.
8. **Social network analysis:** In these applications, the structure of a social network is used in order to determine the important communities in the underlying network. Community detection has important applications in social network analysis, because it provides an important understanding of the community structure in the network. Clustering also has applications to social network summarization, which is useful in a number of applications.

The above-mentioned list of applications represents a good cross-section of the wide diversity of problems that can be addressed with the clustering algorithms.

9.2.3 Clustering Strategies

Distinction in clustering approaches, that is between hierarchical and partitional approaches, is highlighted below:

1. **Hierarchical techniques** produce a nested arrangement of partitions, with a single cluster at the top consisting of all data points and singleton clusters of individual points at the bottom. Each intermediate level can be viewed as combining (splitting) two clusters from the next lower (next higher) level. Hierarchical clustering techniques which start with one cluster of all the points and then keep progressively splitting the clusters till singleton clusters are reached are called “divisive” clustering. On the other hand, approaches that start with singleton clusters and go on merging close clusters at every step until they reach one cluster consisting of the entire dataset are called “agglomerative” methods. While most hierarchical algorithms just join two clusters or split a cluster into two sub-clusters at every step, there exist hierarchical algorithms that can join more than two clusters in one step or split a cluster into more than two sub-clusters.
2. **Partitional techniques** create one-level (un-nested) partitioning of the data points. If K is the desired number of clusters, then partitional approaches typically find all K clusters in one step. The important issue is we need to have predefined value of K , the number of clusters we propose to identify in the dataset.

Of course, a hierarchical approach can be used to generate a flat partition of K clusters, and likewise, the repeated application of a partitional scheme can provide a hierarchical clustering.

There are also other important distinctions between clustering algorithms as discussed below:

1. Does a clustering algorithm use all attributes simultaneously (polythetic) or use only one attribute at a time (monothetic) to compute the distance?
2. Does a clustering technique use one object at a time (incremental) or does the algorithm consider all objects at once (non-incremental)?
3. Does the clustering method allow a point to belong to multiple clusters (overlapping) or does it insist that each object can belong to one cluster only (non-overlapping)? Overlapping clusters are not to be confused with fuzzy clusters, as in fuzzy clusters objects actually belong to multiple classes with varying levels of membership.

Algorithms for clustering big data can also be distinguished by

1. Whether the dataset is treated as a Euclidean space, and if the algorithm can work for any arbitrary distance measure. In a Euclidean space where data is represented as a vector of real numbers,

the notion of a Centroid which can be used to summarize a collection of data points is very natural - the mean value of the points. In a non-Euclidean space, for example, images or documents where data is a set of words or a group of pixels, there is no notion of a Centroid, and we are forced to develop another way to summarize clusters.

2. Whether the algorithm is based on the assumption that data will fit in main memory, or whether data must reside in secondary memory, primarily. Algorithms for large amounts of data often must take shortcuts, since it is infeasible to look at all pairs of points. It is also necessary to summarize the clusters in main memory itself as is common with most big data algorithms.

9.2.4 Curse of Dimensionality

It was Richard Bellman who apparently originated the phrase “the curse of dimensionality” in a book on control theory. In current times any problem in analyzing data that occurs due to a very large number of attributes that have to be considered is attributed to the, “curse of dimensionality”.

Why should large number of dimensions pose problems? Firstly a fixed number of data points become increasingly “sparse” as the dimensionality increase. To visualize this, consider 100 points distributed with a uniform random distribution in the interval $[-0, 1]$. If this interval is broken into 10 cells. Then it is highly likely that all cells will contain some points. But now if we keep the number of points the same, but (project) distribute the points over the unit square that is 2-D and fix the unit of discretization to be 0.1 for each dimension, then we have 100 2-D cells, and it is quite likely that some cells will be empty. For 100 points and three dimensions, most of the 1000 cells will be empty since there are far more points than the cells. Thus our data is “lost in space” as we go to higher dimensions.

We face one more problem with high dimensions in the clustering scenario. This concerns the effect of increasing dimensionality on distance or similarity. The usefulness of a Clustering technique is critically dependent on the measure of distance or similarity used. Further for clustering to be meaningful, we require that objects within clusters should be closer to each other than to objects in other clusters. One way of analyzing the existence of clusters in a dataset is to plot the histogram of the pairwise distances of all points in a dataset (or of a sample of points in a large dataset). If clusters exist in the data, then they will show up in the graph as two peaks; one representing the distance between the points in clusters, and the other representing the average distance between the points. If only one peak is present or if the two peaks are close to each other, then clustering via distance-based approaches will likely be difficult.

Studies have also shown that for certain data distributions, the relative difference of the distances of the closest and farthest data points of an independently selected point goes to 0 as the dimensionality increases, that is,

$$\lim \frac{(\max \text{ dist} - \min \text{ dist})}{\min \text{ dist}} = 0 \text{ as } d \rightarrow \infty$$

Thus, it is often said, “in high-dimensional spaces, distances between points become relatively uniform”. In such cases, the notion of the nearest neighbor of a point is meaningless. To understand this in a more geometrical way, consider a hyper-sphere whose center is the selected point and whose radius is the distance to the nearest data point. Then, if the relative difference between the distance to nearest and farthest neighbors is small, expanding the radius of the sphere “slightly” will include many more points.

9.3 Hierarchical Clustering

Hierarchical clustering builds a cluster hierarchy or, in other words, a tree of clusters, also known as a dendrogram. Every cluster node contains child clusters; sibling clusters partition the points covered by their common parent. Such an approach allows exploring data on different levels of granularity. Hierarchical clustering methods are categorized into agglomerative (bottom-up) and divisive (top-down).

The advantages of hierarchical clustering include: embedded flexibility regarding the level of granularity; ease of handling of any forms of similarity or distance; can be extended to be applicable with any attribute types. But these algorithms also suffer from the following drawbacks: ambiguity in termination criteria; the fact that most hierarchical algorithms cannot disturb an earlier intermediate cluster even for improvement of cluster quality.

Hierarchical clustering can also be differentiated based on whether we are dealing with Euclidean or non-Euclidean space.

9.3.1 Hierarchical Clustering in Euclidean Space

These methods construct the clusters by recursively partitioning the instances in either a top-down or bottom-up fashion. These methods can be subdivided as following:

- 1. Agglomerative hierarchical clustering:** Each object initially represents a cluster of its own. Then clusters are successively merged until the desired clustering is obtained.
- 2. Divisive hierarchical clustering:** All objects initially belong to one cluster. Then the cluster is divided into sub-clusters, which are successively divided into their own sub-clusters. This process continues until the desired cluster structure is obtained.

The result of the hierarchical methods is a dendrogram, representing the nested grouping of objects and similarity levels at which groupings change. A clustering of the data objects is obtained by cutting the dendrogram at the desired similarity level. Figure 9.1 shows a simple example of hierarchical clustering. The merging or division of clusters is to be performed according to some similarity measure, chosen so as to optimize some error criterion (such as a sum of squares).

The hierarchical clustering methods can be further divided according to the manner in which inter-cluster distances for merging are calculated.

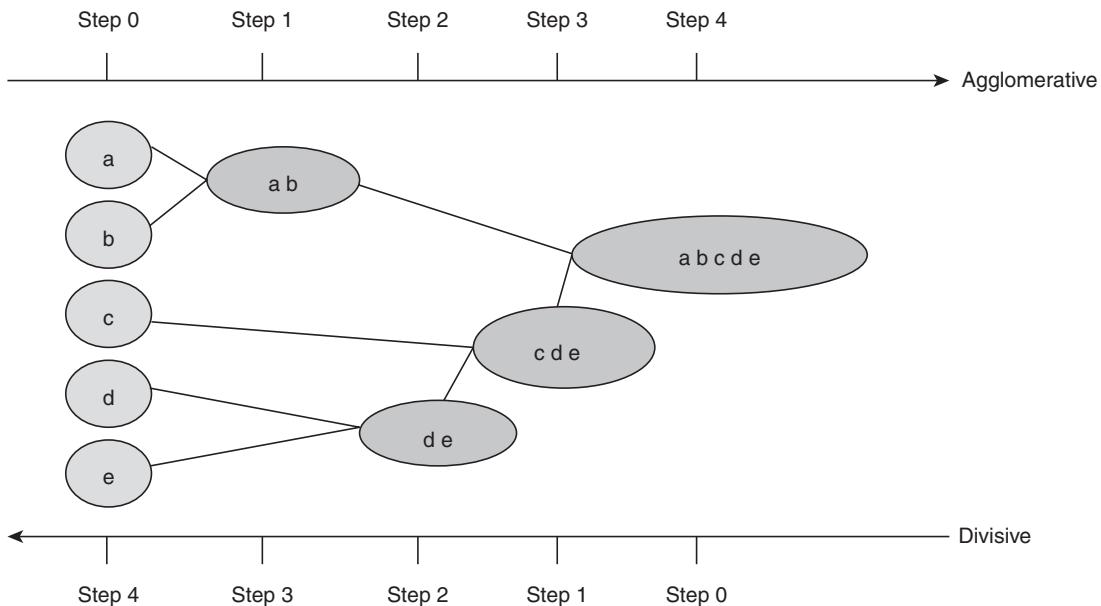


Figure 9.1 Hierarchical clustering.

1. **Single-link clustering:** Here the distance between the two clusters is taken as the shortest distance from any member of one cluster to any member of the other cluster. If the data consist of similarities, the similarity between a pair of clusters is considered to be equal to the greatest similarity from any member of one cluster to any member of the other cluster.
2. **Complete-link clustering:** Here the distance between the two clusters is the longest distance from any member of one cluster to any member of the other cluster.
3. **Average-link clustering:** Here the distance between two clusters is the average of all distances computed between every pair of two points one from each cluster.
4. **Centroid link clustering:** Here the distance between the clusters is computed as the distance between the two mean data points (average point) of the clusters. This average point of a cluster is called its **Centroid**. At each step of the clustering process we combine the two clusters that have the smallest Centroid distance. The notion of a Centroid is relevant for Euclidean space only, since all the data points have attributes with real values. Figure 9.2 shows this process.

Another decision point for hierarchical clustering would be the stopping criteria. Some choices are: stop merging (or splitting) the clusters when

1. Some pre-determined number of clusters is reached. We know beforehand that we need to find a fixed number of clusters “ K ” and we stop when we have K clusters.

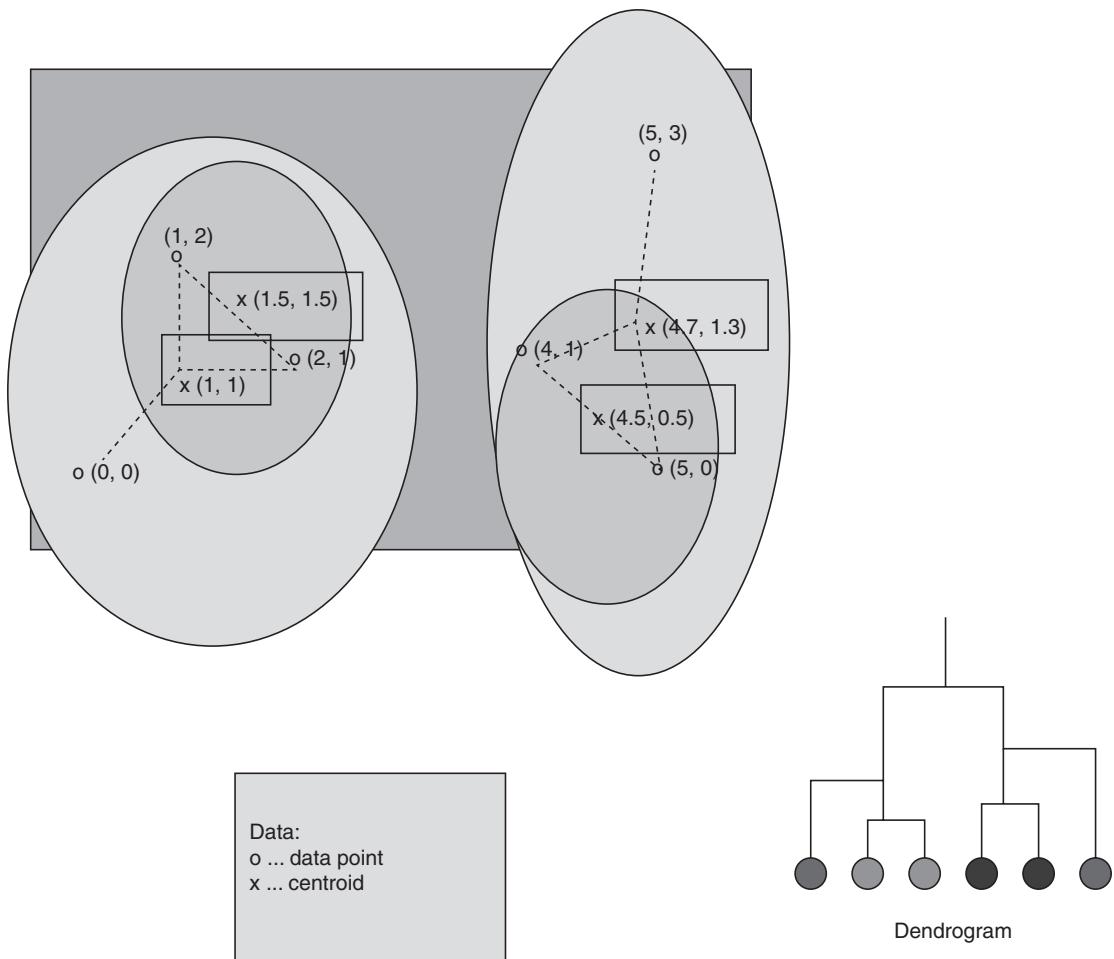


Figure 9.2 Illustrating hierarchical clustering.

2. If across a series of merges or splits, very little change occurs to the clustering, it means clustering has reached some stable structure.
3. If the maximum distance between any two points in a cluster becomes greater than a pre-specified value or threshold we can stop further steps.
4. Combination of the above conditions.

The main disadvantages of the hierarchical methods are as follows:

1. Inability to scale well – The time complexity of hierarchical algorithms is at least $O(m^2)$ (where m is the total number of instances), which is non-linear with the number of objects.

Clustering a large number of objects using a hierarchical algorithm is also characterized by huge I/O costs.

2. Hierarchical methods can never undo what was done previously. Namely there is no back-tracking capability.

9.3.2 Hierarchical Clustering in Non-Euclidean Space

When the space is non-Euclidean, as is the case when data points are documents or images etc., we need to use some distance measure, such as Jaccard, cosine, or edit distance. We have seen in Chapter 5 how to compute such distances between two such data points. We need some mechanism for finding inter-cluster distance using Centroids. A problem arises when we need to represent a cluster, because we cannot replace a collection of points by their Centroid.

Example 3

Suppose we are using Jaccard distances and at some intermediate stage we want to merge two documents with Jaccard distance within a threshold value. However, we cannot find a document that represents their average, which could be used as its Centroid. Given that we cannot perform the average operation on points in a cluster when the space is non-Euclidean, our only choice is to pick one of the points of the cluster itself as a representative or a prototype of the cluster. Ideally, this point is similar to most points of the cluster, so it in some sense the “center” point.

This representative point is called the “Clustroid” of the cluster. We can select the Clustroid in various ways. Common choices include selecting as the Clustroid the point that minimizes:

1. The sum of the distances to the other points in the cluster.
2. The maximum distance to another point in the cluster.
3. The sum of the squares of the distances to the other points in the cluster.

9.4 Partitioning Methods

Partition-based clustering techniques create a flat single-level partitioning of the data points. If K is the desired number of clusters we want to discover in the data set, partition-based approaches find all K clusters in one step. Partitioning methods start with a random initial partition and keep reallocating the data points to different clusters in an iterative manner till the final partition is attained. These methods typically require that the number of clusters K will be predefined by the user. To achieve optimal clusters in partitioned-based clustering, a brute force approach will need to check all possible partitions of

the data set and compare their clustering characteristics to come up with the ideal clustering. Because this is not computationally feasible, certain greedy heuristics are used and an iterative optimization algorithm is used.

The simplest and most popular algorithm in this class is the K -means algorithm. But its memory requirements dictate they can only be used on small datasets. For big datasets we discuss a variant of K -means called the BFR algorithm.

9.4.1 K-Means Algorithm

The K -means algorithm discovers K (non-overlapping) clusters by finding K centroids (“central” points) and then assigning each point to the cluster associated with its nearest centroid. A cluster centroid is typically the mean or median of the points in its cluster and “nearness” is defined by a distance or similarity function. In the ideal case, the Centroids are chosen to minimize the total “error”, where the error for each point is given by a function that measures the dispersion of a point from its cluster Centroid, for example, the squared distance.

The algorithm starts with an initial set of cluster Centroids chosen at random or according to some heuristic procedure. In each iteration, each data point is assigned to its nearest cluster Centroid. Nearness is measured using the Euclidean distance measure. Then the cluster Centroids are re-computed. The Centroid of each cluster is calculated as the mean value of all the data points that are assigned to that cluster.

Several termination conditions are possible. For example, the search may stop when the error that is computed at every iteration does not reduce because of reassignment of the Centroids. This indicates that the present partition is locally optimal. Other stopping criteria can be used also such as stopping the algorithms after a pre-defined number of iterations. The procedure for the K -means algorithm is depicted in Fig. 9.3.

Algorithm

Input: S (data points), K (number of clusters)

Output: K clusters

1. Choose initial K cluster Centroids randomly.
 2. **while** termination condition is not satisfied **do**
 - (a) Assign data points to the closest Centroid.
 - (b) Recompute Centroids based on current cluster points.
- end while**

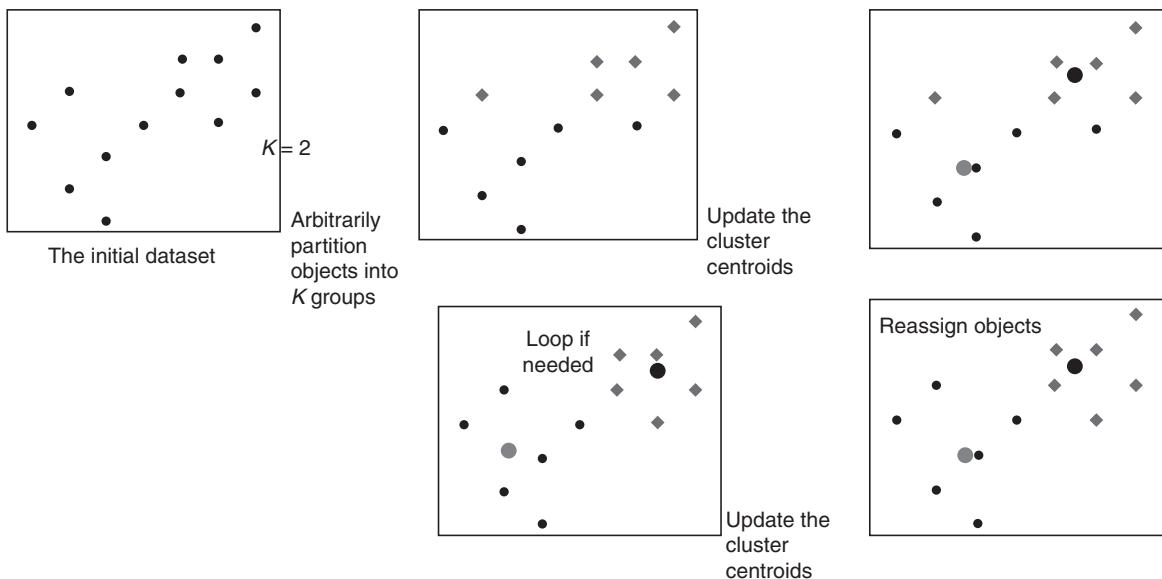


Figure 9.3 Illustrating K -means.

The K -means algorithm is a gradient-decent procedure which begins with an initial set of K cluster-centers and iteratively updates it so as to decrease the error function. The complexity of T iterations of the K -means algorithm performed on a sample size of m instances, each characterized by N attributes, is $O(T * K * m * N)$. This linear complexity is one of the reasons for the popularity of the K -means algorithms. Even if the number of data points is sufficiently large, this algorithm is computationally attractive. Other reasons for the algorithm's popularity are its ease of interpretation, simplicity of implementation, speed of convergence and adaptability to sparse data.

The major disadvantage of the K -means algorithm involves the selection of the initial partition or the initial Centroids. The algorithm is very sensitive to this selection, which may make the difference between the algorithm converging at global or local minimum.

Further, the K -means algorithm works well only on datasets having circular clusters. In addition, this algorithm is sensitive to noisy data and outliers; a single outlier can increase the squared error dramatically. It is applicable only when mean is defined (namely, for numeric attributes) and it requires the number of clusters in advance, which is not trivial when no prior knowledge is available.

Since the K -means algorithm requires Euclidean space that is often limited to the numeric attributes to find the Centroid, the K -prototypes algorithm was proposed. This algorithm is similar to the K -means algorithm but can process attributes that are non-numeric too. Instead of using the Centroid it uses a prototype which captures the essence of the cluster to which it belongs. The algorithm then clusters objects with numeric and categorical attributes in a way similar to the K -means algorithm. The similarity measure on numeric attributes is the square Euclidean distance; the similarity measure on the categorical attributes is the number of mismatches between objects and the cluster prototypes.

One such algorithm is the K -medoids or partition around medoids (PAM). Each cluster is represented by the central object in the cluster, rather than by the mean that may not even belong to the cluster. The K -medoids method is more robust than the K -means algorithm in the presence of noise and outliers because a medoid is less influenced by outliers or other extreme values than a mean. However, its processing is more costly than the K -means method. Both methods require the user to specify K , the number of clusters.

9.4.2 K-Means For Large-Scale Data

We shall now examine an extension of K -means that is designed to deal with datasets so large that they cannot fit in main memory. This algorithm does not output the clustering per se but just determines the cluster Centroids. If we require the final clustering of points, another pass is made through the dataset, assigning each point to its nearest Centroid which is then returned as the result.

This algorithm, called the BFR algorithm, for its authors (Bradley, Fayyad and Reina), assumes the data points are from an n -dimensional Euclidean space. Therefore, we can use the Centroids to represent clusters, as they are being formed. The BFR algorithm also assumes that the quality of a cluster can be measured by the variance of the points within a cluster; the variance of a cluster is the average of the square of the distance of a point in the cluster from the Centroid of the cluster.

This algorithm for reasons of optimality does not record the Centroid and variance, but rather stores the following $(2n + 1)$ summary statistics for each cluster:

1. N , the number of points in the cluster.
2. For each dimension i , the sum of the i^{th} coordinates of the points in the cluster, denoted SUM_i .
3. For each dimension i , the sum of the squares of the i^{th} coordinates of the points in the cluster, denoted as SUMSQ_i .

The reasoning behind using these parameters comes from the fact that these parameters are easy to compute when we merge two clusters. We need to just add the corresponding values from the two clusters. Similarly we can compute the Centroid and variance also very easily from these values as:

1. The i^{th} coordinate of the Centroid is $\frac{\text{SUM}_i}{N}$.
2. The variance in the i^{th} dimension is $\frac{\text{SUMSQ}_i}{N} - \left(\frac{\text{SUM}_i}{N}\right)^2$.
3. The standard deviation in the i^{th} dimension is the square root of the variance in that dimension.

Initially, the BFR algorithm selects k points, either randomly or using some preprocessing methods to make better choices. In the next step the data file containing the points of the dataset are in chunks. These chunks could be from data stored in a distributed file system or there may be one monolithic

huge file which is then divided into chunks of the appropriate size. Each chunk consists of just so many points as can be processed in the main memory. Further some amount of main memory is also required to store the summaries of the k clusters and other data, so the entire memory is not available to store a chunk.

The data stored in the main-memory other than the chunk from the input consists of three types of objects:

1. **The discard set:** The points already assigned to a cluster. These points do not appear in main memory. They are represented only by the summary statistics for their cluster.
2. **The compressed set:** There are several groups of points that are sufficiently close to each other for us to believe they belong in the same cluster, but at present they are not close to any current Centroid. In this case we cannot assign a cluster to these points as we cannot ascertain to which cluster they belong. Each such group is represented by its summary statistics, just like the clusters are, and the points themselves do not appear in main memory.
3. **The retained set:** These points are not close to any other points; they are “outliers.” They will eventually be assigned to the nearest cluster, but for the moment we have to retain each such point in main memory.

These sets will change as we bring in successive chunks of data into the main memory. Figure 9.4 indicates the state of the data after a few chunks of data have been processed by the BFR algorithm.

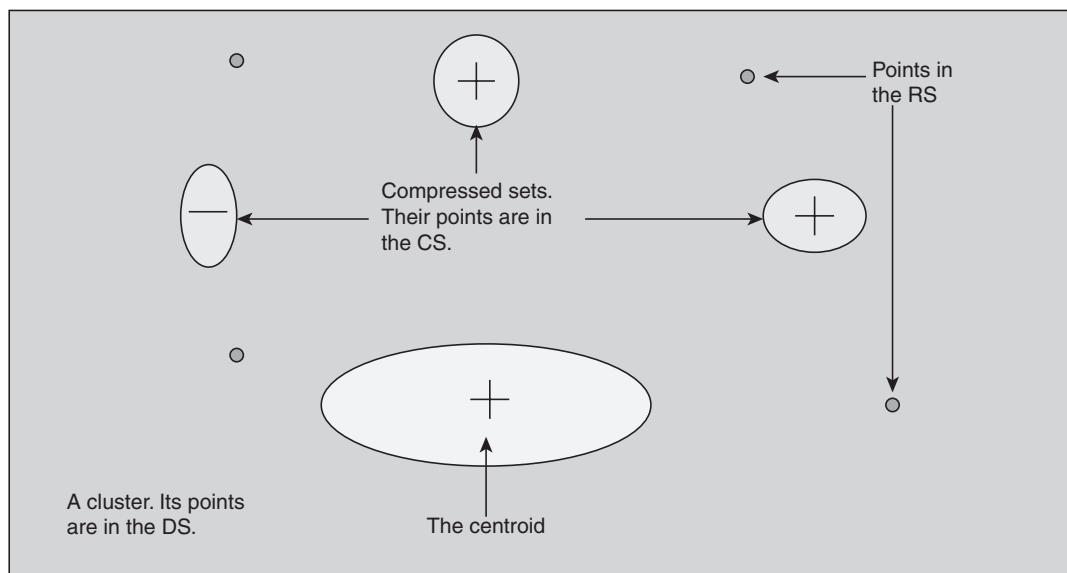


Figure 9.4 The three sets of points identified by BFR.

9.4.2.1 Processing a Memory Load of Points

We shall now describe how one chunk of data points are processed. We assume that the main memory currently contains the summary statistics for the K clusters and also for zero or more groups of points that are in the compressed set. The main memory also holds the current set of points in the retained set. We then perform the following steps:

1. For all points (x_1, x_2, \dots, x_n) that are “sufficiently close” (based on distance threshold) to the Centroid of a cluster, add the point to that cluster. The point then is added to the discard set. We add 1 to the value N in the summary statistics for that cluster indicating that this cluster has grown by one point. We also add X_j to SUM_i and add X_j^2 to SUMSQ_i for that cluster.
2. If this is the last chunk of data, merge each group from the compressed set and each point of the retained set into its nearest cluster. We have seen earlier that it is very simple and easy to merge clusters and groups using their summary statistics. Just add the counts N , and add corresponding components of the SUM and SUMSQ vectors. The algorithm ends at this point.
3. Otherwise (this was not the last chunk), use any main-memory clustering algorithm to cluster the remaining points from this chunk, along with all points in the current retained set. Set a threshold on the distance values that can occur in the cluster, so we do not merge points unless they are reasonably close.
4. Those points that remain isolated as clusters of size 1 (i.e., they are not near any other point) become the new retained set. Clusters of more than one point become groups in the compressed set and are replaced by their summary statistics.
5. Further we can consider merging groups in the compressed set. Use some threshold to decide whether groups are close enough; the following section outlines a method to do this. If they can be merged, then it is easy to combine their summary statistics, as in (2) above.

9.4.2.2 How to Decide Whether a Point is near a Cluster or Not?

Intuitively, each cluster has a threshold limit in each dimension which indicates the maximum allowed difference between values in that dimension. Since we have only the summary statistics to work with, the appropriate statistic is the standard deviation in that dimension. Recall that we can compute the standard deviations from the summary statistics; the standard deviation is the square root of the variance. We want to include a point if its distance from the cluster Centroid is not too many standard deviations in any dimension.

Thus, the first thing to do with a point p that we are considering for inclusion in a cluster is to normalize p relative to the Centroid and the standard deviations of the cluster say p' . The normalized distance of p from the centroid is the absolute distance of p' from the origin. This distance is sometimes called the Mahalanobis distance.

Example 4

Suppose p is the point $(5,10,15)$, and we are considering whether to include p in a cluster with centroid $(10,20,5)$. Also, let the standard deviation of the cluster in the three dimensions be 1, 2, and 10, respectively. Then the Mahalanobis distance of p is

$$\text{SQRT} \left\{ \left(\frac{(5-10)}{1} \right)^2 + \left(\frac{(10-20)}{2} \right)^2 + \left(\frac{(15-5)}{10} \right)^2 \right\} = 7.14$$

Having computed the Mahalanobis distance of point p , we can apply a threshold to decide whether or not to include p in the cluster. For instance, suppose we use 3 as the threshold; that is, we shall include the point if and only if its Mahalanobis distance from the centroid is not greater than 3. If values are normally distributed, then very few of these values will be more than 3 standard deviations from the mean (approximately one in a million will be that far from the mean). Thus, we would only reject one in a million points that belong in the cluster. There is a good chance that, at the end, the rejected points would wind up in the cluster anyway, since there may be no closer cluster.

We also need to decide whether to merge groups in the compressed set. We can make a decision using only the summary statistics for the two groups. Choose an upper bound on the sum of the variances of the combined group in each dimension. Recall that we compute the summary statistics for the combined group by just adding corresponding components, and compute the variance in each dimension using the formula discussed in the beginning of this section. Another way would be, put an upper limit on the diameter in any dimension. Since we do not know the locations of the points exactly, we cannot compute the exact diameter. However, we could estimate the diameter in the i^{th} dimension as the distance between the Centroids of the two groups in dimension i plus the standard deviation of each group in dimension i . This approach also limits the size of the region of space occupied by a group.

9.5 The CURE Algorithm

This section describes another large-scale-clustering algorithm which uses a combination of partition-based and hierarchical algorithms. This algorithm, called Clustering Using Representatives (CURE), assumes a Euclidean space. However, it does not assume anything about the shape of clusters; they need not be normally distributed, and can even have strange curves, S-shapes, or even rings. Instead of representing the clusters by their Centroid, it uses a collection of representative points, as is implied by the name of the algorithm.

9.5.1 Overview of the Algorithm

One way to manage clustering in large datasets would be to use one single data point to represent a cluster. Conceptually, we implicitly assume that each cluster has a spherical shape and thus its radius point is representative of the entire cluster. This is not true for many real-life applications where the clusters can exist in many other complicated shapes. At the other extreme, we can keep all the data points within each cluster, which is expensive both computation wise well as space wise, especially for large datasets. To address this issue, CURE proposes to use a set of well-scattered data points to represent a cluster.

CURE is essentially a hierarchical clustering algorithm. It starts by treating each data point as a single cluster and then recursively merges two existing clusters into one until we have only k clusters. In order to decide which two clusters to be merged at each iteration, it computes the minimum distance between all the possible pairs of the representative points from the two clusters. CURE uses two major data structures to perform this operation efficiently. First, it uses a heap to track the distance of each existing cluster to its closest cluster. Additionally, it also uses a k - d tree to store all the representative points for each cluster.

In order to speed up the computation, CURE first draws a sample of the input dataset and then runs the above procedure on the sampled data. When the original dataset is large, the different clusters might overlap each other, which in turn will require a large sample size. To overcome this problem, CURE uses partitions to speed up. It first partitions the n' sampled data points into p partitions. Within each partition, it then runs a *partial* hierarchical clustering algorithm until either a pre-defined number of clusters is reached or the distance between the two clusters to be merged exceeds some threshold. After that, it runs another clustering pass on all the partial clusters from all the p partitions (“global clustering”). Finally, each non-sampled data point is assigned to a cluster based on its distance to the representative point (“labeling”).

9.5.2 CURE Implementation

9.5.2.1 Initialization Step

Start with an initial sample of the data and cluster it using any traditional main memory clustering technique. Although any clustering method can be used, since CURE is designed more to handle oddly shaped clusters, hierarchical methods are the better choices. Agglomerative hierarchical methods work the best with CURE.

Select a small set of points from each cluster to be representative points. Choose these representatives so that they are wide apart, that is, as far from each other as possible.

Move each of the representative points by some fixed fraction of the distance between its location and the Centroid of its cluster. In effect we are shrinking these representative points towards the Centroid of the cluster by a fraction. This fraction could be about 20% or 30% of the original distance. (Note that this step requires a Euclidean space, since otherwise, we cannot define a line joining two points). The distance between any two clusters is now computed as the distance between the closest pair of representative points – one belonging to each of the two clusters. Thus, only the representative points of a cluster are used to compute its distance from other clusters.

The representative points attempt to capture the physical shape and geometry of the cluster. Furthermore, shrinking the scattered points toward the mean by a factor gets rid of any anomalies caused by outliers or random noise points. The reason for this is that outliers will normally be far off from the cluster center, and thus shrinking would cause the outliers to come closer to the center while the remaining representative points would undergo very small shifts. The larger movements in the outliers would thus reduce their ability to cause the wrong clusters to be merged. The shrinking parameter can also be used to control the shapes of clusters. A smaller value shrinks the scattered points very little and, thus, favors the elongated clusters. On the other hand, with larger values the scattered points get located closer to the mean, resulting in compact clusters.

9.5.2.2 Completion Phase

Once the initialization step is complete, we now have to cluster the remaining points and output the final clusters:

1. The final step of CURE is to merge two clusters if they have a pair of representative points, one from each cluster, that are sufficiently close. The user may pick the distance threshold at which points may be considered “close”. This merging step can be repeated until there are no more sufficiently close clusters.
2. Each point p is brought from secondary storage and compared with the representative points. We assign p to the cluster of the representative point that is closest to p .

Figures 9.6 to 9.9 illustrate the working of the CURE algorithm.

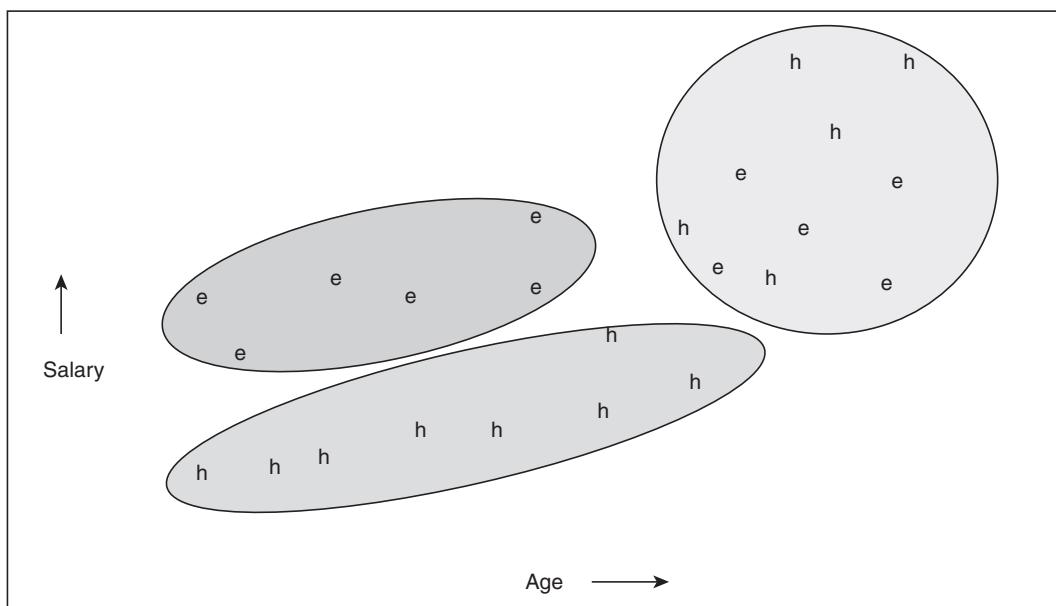
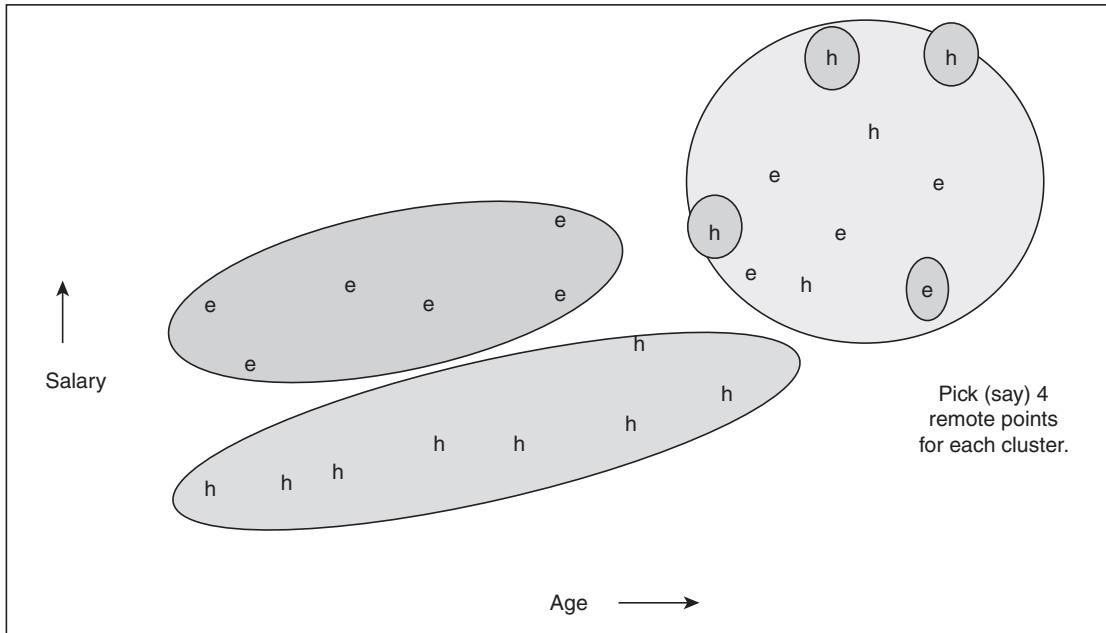
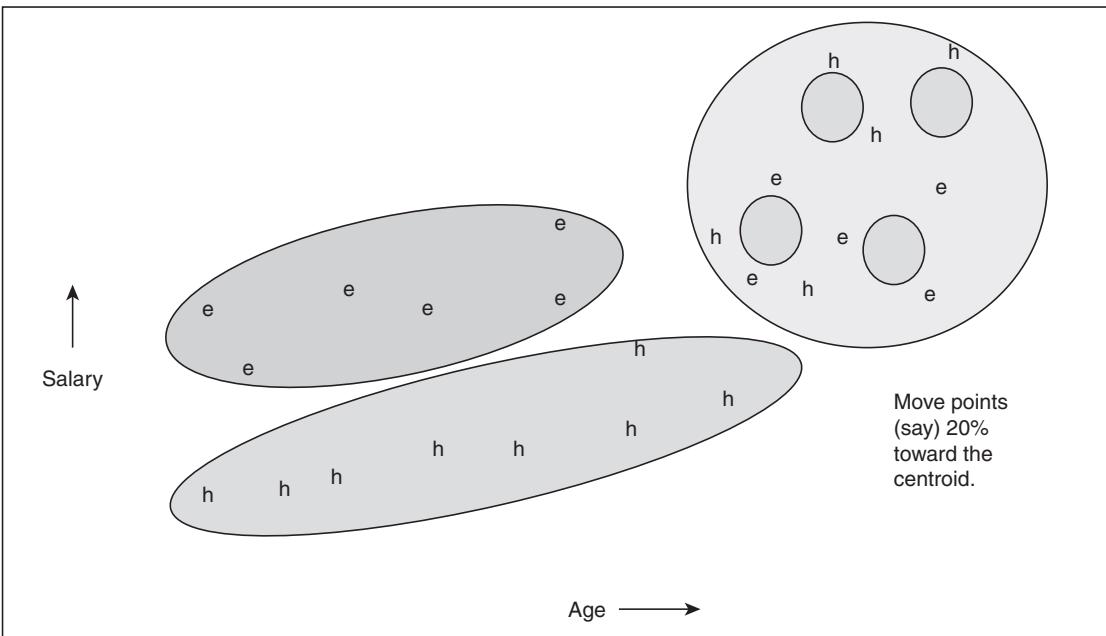


Figure 9.6 Example of initial clusters.

**Figure 9.7** Pick dispersed points.**Figure 9.8** Shrink points 20%.

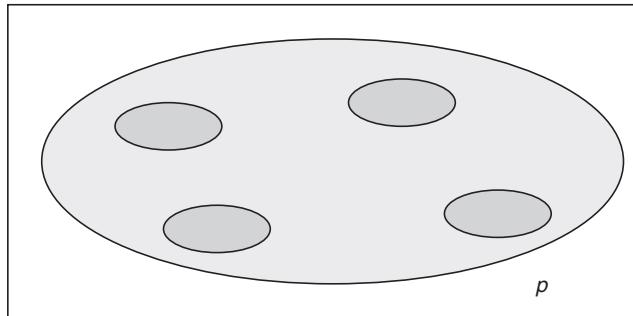


Figure 9.9 Completing CURE – Pick (say) 4 remote points for each cluster.

9.6 Clustering Streams

Most of the classical methods in clustering cannot be simply extended to be used with very large datasets or data streams. The stream scenario brings a unique set of challenges with it, as follows:

1. Data Streams are normally massive in size and thus they cannot be stored explicitly even on disk. Therefore, the data needs to be processed in a single pass, in which all the summary information required for the clustering process must be collected and stored. The time needed to process each record must be minimum and constant. Otherwise, the summarization process will not be in synchronization with the data arriving in the stream.
2. The patterns in the data stream may continuously evolve and change over time. For the purpose of stream clustering this means that we cannot have a static clustering model but we need a model capable of evolving with changes in the stream data. Further at any instant of time a current clustering must be generated so that it can be used by an analytics application. This is required also because data streams may never end.
3. Based on the domain of origin of the data stream, clustering streams face different challenges. For example, if the data values consist of very high dimensional data made up of discrete attributes, it may become impossible to store summary representations of the clusters efficiently. Therefore, space-efficient methods are needed when data for clustering such data streams.

The problem of data stream clustering can occur in two flavors: Firstly, we want to be able to access large streaming sources like network traffic, satellite imagery, streaming videos, etc., which, because of their sheer volume and very high velocity of arrival, cannot be stored for offline processing. Thus clustering can aggregate and summarize data in real time from these streams and then discard the actual data. Secondly, stream clustering methods can also be used to access very fast changing databases like web databases; Google's MapReduce computational framework is one example which uses stream clustering techniques.

In this section, we provide a very brief overview of some stream clustering algorithms. The reader is referred to the references for further information on Stream Clustering.

9.6.1 A Simple Streaming Model

A simple Stream Clustering model is based on the K -medians (K -means) clustering methodology. The central scheme is to divide stream into chunks, which can be managed and stored in main memory. We assume that the original data stream D is divided into chunks D_1, \dots, D_r, \dots , each of which contains at most m data points. The value of m is dictated by the size of the main memory. Since each chunk fits in main memory, we can use complex clustering algorithms on each chunk of data.

Each stream element is represented as one point in some dimensional space. A sliding window consists of the most recent N points. Stream clustering attempts to group subsets of the points in the stream, so that we can quickly answer queries which need the clustering structure of the last m points, for any $m \leq N$. Further given a value K we may need the last m points to be grouped into K clusters.

Further there is no restriction regarding the space in which the points of the stream may be represented. It may be a Euclidean space, in which case clustering will work with Centroids (means) of the selected clusters. If the space is non-Euclidean, clustering algorithms will process Clustroids (medians) of the selected clusters.

Further, for ease of designing efficient streaming algorithms it is assumed that the stream elements are fairly stable and their summary statistics that do not vary very much as the stream proceeds. This implies that it is sufficient to consider only a sample of the stream to estimate the clusters, and we can afford to ignore the stream after a while. However, this assumption may not be true in general, which may require further complex algorithms to cluster. For example, the Centroids of the clusters may migrate slowly as time goes on, or clusters may expand, contract, divide or merge.

9.6.2 A Stream Clustering Algorithm

We discuss a simplified version of an algorithm referred to as BDMO (for the authors, B. Babcock, M. Datar, R. Motwani and L. O'Callaghan). The true version of the algorithm provides better performance guarantees in the worst case but is quite complex and out of the scope of this book.

The BDMO algorithm builds on the methodology for counting ones in a stream that was described in Chapter 6. Some of the key similarities and differences are as follows:

1. Similar to the earlier algorithm, the points of the stream are partitioned into, and summarized by, buckets whose sizes are a power of two. Here, the size of a bucket is the number of points it represents, rather than the number of stream elements that are 1.
2. As before, the sizes of buckets follow the rule that there may be only one or two buckets of each size, up to some limit. But unlike the previous algorithm, the sequence of possible bucket sizes need not start at 1. They are required only to form a sequence where each size is twice the previous size, for example, 3, 6, 12, 24,

3. Like before Bucket sizes have to be non-decreasing as we go back in time. As was discussed in Chapter 6 we can conclude that there will be $O(\log N)$ buckets.

The contents of a bucket are as follows:

1. The size of the bucket.
2. The timestamp of the bucket, this is, the most recent point that contributes to the bucket. As before we can record the timestamps with modulo N .
3. A collection of records that represent the clustering of the points that are in this bucket. These records contain: the number of points in the cluster; the Centroid or Clustroid of the cluster; other necessary parameters useful for merging clusters and maintaining approximations to the full set of parameters for the merged cluster.

9.6.2.1 Initializing Buckets

The smallest bucket size will be p , a power of 2. Thus, for every p stream elements, we have to construct a new bucket, with the most recent p points. The timestamp for this bucket is the timestamp of the most recent point in the bucket. We can leave each point to form a cluster by itself, or we may apply some clustering strategy on the entire bucket of points. For instance, if we choose a K -means algorithm, then (assuming $K < p$) we cluster the points into K clusters.

After performing the initial clustering we compute the Centroids or Clustroids for the clusters and count the points in each cluster. This information is added to the record for each cluster. We also compute the other needed parameters for merging clusters ahead.

9.6.2.2 Merging Buckets

Whenever we create a new bucket, we need to review the sequence of buckets.

1. First, if some bucket has a timestamp that is more than N time units prior to the current time, then nothing of that bucket is in the window, and we may drop it from the list.
2. Second, we may have created three buckets of size p , in which case we must merge the oldest two of the three. The merger may create two buckets of size $2p$, in which case we may have to merge buckets of increasing sizes, recursively. (Similar to our earlier algorithm.)

To merge two consecutive buckets:

1. Record the size of the bucket as twice the size of the two buckets being merged.
2. The timestamp for the merged bucket is the timestamp of the more recent of the two consecutive buckets.
3. If we decide to merge the clusters we need to compute the parameters of the newly formed cluster.

Example 5

The simplest case is using a K -means approach in a Euclidean space. We represent the clusters by the count of their points and their Centroids. Each bucket will have exactly k clusters, so we can pick $p=k$, or $p>k$ and cluster the p points into k clusters initially. We then find the best matching between the k clusters of the first bucket and the k clusters of the second; the matching that minimizes the sum of the distances between the Centroids of the matched clusters.

When we decide to merge two clusters, one from each bucket, the number of points in the merged cluster is just the sum of the number of points in the two clusters. The Centroid of the merged cluster is the weighted average (by number of points in each cluster) of the two Centroids.

This procedure can be easily extended to clustering in non-Euclidean spaces with Clustroids instead of Centroids.

9.6.3 Answering Queries

One main goal of stream clustering is to be able to answer any queries posed on a current window of data. One common query is a request for the clustering structure of the most recent m points in the stream, where $m \leq N$. The above discussed strategy involves combining of buckets as we go back in time, and so we may not always be able to find a set of buckets that covers exactly the last m points. However, if we choose the smallest set of buckets that cover the last m points, we can easily see that the construction of these buckets restricts the number of points to no more than the last $2m$ points. This query will provide as answer, the Centroids or Clustroids of all the points in the selected buckets.

In order for the result to be a reasonable clustering for exactly the last m points, we must assume that the points between $2m$ and $m + 1$ will not have radically different statistics as compared to the most recent m points. This was our initial assumption about the data. However, if the statistics vary too rapidly, more complex bucketing strategies have to be used.

Having selected the desired buckets, we pool all their clusters. We then use some methodology for deciding which clusters to merge. For instance, if we are required to produce exactly k clusters, then we can merge the clusters with the closest centroids until we are left with only k clusters.

Summary

- **Clustering:** The goal of data clustering, also known as “Cluster Analysis”, is to discover the *natural* grouping(s) of a set of patterns, points, or objects. To cluster points, we need

a distance measure on that space. Ideally, points in the same cluster have small distances between them, while points in different clusters have large distances between them.

- **Clustering algorithms:** Clustering algorithms generally have one of two forms. Hierarchical clustering algorithms begin with all points in a cluster of their own, and nearby clusters are merged iteratively. Partitional clustering algorithms consider points in turn and assign them to the cluster in which they best fit.
- **The curse of dimensionality:** Points in high-dimensional Euclidean spaces, as well as points in non-Euclidean spaces often behave unintuitively. Two unexpected properties of these spaces are that the random points are almost always at about the same distance, and random vectors are almost always orthogonal.
- **Hierarchical clustering:** This family of algorithms has many variations, which differ primarily in two areas. First, we may choose in various ways in which two clusters to merge next. Second, we may decide when to stop the merge process in various ways.
- **K-means algorithms:** This family of algorithms is of the point-assignment type and assumes a Euclidean space. It is assumed that there are exactly k clusters for some known k . After picking k initial cluster centroids, the points are considered one at a time and assigned to the closest centroid. The centroid of a cluster can migrate during point assignment, and an optional last step is to reassign all the points, while holding the centroids fixed at their final values obtained during the first pass.
- **The BFR algorithm:** This algorithm is a version of K -means designed to handle data that is too large to fit in main memory. It assumes clusters are normally distributed about the axes. It works on chunks of data at a time. It stores a set of summary statistics only and is able to perform the full procedure of clustering. One extra pass is required to enumerate the clusters.
- **The CURE algorithm:** This algorithm is of the point-assignment type. It is designed for a Euclidean space, but clusters can have any shape. It handles data that is too large to fit in main memory. It works by selecting representative points for each cluster. However, the representative points are then moved a fraction of the way toward the centroid of the cluster, so they lie somewhat in the interior of the cluster. After creating representative points for each cluster, the entire set of points can be read from disk and assigned to a cluster. We assign a given point to the cluster of the representative point that is closest to the given point.
- **Clustering streams:** A generalization of the DGIM Algorithm (for counting 1s in the sliding window of a stream) can be used to cluster points that are part of a slowly evolving stream. The BDMO algorithm uses buckets similar to those of DGIM, with allowable bucket sizes forming a sequence where each size is twice the previous size.

Exercises

1. Suppose that the data mining task is to cluster the following eight points (with (x, y) representing location) into three clusters.

A1(2, 10), A2(2, 5), A3(8, 4), B1(5, 8), B2(7, 5), B3(6, 4), C1(1, 2), C2(4, 9). The distance function is the Euclidean distance.

Suppose initially we assign A1, B1, and C1 as the center of each cluster, respectively. Use the K -means algorithm to show only (a) the three cluster centers after the first round of execution and (b) the final three clusters.

2. Given a 1-D dataset {1,5,8,10,2}, use the agglomerative clustering algorithms with Euclidean distance to establish a hierarchical grouping relationship. Draw the dendrogram.
3. Both K -means and K -medoids algorithms can perform effective clustering. Illustrate the strength and weakness of K -means in comparison with the K -medoids algorithm. The goal of K -medoid algorithm is the same as K -means: minimize the total sum of the distances from each data point to its cluster center. Construct a simple 1-D example where K -median gives an output that is different from the result retuned by the K -means algorithm. (Starting with the same initial clustering in both cases.)
4. Suppose a cluster of 3-D points has standard deviations of 2, 3, and 5, in the three dimensions, in that order. Compute the Mahalanobis distance between the origin (0, 0, 0) and the point (1, -3, 4).
5. For the 2-D dataset (2, 2), (3, 4), (4, 8), (4, 10), (5, 2), (6, 8), (7, 10), (9, 3), (10, 5), (11, 4), (12, 3), (12, 6), we can make three clusters:
 - Compute the representation of the cluster as in the BFR algorithm. That is, compute N, SUM, and SUMSQ.

- Compute the variance and standard deviation of each cluster in each of the two dimensions.
6. Execute the BDMO algorithm with $p = 3$ on the following 1-D, Euclidean data: 1, 45, 80, 24, 56, 71, 17, 40, 66, 32, 48, 96, 9, 41, 75, 11, 58, 93, 28, 39, 77. The clustering algorithms is k -means with $k = 3$. Only the centroid of a cluster, along with its count, is needed to represent a cluster.
 7. Using your clusters from Exercise 6, produce the best centroids in response to a query asking for a clustering of the last 10 points.
 8. In certain clustering algorithms, such as CURE, we need to pick a representative set of points in a supposed cluster, and these points should be as far away from each other as possible. That is, begin with the two furthest points, and at each step add the point whose minimum distance to any of the previously selected points is maximum.

Suppose you are given the following points in 2-D Euclidean space:

$$x = (0,0); y = (10,10); a = (1,6); b = (3,7); c = (4,3); d = (7,7); e = (8,2); f = (9,5).$$

Obviously, x and y are furthest apart, so start with these. You must add five more points, which we shall refer to as the first, second,..., fifth points in what follows. The distance measure is the normal Euclidean distance. Identify the order in which the five points will be added.

Programming Assignments

1. Implement the BFR (modified K -mean) algorithm on large set of 3-d points using MapReduce.
2. Implement the CURE algorithm using MapReduce on the same dataset used in Problem 1.

References

1. B. Babcock, M. Datar, R. Motwani, L. O'Callaghan (2003). Maintaining Variance and k-medians over Data Stream Windows. *Proc. ACM Symp. on Principles of Database Systems*, pp. 234–243.
2. P.S. Bradley, U.M. Fayyad, C. Reina (1998). Scaling Clustering Algorithms to Large Databases. *Proc. Knowledge Discovery and Data Mining*, pp. 9– 15.
3. V. Ganti, R. Ramakrishnan, J. Gehrke et al. (1999). Clustering Large Datasets in Arbitrary Metric Spaces. *Proc. Inrl. Conf. on Data Engineering*, pp. 502–511.
4. H. Garcia-Molina, J.D. Ullman, J. Widom (2009). *Database Systems: The Complete Book*, Second Edition. Prentice-Hall, Upper Saddle River, NJ.
5. S. Guha, R. Rastogi, K. Shim (1998). CURE: An Efficient Clustering Algorithm for Large Databases. *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 73–84.
6. L. Rokach, O. Maimon (2005). Clustering Methods. In: O. Maimon, L. Rokach (Eds.), *Data Mining and Knowledge Discovery Handbook*. Springer, New York, pp. 321–352.
7. C. C. Aggarwal, C. Reddy (2013). *Data Clustering: Algorithms and Applications*. CRC Press.
8. M. Steinbach, L. Ertoz, V. Kumar (2003). Challenges of Clustering High Dimensional Data. In: L. T. Wille (Ed.), *New Vistas in Statistical Physics – Applications in Econophysics, Bioinformatics, and Pattern Recognition*. Springer-Verlag.

10

Recommendation Systems

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Learn the use of Recommender system.
- Understand various models of Recommender system.
- Learn collaborative filtering approach for Recommender system.
- Learn content based approach for Recommender system.
- Understand the methods to improve prediction function.

10.1 Introduction

10.1.1 What is the Use of Recommender System?

For business, the recommender system can increase sales. The customer service can be personalized and thereby gain customer trust and loyalty. It increases the knowledge about the customers. The recommender system can also give opportunities to persuade the customers and decide on the discount offers.

For customers, the recommender system can help to narrow down their choices, find things of their interests, make navigation through the list easy and to discover new things.

10.1.2 Where Do We See Recommendations?

Recommendations are commonly seen in e-commerce systems, LinkedIn, friend recommendation on Facebook, song recommendation at FM, news recommendations at Forbes.com, etc.

10.1.3 What Does Recommender System Do?

It takes the user model consisting of ratings, preferences, demographics, etc. and items with its descriptions as input, finds relevant score which is used for ranking, and finally recommends items that are relevant to the user. By estimating the relevance, information overload can be reduced. But the relevance can be context-dependent.

10.2 A Model for Recommendation Systems

Recommendation system is a facility that involves predicting user responses to options in web applications. We have seen the following recommendations:

1. “You may also like these...”, “People who liked this also liked...”.
2. If you download presentations from slideshare, it says “similar content you can save and browse later”.

These suggestions are from the recommender system. The paradigms used are as follows:

1. **Collaborative-filtering system:** It uses community data from peer groups for recommendations. This exhibits all those things that are popular among the peers. Collaborative filtering systems recommend items based on similarity measures between users and/or items. The items recommended to a user are those preferred by similar users (community data). In this, user profile and contextual parameters along with the community data are used by the recommender systems to personalize the recommendation list.
2. **Content-based systems:** They examine properties of the items recommended. For example, if a user has watched many “scientific fiction” movies, then the recommender system will recommend a movie classified in the database as having the “scientific fiction” genre. Content-based systems take input from the user profile and the contextual parameters along with product features to make the recommendation list.

10.3 Collaborative-Filtering System

Collaborative-filtering systems focus on the relationship between the users and the items. Similarity of items is determined by the similarity of the ratings of those items by the users who have rated both the items. This is the most prominent approach in commercial e-commerce sites, also applicable in domains such as books, movies and videos recommendations. The approach is to group people with similar taste, and feedback from any or some can be used to recommend items to the group. Hence, the basic assumption for this approach is (a) user gives ratings to items in the catalog, (b) customers who had similar taste in the past will have similar taste in the future too and (c)users who agreed in their subjective evaluations in the past will agree in the future too.

10.3.1 Nearest-Neighbor Technique

The idea is to predict the rating an “Active User” would give for an unseen item. The first step is to select a set of users (peers) who liked the same items as the “Active User” in the past and rated them too. In order to predict the Active User’s rating for the unseen item, we can use the average rating given by the peers for that unseen item.

To generate prediction using this technique, first we need to measure the similarity and select the peers whose ratings have to be considered. Similarity is perfect when the rating of the two is indicated by a straight line graph. The Pearson Correlation Coefficient is a measure of correlation between two variables. Using Pearson's Correlation in peer-based collaborative filtering, the formula turns out to be

$$\text{sim}(a,b) = \frac{\sum_{p \in P} (r_{a,p} - \bar{r}_a)(r_{b,p} - \bar{r}_b)}{\sqrt{\sum_{p \in P} (r_{a,p} - \bar{r}_a)^2} \sqrt{\sum_{p \in P} (r_{b,p} - \bar{r}_b)^2}}$$

where “ a ” and “ b ” are users; $r_{a,p}$ is the rating of user “ a ” for item “ p ”; “ P ” is the set of items that are rated by both “ a ” and “ b ”.

Similarity measure lies in the range of -1 and 1 , where -1 indicates very dissimilar and 1 indicates perfect similarity. A common prediction function using the above similarity function is

$$\text{pred}(a, p) = \bar{r}_a + \frac{\sum_{b \in N} \text{sim}(a, b) \times (r_{b,p} - \bar{r}_b)}{\sum_{b \in N} \text{sim}(a, b)}$$

Using this, we can calculate whether the neighbors' ratings for the unseen item i are higher or lower than their average, then combine the rating differences using the similarity as a weight. Finally the neighbor's bias is added to or subtracted from the Active User's average and used as a prediction.

10.3.2 Collaborative Filtering Example

Consider a movie rating system. Rating is done on the scale of $1\text{--}5$. Rating 1 denotes “dislike” and rating 5 “love it”. In Table 10.1, the ratings given by Jack, Tom, Dick and Harry for five different movies are given. Tim has seen four of those movies. Predict whether the fifth movie is to be recommended for Tim.

Table 10.1 Movie rating by different users

	<i>Movie 1</i>	<i>Movie 2</i>	<i>Movie 3</i>	<i>Movie 4</i>	<i>Movie 5</i>
Tim	5	3	4	4	?
Jack	3	1	2	3	3
Tom	4	3	4	3	5
Dick	3	3	1	5	4
Harry	1	5	5	2	1

Steps for finding the recommendation for Movie 5 to Tim:

1. A set of users (peers) who had seen the movies that Tim saw in the past **and** who have rated them.
2. The average of the peer ratings in the movies seen by Tim is used to find the similarity between Tim and others in order to recommend Movie 5.
3. Use Pearson Correlation for finding the similarity measure.

Table 10.2 gives the similarity measure among different users with their movie ratings.

Table 10.2 Similarity measure among different users with their movie ratings

	<i>Movie 1</i>	<i>Movie 2</i>	<i>Movie 3</i>	<i>Movie 4</i>	<i>Movie 5</i>	<i>Average</i>	<i>Sim</i>
Jack	3	1	2	3	3	2.25	0.85
Tom	4	3	4	3	5	3.5	0.7
Dick	3	3	1	5	4	3	
Harry	1	5	5	2	1	3.25	-0.79

10.3.3 Methods for Improving Prediction Function

Few of the assumptions made may not be correct. For example, all neighbor ratings may not be equally “valuable”.

Agreement on commonly liked movies is not so informative as agreement on controversial ones. Possible solutions are as follows:

1. By giving more weight to movies that have a higher variance.
2. Neighborhood selection: Giving more weight to “very similar” neighbors, that is, where the similarity value is close to 1.
3. Use similarity threshold or fixed number of neighbors.

Item-based collaborative-filtering recommendation algorithms have the following issues:

1. Scalability issues arise if there are many more users than items

$$(m \gg n, m = |\text{users}|, n = |\text{items}|)$$

For example, Amazon.com

2. High sparsity leads to a few common ratings between the two users.

The basic idea is that “Item-based CF” exploits relationships between items first, instead of relationships between the users. It produces better results in item-to-item filtering. For example: in Table 10.1, look for items that are similar to Item5. We find it is appropriate to take Tom’s ratings for these items to predict the rating for Item5.

Ratings are seen as vector in n -dimensional space. Similarity is calculated on the basis of angle between the vectors:

$$\text{sim}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \times |\vec{b}|}$$

Adjusted cosine similarity: Take average user ratings into account and transform the original ratings

$$\text{sim}(a, b) = \frac{\sum_{u \in U} (r_{u,a} - \bar{r}_u)(r_{u,b} - \bar{r}_u)}{\sqrt{\sum_{u \in U} (r_{u,a} - \bar{r}_u)^2} \sqrt{\sum_{u \in U} (r_{u,b} - \bar{r}_u)^2}}$$

where U is the set of users who have rated both items a and b .

But the problem arises when it is a cold start. How to recommend new items? What to recommend to the new users? In the initial phase, the user can be requested to rate a set of items. Otherwise either demographic data or non-personalized data can be used.

In nearest-neighbor approaches, the set of sufficiently similar neighbors might be small to make good predictions. Alternatively, transitivity of neighborhoods can be assumed.

10.4 Content-Based Recommendations

10.4.1 Discovering Features of Documents

Document collections and images are other classes of items where the values of features are not immediately apparent. When we consider documents, the technology for extracting features is important whereas in images, user-supplied features have some value.

There are many kinds of documents for which a recommendation system can prove to be useful. Web pages are also a collection of documents. For example, there are many news articles published each day, and all cannot be read. A recommendation system can suggest articles on topics a user is interested in. The question is: How to distinguish among the topics?

These classes of documents do not have readily available information giving the required features. A useful practice is the identification of words that characterize the topic of a document. After eliminating stop words and the most common words, which say little about the topic of a document, for the remaining words, (Term Frequency) TF.IDF (Inverse Document Frequency) score for each word in the document is computed. The set of n words with the highest scores or above a certain TF.IDF score (threshold) are taken as features that characterize the document.

Now, documents can be represented by sets of words. We assume these words to express the subjects or main ideas of the document. For example, in a news article, we would expect the words with the highest TF.IDF score to include the names of people discussed in the article, unusual properties of the event described, and the location of the event.

To measure the similarity of the two documents, there are several natural distance measures we use:

1. The Jaccard distance between the sets of words.
2. The cosine distance between the sets, treated as vectors.

To compute the cosine distance in option (2), think of the sets of high TF.IDF words as a vector, with one component for each possible word. The vector has 1 if the word is in the set and 0 if it is not. Since between two documents there are only a finite number of words among their two sets, the infinite dimensionality of the vectors is unimportant.

The following two kinds of document similarity exist:

1. **Lexical similarity:** Documents are similar if they contain large, identical sequences of characters.
2. **Semantic similarity:** Semantic similarity is defined over a set of documents or terms, where the idea of distance between them is based on the likeness of their meaning or semantic content.

Semantic similarity can be estimated by defining a topological similarity using ontologies to define the distance between terms/concepts. For example, a partially ordered set, represented as nodes of a directed acyclic graph, is used for the comparison of concepts ordered. Based on text analyses, semantic relatedness between units would be the shortest path linking the two concept nodes. This can also be estimated using statistical means, such as a vector space model, to correlate words and textual contexts from a suitable text corpus.

For recommendation systems, the notion of similarity is different. We are interested only in the occurrences of many important words in both documents, even if there is little lexical similarity between the documents. However, the methodology for finding similar documents remains almost the same.

Content-based recommendation system considers user profile and contextual parameters along with product features to bring out the similar kind of stuff the user liked in the past. Comparisons between items are done and no community reviews are required.

Content-based systems thus focus on properties of items. Similarity of items is determined by measuring the similarity in their properties. The content-based recommendation system needs some information related to the content (e.g. “genre”) of available items, but not complete information. It certainly requires user profile describing what the user likes (or preferences). Learning the user preferences, the system locates and recommends items similar to the user preferences.

Item Profile: In a content-based system, each item is a profile, which is a record or a collection of records representing important characteristics of that item, is first constructed. For example, for a movie recommendation system, the important characteristics are:

1. The set of actors of the movie.
2. The director.
3. The year in which the movie was made.
4. The genre or general type of movie, and so on.

Most content-based recommendation systems originate from Information Retrieval methods. The item descriptions (keywords/features) are automatically extracted. The objective of this system is to find and rank things (documents) according to the user preferences.

Books have descriptions similar to those for movies, so we can obtain features such as author, year of publication, and genre. Music products such as CDs and MP3 download have available features, such as artist, composer, and genre.

To find similarity between the items, Dice co-efficient given below is used. Let b_1, b_2 be two items. Similarity between the two is given by

$$\text{sim}(b_1, b_2) = \frac{2 * (\text{keywords}(b_1) \cap \text{keywords}(b_2))}{(\text{keywords}(b_1) + \text{keywords}(b_2))}$$

This approach has the basic assumptions that all keywords are of equal importance. There is a chance of finding more similarity with longer documents. This can be improved by using TF-IDF method.

10.4.2 Standard Measure: Term Frequency–Inverse Document Frequency

TF relates to the density of the term, that is, the frequency of the term in the document. This assumes important terms appear more number of times and normalization is done to take into account the document length. IDF reduces the weight of terms that appear in all the documents. For keyword i and document j , the importance of keywords is computed using the formula

$$\text{TF-IDF}(I, j) = \text{TF}(I, j) * \text{IDF}(i)$$

where Term Frequency is given by

$$\text{TF}(I, j) = \frac{\text{freq}(I, j)}{\text{maxOthers}(I, j)}$$

Here $\text{freq}(I, j)$ is the frequency of keyword I in document j and $\text{maxOthers}(I, j)$ is the maximum number of times another keyword of j occurred.

Inverse Document Frequency: IDF is given by

$$\text{IDF}(i) = \log(N/n(i))$$

where N is the number of all recommendable documents and $n(i)$ is the number of documents in which keyword I appears. This approach can be improved by

1. Removing stop words/articles.
2. Using stemming.
3. Limiting to the top n keywords.
4. Usage of word in negative context.
5. Using better methods for keyword/feature selection.
6. Using phrases as terms.

An information need may be expressed using different keywords (i.e., a word's various synonyms). For example: ship and boat, aircraft and airplane. The solutions to this is either refining queries manually or expanding queries semi-automatically. Semi-automatic query expansion is done by local or global methods. Local methods are based on the retrieved documents and the query (e.g., Relevance Feedback). Global methods use, for example Thesaurus, spelling corrections which are independent of the query and results.

10.4.3 Obtaining Item Features from Tags

Consider that images of features have been obtained for items. The problem with images is that their data, which is an array of pixels, does not tell us anything useful about their features. In order to obtain information about features of items, request users to tag the items by entering words or phrases that describe the item. Almost any kind of data can have its features described by tags. Users can enter a set of tags as their search query, and the system retrieves the Web pages that had been tagged that way. However, it is also possible to use the tags as are commendation system. If it is observed that a user retrieves or bookmarks many pages with a certain set of tags, then other pages with the same tags can be recommended.

The problem with tagging as an approach to feature discovery is that the process only works if users are willing to take the trouble to create the tags, and the number of erroneous tags are minimal/negligible when compared to total number of tags.

This vector approach can be generalized to all sorts of features that can be represented as Boolean or sets of discrete values. For example, if one feature of movies is the set of actors, then imagine that there is a component for each actor, with 1 if the actor is in the movie and 0 if he/she is not. Likewise, we can have a component for each possible director, and each possible genre. All these features can be represented using only 0s and 1s.

There is another class of features that is not readily represented by Boolean vectors;these features are numerical. For instance, if we take the average rating for movies to be a feature, this average is a real number.

Two ratings that are close but not identical are considered more similar than widely differing ratings. In the same way, numerical features of products, such as screen size or disk capacity, should be considered similar if their values do not differ greatly. Numerical features should be represented by single components of vectors representing items. These components hold the exact value of that feature.

To compute the cosine distance between vectors, consider the movie recommendation system. Suppose the only features of movies are the set of actors and the average rating. Consider two movies with five actors each. Two of the actors are in both movies. Also, one movie has an average rating of 3 and the other has an average of 4. The vectors look something like

$$0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 3\alpha$$

$$1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 4\alpha$$

There is no need to consider the effect of actors since cosine distance of vectors is not affected by components (both vectors have 0).

The last component represents the average rating with an unknown scaling factor α . Computing in terms of α , the cosine of the angle between the vectors is

$$\frac{2 + 12\alpha^2}{\sqrt{25 + 125\alpha^2 + 144\alpha^4}}$$

where the dot product is $2 + 12\alpha^2$ and the length of the vectors are $\sqrt{5+9\alpha^2}$ and $\sqrt{5+16\alpha^2}$.

For $\alpha = 1$, the cosine is 0.816.

For $\alpha = 2$, the cosine is 0.940, the vectors are closer in direction than if we use $\alpha = 1$.

10.4.4 User Profiles

To consider the user profiles, vectors with the same components that describe user's preferences are created. The utility matrix represents the connection between the users and the items. The entries in the utility matrix to represent user purchases or a similar connection could be just 1s, or they could be any number representing a rating or liking that the user has for the item. This information can be used to find the best estimate regarding which items the user likes. This is taken as aggregation of the profiles of those items. If the utility matrix has only 1s, then the natural aggregate is the average of the components of the vectors representing the item profiles for the items in which the utility matrix has 1 for that user.

Suppose items are movies, represented by Boolean profiles with components corresponding to actors. Also, the utility matrix has a 1 if the user has seen the movie and is blank otherwise. If 25% of the movies that user U likes have Tom Hanks as one of the actors, then the user profile for U will have 0.25 in the component for Tom Hanks.

If the utility matrix is not Boolean, for example, ratings 1–5, then we can weigh the vectors representing the profiles of items by the utility value. The utilities are normalized by subtracting the average value for a user. For example, user U gives an average rating of 3, and has also rated three movies with Tom Hanks.

User U gives these three movies ratings of 1, 3 and 4. The user profile for U has, in the component for Tom Hanks, the average of $1 - 3, 3 - 3$ and $4 - 3$, that is, the value $-1/3$. Therefore, items with a below-average rating get negative weights, and items with above-average ratings get positive weights. With profile vectors for both users and items, the degree to which a user would prefer an item can be estimated by computing the cosine distance between the user's and item's vectors.

The vector for a user will have positive numbers for actors who appear in movies the user likes and have negative numbers for actors appearing in movies the user does not like.

1. **Case 1:** Consider a movie with many actors the user likes, and only a few or none that the user does not like. The cosine of the angle between the user's and movie's vectors will be a large positive fraction. That implies an angle close to 0, and therefore a small cosine distance between the vectors.
2. **Case 2:** Consider a movie with as many actors that the user likes as those the user does not like. In this situation, the cosine of the angle between the user and movie is around 0, and therefore the angle between the two vectors is around 90° .
3. **Case 3:** Consider a movie with mostly actors the user does not like. In that case, the cosine will be a large negative fraction, and the angle between the two vectors will be close to 180° —the maximum possible cosine distance.

10.4.5 Classification Algorithms

Another approach to a recommendation system is to treat this as the machine learning problem using item profiles and utility matrices. The given data is taken as a training set, and for each user, classifier that predicts the rating of all items is built.

There are a number of different classifiers, and the most common one is the decision trees. A decision tree is a collection of nodes, arranged as a binary tree. The leaves render decisions; in movie recommendation system, the decision would be “likes” or “does not like”. Each interior node is a condition on the objects being classified; here the condition would be a predicate involving one or more features of an item. To classify an item, starting from the root, the predicate is applied from the root to the item. If the predicate is true, the path to the left (child) is taken, and if it is false, the path to right (child) is taken. Then the same process is repeated at all the nodes visited, until a leaf is reached. It is that leaf which classifies the item as liked or not.

After selecting a predicate for a node N , the items are divided into two groups: one that satisfies the predicate and the other that does not. For each group, the predicate that best separates the positive and negative examples in that group is then found. These predicates are assigned

to the children of N . This process of dividing the examples and building children can proceed to any number of levels. We can stop, and create a leaf, if the group of items for a node is homogeneous, that is, they are all positive or all negative examples. However, we can stop and create a leaf with the majority decision for a group, even if the group contains both positive and negative examples. The reason is that the statistical significance of a small group may not be high enough to rely on. For that reason, a variant strategy to an ensemble of decision trees, each using different predicates, but allowing the trees to be deeper than what the available data justifies, is created. Such trees are called over-fitted trees.

Summary

- Recommender system is a tool for providing suggestions to a user when abundant information is available and decision-making becomes difficult. The system assists the users in various decision-making processes, such as what items to buy, what music to download, which movies to see, etc.
- Fundamental techniques used for building recommender systems, are collaborative filtering and content-based filtering.
- Recommendation system deals with users and items. It takes the user model consisting of ratings, preferences, demographics, etc. and items with its descriptions as input, finds relevant score which is used for ranking, and finally recommends items that are relevant to the user. Utility matrix offers known information about the degree to which a user likes an item and predicts the values of the unknown entries based on the values of the known entries.
- **Collaborative filtering systems** recommend items based on similarity measures between users and/or items. The items recommended to a user are those preferred by similar users. In this, user profile and contextual parameters along with community data are used by recommender systems to personalize the recommendation list.
- **Nearest-neighbor technique:** Use the average rating of the peers to rate the unseen item by the Active User. To generate prediction using this technique, first we need to measure similarity and select the peers whose ratings have to be considered. Similarity is perfect when the rating of the two is indicated by a straight line graph. The Pearson Correlation Coefficient is a measure of correlation between two variables.
- **Content-based systems** examine properties of the items recommended. Content-based systems take input from user profile and contextual parameters along with product features to make the recommendation list.
- **Classification algorithms:** Another approach to recommendation system is to treat this as the machine learning problem using item profiles and utility matrices. The given data is taken as a training set, and for each user, classifier that predicts the rating of all items is built.

Review Questions

1. What is the use of recommendation system?
2. Where do we see recommendations?
3. Explain collaborative filtering system. How is it different from content-based system?
4. How is similarity measure used in selecting peers for prediction in recommendation system? Give an example and explain this technique
5. What are the shortcomings of nearest neighbor technique in collaborative filtering method? Suggest some improvements.
6. How would you get the features of the document in content-based system? Explain document similarity.
7. What are the shortcomings of profile-based recommendation and how it can be improved by TF-IDF?
8. How is classification algorithm used in recommendation system?

Laboratory Exercise: Movie Recommendation System based on User Rating

Sample data.txt

196,242,3,881250949

186,302,3,891717742

22,377,1,878887116

.....

Code for movie Recommendation system based on user rating using correlation method in Pig

-- Loading database

```
grunt>movies_ratings = LOAD '/user/cloudera/pig/example/data.txt' USING PigStorage(',') AS  
(user_id:int, movie_id:int, rating:int);
```

-- Starting by limiting the dataset to movies with at least 35 ratings ;

```
grunt> A = GROUP movies_ratings BY movie_id ;
```

```
grunt> C = FOREACH A GENERATE group AS movie_id, COUNT($1) AS count ;
```

```
grunt> D = FILTER C BY count >= 35 ;
```

```
grunt> E = FOREACH D GENERATE movie_id AS movies_ok ;
```

```
grunt> F = JOIN movies_ratings BY movie_id, E BY movies_ok ;
```

```
grunt> filtered = FOREACH F GENERATE user_id, movie_id, rating ;
-- Creating co-ratings with a self join ;
grunt> filtered_2 = FOREACH F GENERATE user_id AS user_id_2, movie_id AS movie_id_2,
rating AS rating_2 ;
grunt> pairs = JOIN filtered BY user_id, filtered_2 BY user_id_2 ;
-- Eliminate dupes ;
grunt> I = FILTER pairs BY movie_id < movie_id_2 ;
-- Core data ;
grunt> J = FOREACH I GENERATE
>>movie_id ,
>> movie_id_2 ,
>>rating ,
>> rating_2 ,
>> rating * rating AS ratingSq ,
>> rating_2 * rating_2 AS rating2Sq ,
>> rating * rating_2 AS dotProduct ;
grunt> K = GROUP J BY (movie_id, movie_id_2) ;
grunt> co = FOREACH K GENERATE
>>group ,
>>COUNT(J.movie_id) AS N ,
>>SUM(J.rating) AS ratingSum ,
>>SUM(J.rating_2) AS rating2Sum ,
>>SUM(J.ratingSq) AS ratingSqSum ,
>>SUM(J.rating2Sq) AS rating2SqSum ,
>>SUM(J.dotProduct) AS dotProductSum ;
grunt>coratings = FILTER co BY N >= 35 ;
grunt> recommendations = FOREACH coratings GENERATE
>>group.movie_id ,
>> group.movie_id_2 ,
```

```
>> (double)(N * dotProductSum - rating-
Sum * rating2Sum) / ( SQRT((double)(N
* ratingSqSum - ratingSum * ratingSum)) *
SQRT((double)(N * rating2SqSum - rating-
2Sum * rating2Sum)) ) AS correlation ;
grunt> O = ORDER recommendations BY
movie_id, correlation DESC ;
grunt> P = LIMIT O 60 ;
grunt> DUMP P ;
```

Sample O/P:

```
(1,819,0.6234596040989883)
(1,928,0.5490995330508588)
(1,562,0.494305135376554)
(1,931,0.4904910708478703)
.......
```

11

Mining Social Network Graphs

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Get introduced to the concept of a social network and understand the different types of networks and their applications.
- Learn the concept of representing a Social Network as a Graph.
- Understand the importance of the concept of communities in a social graph.
- Learn about clustering techniques that can be used for community detection in a social graph.
- Learn algorithms for Direct Discovery of Communities in a social network.
- Learn about some social network based concepts like SimRank and Counting triangles and algorithms to compute them.

11.1 Introduction

The increasing popularity of social networks is clearly demonstrated by the huge number of users acquired in a short amount of time. Social networks now have gathered hundreds of millions of users, prominent among them being Facebook, MySpace, Twitter, LinkedIn, Instagram, etc. Further data from these networks increases in volumes at every instance in time. Facebook has over one billion active users with thousands of new users added every day. The number of interactions added everyday amounts to millions. Moreover, most of these networks allow interactions using different media like voice, images, and even videos. Thus, social network data is “Big Data”.

Massive quantities of data on very large social networks are now freely available from many social networks like blogs, social networking sites, newsgroups, chat rooms, etc. These networks typically contain substantial quantities of information at the level of the individual node. Mining these structures for patterns yields knowledge that can effectively be used for predictions and decision-making.

In this chapter, we provide a brief introduction to mining such networks. We will first introduce the notion of a social network as a graph and then discuss different types of social networks and their applications.

One very fundamental task needed for social network mining is community detection which basically identifies dense subgraphs from the social network graphs. We discuss clustering techniques which can be used to identify communities in a social network scenario.

We further look at algorithms for two very interesting problems in social networks. The “SimRank” algorithm provides a way to discover similarities among the nodes of a graph. We also explore triangle counting as a way to measure the connectedness of a community.

11.2 Applications of Social Network Mining

Applications of social network mining encompass numerous fields. A few popular ones are enumerated here:

1. Viral marketing is an application of social network mining that explores how individuals can influence the buying behavior of others. Viral marketing aims to optimize the positive word-of-mouth effect among customers. Social network mining can identify strong communities and influential nodes. It can choose to spend more money marketing to an individual if that person has many social connections.
2. Similarly in the e-commerce domain, the grouping together of customers with similar buying profiles enables more personalized recommendation engines. Community discovery in mobile ad-hoc networks can enable efficient message routing and posting.
3. Social network analysis is used extensively in a wide range of applications, which include data aggregation and mining, network propagation modeling, network modeling and sampling, user attribute and behavior analysis, community-maintained resource support, location-based interaction analysis, social sharing and filtering, recommender systems, etc.
4. Many businesses use social network analysis to support activities such as customer interaction and analysis, information system development analysis, targeted marketing, etc.

11.3 Social Networks as a Graph

When the objective is to mine a social network for patterns, a natural way to represent a social network is by a graph. The graph is typically very large, with nodes corresponding to the objects and the edges corresponding to the links representing relationships or interactions between objects. One node indicates one person or a group of persons.

As an example, consider a set of individuals on a social networking site like LinkedIn. LinkedIn allows users to create profiles and “connections” with each other in an online social network which may represent real-world professional relationships. Thus, finding a person on a LinkedIn network is like finding a path in a graph representing the social network. Similarly, a company may want to find the most influential individual in a social network to perform targeted advertising. The hope is that more influential a person, more people can be influenced to buy their goods. This means identifying nodes with very high out-degree in the graph representing the social network.

Finding communities or clustering social networks, all lead to the standard graph algorithms when the social network is modeled as a graph. This section will give an overview of how a social network can be modeled as a graph. A small discussion of the different types of graphs is also provided.

11.3.1 Social Networks

The web-based dictionary Webopedia defines a social network as “A social structure made of nodes that are generally individuals or organizations”. A social network represents relationships and flows between people, groups, organizations, computers or other information/knowledge processing entities. The term “Social Network” was coined in 1954 by J. A. Barnes. Examples of social networks include Facebook, LinkedIn, Twitter, Reddit, etc.

The following are the typical characteristics of any social network:

1. In a social network scenario, the nodes are typically people. But there could be other entities like companies, documents, computers, etc.
2. A social network can be considered as a **heterogeneous** and **multi-relational** dataset represented by a graph. Both nodes and edges can have attributes. Objects may have class labels.
3. There is at least one relationship between entities of the network. For example, social networks like Facebook connect entities through a relationship called friends. In LinkedIn, one relationship is “endorse” where people can endorse other people for their skills.
4. In many social networks, this relationship need not be yes or no (binary) but can have a degree. That means in LinkedIn, we can have a degree of endorsement of a skill like say novice, expert, etc. Degree can also be a real number.
5. We assume that social networks exhibit the property of non-randomness, often called locality. Locality is the property of social networks that says nodes and edges of the graph tend to cluster in communities. This condition is the hardest to formalize, but the intuition is that the relationships tend to cluster. That is, if entity A is related to both B and C, then there is a higher probability than average that B and C are related. The idea is most relationships in the real worlds tend to cluster around a small set of individuals.

11.3.2 Social Network Graphs

A natural representation of social network data is to view its structure as a graph. The basic process is as follows. Entities are converted to nodes in the graph. The nodes in this graph can be of different types, for example, people, organizations and events. This model, thus, slightly deviates from the standard model of a graph in which the node set is one type. Edges in the graph correspond to the relations between entities. If there is a degree associated with the relationship, this degree is represented by labeling the edges. Edges can be one-directional, bi-directional and are not required to be binary.

Example 1

Figure 11.1 shows a small graph of the “followers” network of Twitter. The relationship between the edges is the “follows” relationship. Jack follows Kris and Pete shown by the direction of the edges. Jack and Mary follow each other shown by the bi-directional edges. Bob and Tim follow each other as do Bob and Kris, Eve and Tim. Pete follows Eve and Bob, Mary follows Pete and Bob follows Alex. Notice that the edges are not labeled, thus follows is a binary connection. Either a person follows somebody or does not.

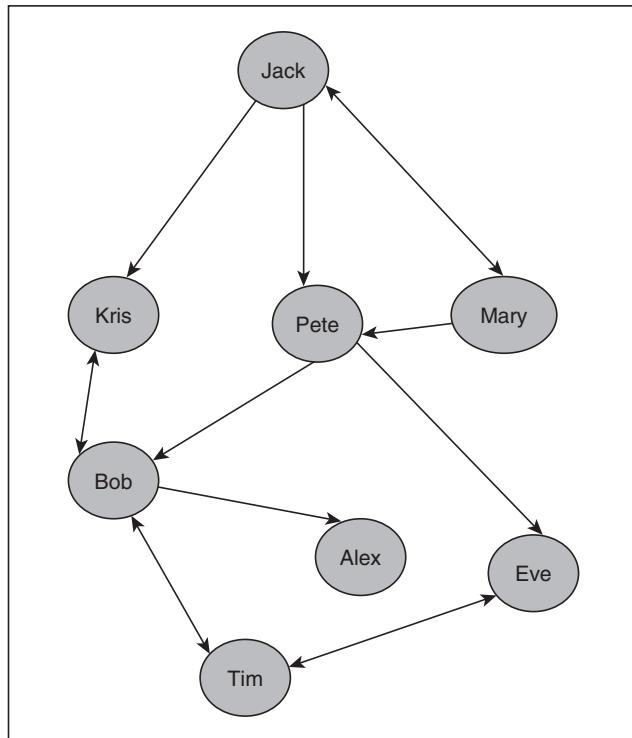


Figure 11.1 A social graph depicting follows relationship in Twitter.

Example 2

Consider LiveJournal which is a free on-line blogging community where users declare friendship to each other. LiveJournal also allows users to form a group which other members can then join. The graph depicted in Fig. 11.2 shows a portion of such a graph. Notice that the edges are undirected, indicating that the “friendship” relation is commutative.

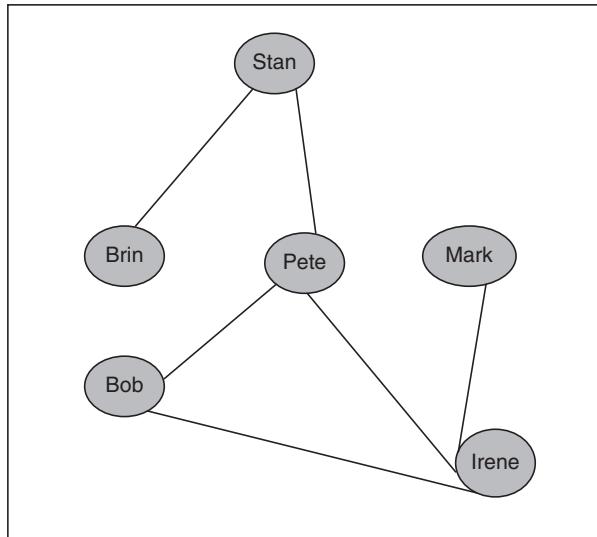


Figure 11.2 Social graph of LiveJournal.

Example 3

As a third example, we consider DBLP. Computer science bibliography provides a comprehensive list of research papers in computer science. As depicted in Fig. 11.3, the graph shows a co-authorship relationship where two authors are connected if they publish at least one paper together. The edge is labeled by the number of papers these authors have co-authored together.

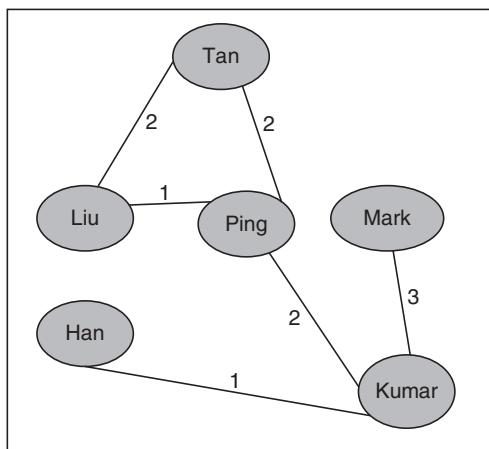


Figure 11.3 Social graph of DBLP.

11.4 Types of Social Networks

While Facebook, Twitter and LinkedIn might be the first examples that come to mind when discussing social networks, these examples in no way represent the full scope of social networks that exist. In this section, we give a brief overview of different scenarios that may result in social networks. This is by no means an exhaustive list. The interested user can refer to “<http://snap.stanford.edu/data>” for a rather large list of social network examples.

1. **Collaboration Graph:** Collaboration graphs display interactions which can be termed collaborations in a specific setting; co-authorships among scientists and co-appearance in movies by actors and actresses are two examples of collaboration graphs. Other examples include the Wikipedia collaboration graph (connecting two Wikipedia editors if they have ever edited the same article), or in sport the “NBA graph” whose vertices are players where two players are joined by an edge if they have ever played together on the same team.
2. **Who-Talks-to-Whom Graphs:** These model a type of social network also called “Communication” network. The best example is the Microsoft Instant Messenger (IM) graph which captures a snapshot of a large group of users engaged in several billion conversations in a time period. The edges between the two nodes of the graph, say, A and B denote that A “talked-to” B in this time period. Edges can be labeled with the strength of the communication too. Examples of similar networks include the email communication networks with edges representing communication between the entities. One good example is the Enron email communication network which captured all the email communication within a dataset of around half million emails. Nodes of the network are email addresses and if an address i sent at least one email to address j , the graph contains an undirected edge from i to j . Similar networks have also been constructed from records of phone calls: Researchers have studied the structure of *call graphs* in which each node is a phone number, and there is an edge between two if they engaged in a phone call over a given observation period.
3. **Information Linkage Graphs:** Snapshots of the web are central examples of such network datasets. Nodes are Web pages and directed edges represent links from one page to another. The web consists of hundreds of millions of personal pages on social networking and blogging sites, each connecting to web pages of interest to form communities. One good example is the Berkeley-Stanford web graph where nodes represent pages from berkeley.edu and stanford.edu domains and directed edges represent hyperlinks (possible connections) between them.
Other examples include product co-purchasing networks (nodes represent products and edges link commonly co-purchased products), Internet networks (nodes represent computers and edges communication), road networks (nodes represent intersections and edges roads connecting the intersections), location-based online social networks, Wikipedia, online communities like Reddit and Flickr, and many such more. The list is endless.
4. **Heterogeneous Social Networks:** Finally, a social network may have heterogeneous nodes and links. This means that the network may consist of different types of entities (multi-mode network)

and relationships (multi-relational network). For example, a simple co-purchasing network may have product nodes and customer nodes. Edges may exist if a customer purchases a product. Similarly, in a typical e-commerce platform like Amazon, it is reasonable to assume the existence of distributor information to add a third type of node. We now have edges between a customer and a product, a product sold by a distributor, and also a customer and a distributor. Different customers could buy the same product from different distributors. The UCI KDD Archive has a social network dataset about movies which contains eight different node types, including actor, movie, role, studio, distributor, genre, award and country and links between them. The natural way to represent such information is as a k -partite graph for some $k > 1$, where k is the number of different node types. In general, a k -partite graph consists of k disjoint sets of nodes, with no edges between nodes of the same set. One special graph is a bi-partite graph where $k = 2$.

Example 4

Figure 11.4 shows an example of a tri-partite graph where $k = 3$. We can assume three types of nodes to indicate actors, movies and directors. Notice no edge exists between the nodes of the same type.

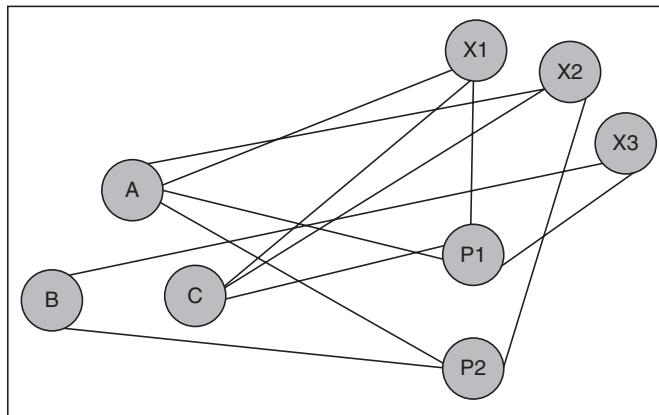


Figure 11.4 A tri-partite social graph.

11.5 Clustering of Social Graphs

The discovery of “communities” in a social network is one fundamental requirement in several social network applications. Fundamentally, communities allow us to discover groups of interacting objects (i.e., nodes) and the relations between them. Typically in a social network we can define a “community” as a collection of individuals with dense relationship patterns within a group and sparse links outside the group.

There are many reasons to seek tightly knit communities in networks; for instance, target marketing schemes can benefit by identifying clusters of shoppers and targeting a campaign wholly customized for them. Communities allow us to discover groups of interacting objects (i.e., nodes) and the relations between them. For example, in co-authorship networks, communities correspond to a set of common scientific disciplines and a closely knit set of researchers on that topic. Identifying clusters of customers with similar interests in the network of purchase relationships between customers and products of online retailers enables to set up efficient recommendation systems.

Community can be considered as a summary of the whole network, thus easy to visualize and understand. Sometimes, community can reveal the properties without releasing the individual privacy information.

One simple method for finding communities in a social graph is to use clustering techniques on the social graph. In this section, we first indicate why traditional clustering techniques like K -means and Hierarchical clustering cannot just be extended to social graphs. We shall then discuss a popular graph clustering technique called the Girvan–Newman Algorithm.

11.5.1 Applying Standard Clustering Techniques

As we are aware, to cluster similar data points in a social graph into groups, we need to decide on some measure of similarity or dissimilarity. When the social graph has labeled edges, we can use that value itself as a measure of similarity or dissimilarity between the two nodes based on what the edge represents. For example, if the social graph depicts the DBLP network discussed earlier, the edge indicates the number of papers the authors have co-authored and thus use that as some notion of similarity between the two authors. But most social graphs would be unlabeled and thus we cannot design a similarity measure.

One solution will be to have a binary distance measure, 1 for no edge and 0 for edge existing between two nodes. Any two values could be used as long as the distance for no edge is greater than the distance for an edge. The main problem is that such a measure will not satisfy the triangular inequality property. If edge AB and edge BC exist and no edge AC exists then $\text{dist}(AB) + \text{dist}(BC) < \text{dist}(AC)$. This is because the notion of distance does not have an inherent meaning in a social network. We can still try to solve this problem by choosing appropriate distance values for existence and non-existence of an edge. One example will be to use a distance of 1 for an edge and distance of 1.5 for a missing edge. We shall now use traditional clustering methods with this distance measure.

Example 5

Consider a sample social graph as depicted in Fig. 11.5. Visually we can see that $\{1, 2, 3, 4\}$ and $\{5, 6, 7, 8\}$ form two clusters (communities). But it can equally be argued that using the above-mentioned distance measure, $\{5, 6, 7\}, \{6, 7, 8\}, \{4, 5, 6\}, \{1, 2, 3\}$ is an equally good clustering. This is an overlapping clustering which would never be identified by a traditional hierarchical clustering algorithm.

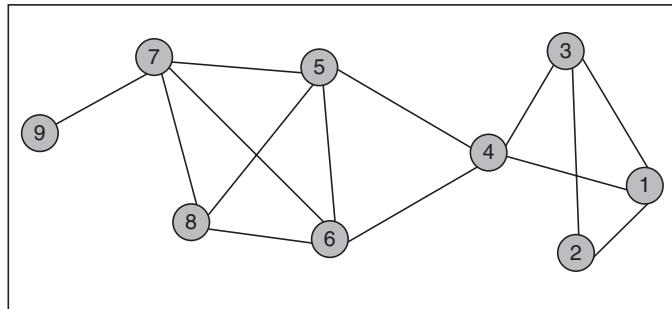


Figure 11.5 Sample social graph.

Further hierarchical clustering with the above-discussed distance measure has a tendency to separate single peripheral vertices from the communities to which they should rightly belong. For example, node 9 in Fig. 11.5. If a node is connected to the rest of a network by only a single edge, then instead of adding to the cluster formed by $\{5, 6, 7, 8\}$, it is left isolated. Further, the presence of a node like node 5 in Fig. 11.5 allows the hierarchical clustering algorithm to combine all the nodes $\{1, 2, 3, 4, 5, 6, 7, 8\}$ into one large cluster.

Since the distance measure is same for all edges, these problems will be magnified in a large graph leading to meaningless clusters. Let us now consider how a partition-based algorithm like K -means will work on clustering social networks. We shall see that here also, the fact that all edges are at the same distance will introduce random results which will lead to wrong assignment of nodes to a cluster.

Example 6

Consider again the graph of Fig. 11.5. K -means clustering will output clusters based on the initial choice of centroids. Let us say $K=2$, and initial choices for centroids are, say, 1 and 3 which belong to the same cluster. Because every edge has same cost, we will land with wrong clusters. (Convince yourself that this is true.) Assume the initial centroids are 4 and 8. Since both nodes 5 and 6 are equidistant from both the centroids, they could be clustered with either even though they should be clustered with 8.

The above problem is compounded in a large social network where any choice of K -means, even far off in the graph, will lead to a bad clustering because at some point a large number of nodes will be equidistant from many means. Further like hierarchical clustering, K -means does not discover overlapping clusters.

In the next section, we define other distance measures and a more practical algorithm for clustering of social networks.

11.5.2 Introducing “Betweenness” Measure for Graph Clustering

A major problem with using traditional clustering techniques was in the choice of distance measure used. These distance measures did not capture the essence of what a cluster in a social graph means. Our aim is to extract dense sub-graphs from the social graph. It will be the case that in a graph we can have a set of dense sub-graphs connected to other dense sub-graphs through a minimum set of edges. If we can identify and remove these connecting edges, we land up with the dense sub-graphs that form our communities.

This motivates the definition of a distance measure which is called “Edge Betweenness” associated with each edge in the social graph. Edge betweenness (EB) is defined in such a way that the edges that are likely to connect different dense regions (communities) of the graph have higher betweenness scores than edges within the communities. Hence, by identifying and progressively discarding edges with high betweenness scores, one can extract communities from a social network.

A formal definition of “edge betweenness” can be given as follows: *For an edge e in a graph, edge betweenness of e is defined as the number of shortest paths between all node pairs (v_i, v_j) in the graph such that the shortest path between v_i and v_j passes through e .*

If there are k different shortest paths between v_i and v_j we divide the number by “ k ”.

Example 7

Consider the graph of Fig. 11.5 reproduced again as Fig. 11.6.

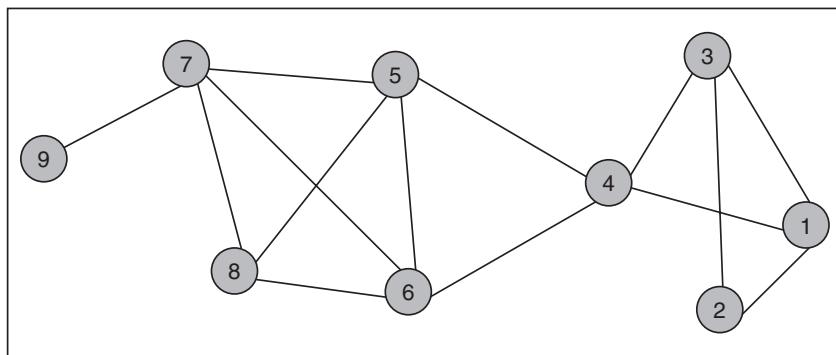


Figure 11.6 A social graph.

We shall denote edge betweenness of edge e as $EB(e)$. We have

1. $EB(1, 2)$ is 4 ($= 6/2 + 1$), as all the shortest paths from 2 to $\{4, 5, 6, 7, 8, 9\}$ go through either $e(1, 2)$ or $e(2, 3)$, and $e(1,2)$ is the shortest path between 1 and 2.

2. $EB(4,5) = \{\text{from } 9, 7, 8 \text{ to } 4, 3, 2, 1 \text{ either through } e(4,5) \text{ or } e(4,6) = 12/2 = 6; \text{ plus from } 5 \text{ to } 1, 2, 3, 4 = 4. \text{ This yields } 6 + 4 = 10.\}$
3. Similarly $EB(4,6) = 10; EB(5,7) = EB(6,7) = 6$ and so on.

As an exercise try and compute the EB of all other edges of the graph.

11.5.3 Girvan–Newman Algorithm

Girvan and Newman proposed a hierarchical divisive clustering technique for social graphs that use EB as the distance measure. The basic intuition behind this algorithm is that edges with EB are the most “vital” edges for connecting different dense regions of the network, and by removing these edges we can naturally discover dense communities. The algorithm is as follows:

1. Calculate EB score for all edges in the graph. We can store it in a distance matrix as usual.
2. Identify the edge with the highest EB score and remove it from the graph. If there are several edges with the same high EB score, all of them can be removed in one step. If this step causes the graph to separate into disconnected sub-graphs, these form the first-level communities.
3. Re-compute the EB score for all the remaining edges.
4. Repeat from step 2. Continue until the graph is partitioned into as many communities as desired or the highest EB score is below a pre-defined threshold value.

For the graph of Fig. 11.6, the following will be the order of edge removal (see Fig. 11.7):

- $e(4, 5)$ (or $e(4, 6)$). By removing $e(4, 5)$.
- We re-compute the EB. $e(4, 6)$ has the highest betweenness value: 20. This is because all shortest paths between $\{1,2,3,4\}$ and $\{5,6,7,8,9\}$ pass $e(4, 6)$.
- Continuing in this way we get three communities as $\{1, 2, 3, 4\}$, $\{5, 6, 7, 8\}$, $\{9\}$.

The major cost for the above algorithm is finding the EB score for every edge in the graph. We need to find the number of shortest paths from all nodes. Girvan and Newman proposed a faster algorithm based on use of Breadth First Search (BFS) algorithm. An overview of their algorithm is given below. The reader is referred to any graph theory book for more details.

For each node N in the graph

1. Perform breadth-first search of graph starting at node N .
2. Determine the number of shortest paths from N to every other node.
3. Based on these numbers, determine the amount of flow from N to all other nodes that use each edge.
4. Divide sum of flow of all edges by 2.

The references describe the method in greater detail.

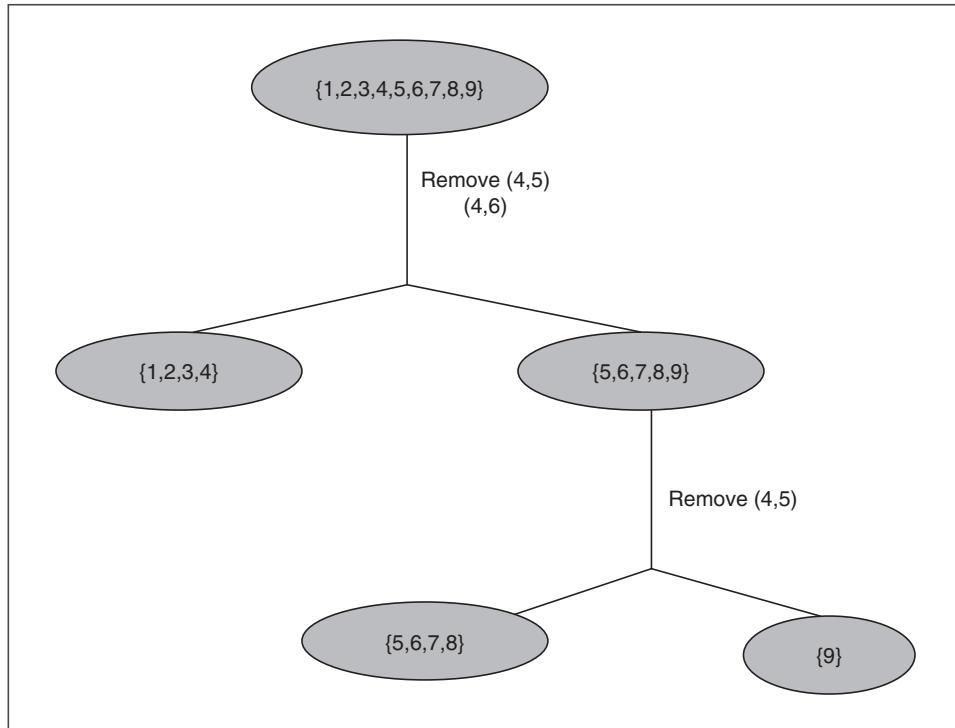


Figure 11.7 Dendrogram illustrating Girvan–Newman Algorithm.

11.6

Direct Discovery of Communities in a Social Graph

In general, we can classify communities in a social network as a group of entities which are closely knit and can belong strictly to a single community or can belong to more than one community. The former defines mutually **disjoint** communities whereas the latter definition pertains to a concept called **overlapping** communities. Many examples are available for both these types of communities but in the social network arena where nodes are individuals, it is natural that the individuals can belong to several different communities at a time and thus overlapping communities would be more natural. In a Twitter network, we can have individuals following several other individuals at the same time and thus participating in several communities simultaneously.

The last section discussed a few algorithms which resulted in mutually disjoint communities. Further these algorithms used graph partitioning techniques for identifying communities. In this section we give a very brief bird's eye view of several other community detection techniques that can also identify overlapping communities. A more detailed explanation is beyond the scope of this text.

11.6.1 Clique Percolation Method (CPM)

This method is based on the concept of finding cliques in a graph. Recall from graph theory that a “clique” in a graph is a fully connected sub-graph. (Every pair of vertices in the sub-graph is connected through an edge.) The problem of finding all cliques of a given size in a graph is an NP-hard problem. In community detection literature, a “ k -clique”, indicates the clique consisting of k vertices. For example, a 6-clique indicates a complete sub-graph having six vertices.

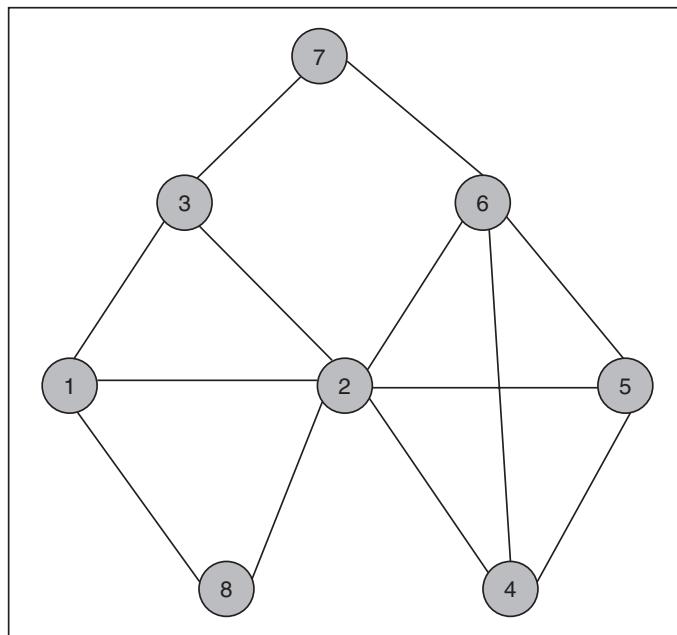


Figure 11.8 A sample social graph.

Example 8

For the graph depicted in Fig. 11.8, we have six 3-cliques $(1, 2, 3), (1, 2, 8), (2, 4, 5), (2, 4, 6), (2, 5, 6)$ and one 4-clique $(2, 4, 5, 6)$.

Clique Percolation Method (CPM) uses cliques in a graph to identify overlapping communities. The intuition is that “In a dense community it is likely to find a large number of edges and thus cliques. Further it is unlikely that edges between communities, that is, inter-community edges, form cliques”. CPM works on the assumption that community is normally formed from overlapping cliques. So this algorithm detects communities by searching for adjacent cliques. We first extract all the k -cliques (clique of size k) in the network. When these have been identified, a new graph called as

“Clique-Graph” is constructed. In this graph each extracted k -clique is compressed as one vertex. We connect two vertices in this clique-graph if the cliques represented by them have $(k - 1)$ members in common. Each connected sub-graph in the clique-graph thus represents one community. The algorithm is as shown below:

Algorithm

Clique Percolation Method

Input: The social graph G , representing a network, and a clique size, k

Output: Set of Discovered Communities, C

Step 1. All k -cliques present in G are extracted.

Step 2. A new graph, the clique-graph, G_C is formed where each node represents an identified clique and two vertices (clique) in G_C are connected by an edge, if they have $k - 1$ common vertices.

Step 3. Connected components in G_C are identified.

Step 4. Each connected component in G_C represents a community.

Step 5. Set C be the set of communities formed for G .

Example 9

For the graph described in Fig. 11.8, we have seen that it contains six 3-cliques. We now form a clique-graph with six vertices, each vertex representing one of these following six cliques:

$$\mathbf{a:(1, 2, 3); b:(1, 2, 8); c:(2, 4, 5); d:(2, 4, 6); e:(2, 5, 6); f:(4, 5, 6)}$$

Figure 11.9 depicts the clique graph so formed.

In the clique-graph since $k = 3$, we add an edge if two cliques share minimum of two vertices. Clique a and clique b have vertices 1 and 2 in common; therefore, they will be connected through an edge. Similarly we add the other edges to form the clique graph G_C as shown in Fig. 11.9. Connected components in G_C are (a, b) and (c, d, e, f) and these form the communities. So, in this case the two connected components correspond to two communities:

1. $c1 : (1, 2, 3, 8)$
2. $c2 : (2, 4, 5, 6)$

Thus the community set $C = \{c1, c2\}$, where vertex 2 overlaps both the communities. Vertex 7 is not part of any community as it is not a part of any 3-cliques.

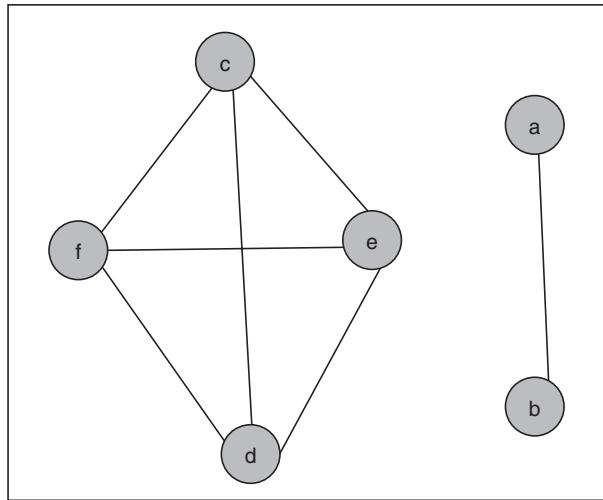


Figure 11.9 Corresponding clique-graph.

11.6.2 Other Community Detection Techniques

The main feature of CPM is that it tries to find fully connected sub-graphs to discover communities. Even though this is a very good criterion for community detection, it is too rigid a definition to be practically useful. If one edge of an otherwise complete sub-graph is missing or if two k -cliques overlap by only $k - 2$ vertices, CPM would not identify them as a community. Further, CPM makes the often erroneous assumption that all communities inherently have the same structure. Additionally, identifying k -cliques of arbitrary sizes is computationally prohibitive.

One set of techniques proposed, which was able to identify structurally different communities, is to use some measure of “local optimality”. These methods initially identify some “seed” communities and then extend or reduce them to optimize an in-community local density function. The seed groups are considered communities when a single vertex addition or removal does not change the quality of the cluster with respect to the density function greatly. The literature presents several algorithms based on using a density function to identify communities. The density function is formed in such a way that it is the ratio of the number of edges within a community to the total number of all edges connected to the community.

One popular technique is the “Iterative Scan” (IS) method. IS method performs a set of repeated “scans”, each scan starting with an initial set developed by the previous scan (a “seed-set” for the first iteration). It examines each vertex of the network once, adding or removing it if it increases the current density of the set. The scans are repeated until the set is locally optimal with respect to a defined density metric.

Another popular class of techniques uses the concept of graph partitioning and extends them to community detection. The most commonly used technique is “Minimum Cut” method. The intuition is that most interactions are within group whereas interactions between groups are few in number. Thus, we can say that community detection leads to the minimum cut problem. A cut is a partition of vertices of a graph into two disjoint sets. The minimum cut method attempts to find a graph partition such that the number of edges between the two sets is minimized. This naturally leads to discovery of communities. For a comprehensive survey of community detection techniques the reader is referred to the literature.

11.7 SimRank

When the social network graph consists of nodes of several types we use a technique called “SimRank” to analyze them. SimRank is a measure of node similarity between nodes of the same type. For example, in a movie database network consisting of nodes representing movie, actors, etc., SimRank can measure similarity between sets of actors, sets of movies and the like. It uses an algorithm simulating random surfers similar to the computation of PageRank. Computational costs limit the sizes of graphs that can be analyzed using SimRank.

11.7.1 SimRank Implementation

SimRank is a general similarity measure, based on a graph-theoretic model. SimRank is applicable in any domain where relationships between objects are represented, like say a social network. This measure gives a numeric value to similarity of the structural context in which objects occur, based on their relationships with other objects. Effectively, SimRank is a measure that says “two objects are considered to be similar if they are referenced by similar objects”.

For our tri-partite graph of Fig. 11.4 which we shall reproduce here as Fig. 11.10, we can say that if two actors are acting in similar movies then they are similar actors.

Consider the notion of a “random walker” similar to the PageRank computation. A walker at a node N of an undirected graph will move with equal probability to any of vertices which are adjacent to N . So if the walker starts at say X_1 then he can reach A or C or P_1 . If he reaches A , he can reach X_1 again or P_1 or P_2 or X_2 . If he reaches C then he can reach X_1 , X_2 or P_1 . Thus from X_1 there is higher chance of reaching X_2 rather than X_3 . Thus we can infer X_1 and X_2 are similar.

This is indeed logical if we assume X_1 and X_2 to be movies and A and C to be actors. So X_1 and X_2 have same set of actors and thus they are similar.

The conditions for the computation are similar to that of PageRank and thus we can use the same iterative procedure used before to compute SimRank. We can extend this to the teleportation case and induce a random teleportation factor to the computation that enables a random walker to arbitrarily reach any node in the graph. For more details the user is referred to the references.

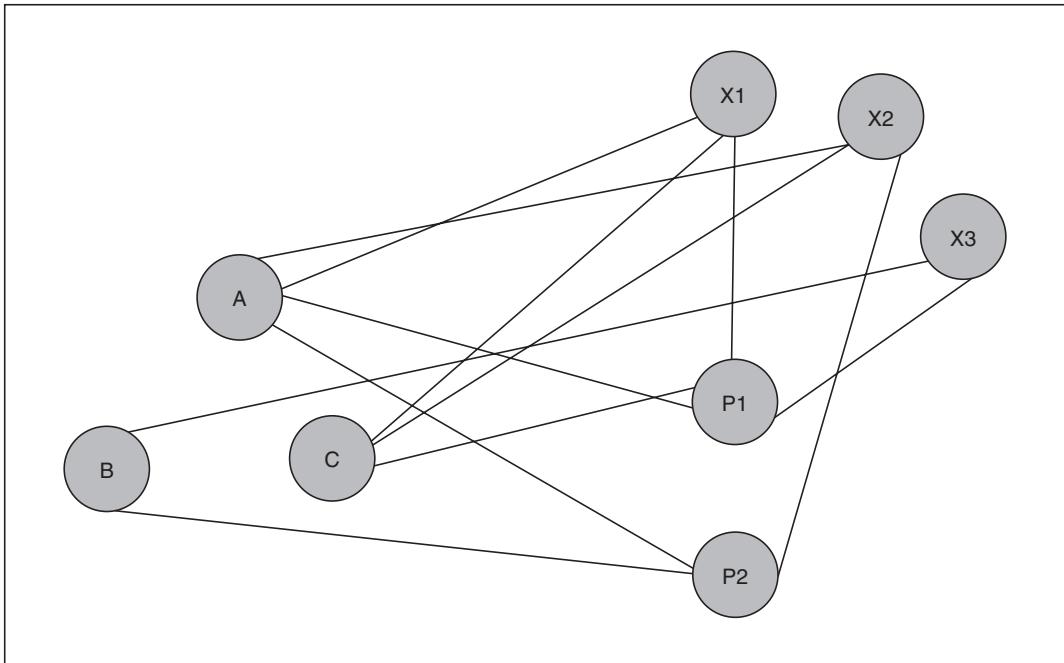


Figure 11.10 Sample social graph for SimRank.

11.8 Counting Triangles in a Social Graph

One important analysis that is particularly useful in the social network context is identifying small communities and counting their occurrence. This basically amounts to finding small connected sub-graphs in the social graph. The most important such sub-graph is the triangle (3-clique). A triangle is the most commonly occurring pattern found in social network graphs. This could be due to two properties that are exhibited by most social graphs: “*homophily*” which is the tendency of individuals to associate and form groups with similar others, and “*transitivity*” which talks of the transitive nature of relationships in a social network. It says if *A* is associated (friend) to *B* and *B* is associated with *C*, then *A* and *C* will also become associates (friends).

The most basic structure in a graph with the above two properties is a triangle or a 3-clique. Thus, counting triangles is an important aspect in any social network analysis application. In this section we present a few motivating examples for counting triangles. We follow it up with an overview of existing triangle counting techniques. As we shall see triangle counting is a computationally expensive task. We give some thoughts on how these algorithms could be set up in a MapReduce framework.

11.8.1 Why Should We Count Triangles?

An important measure one is interested in a social graph is given a node “what is its clustering coefficient?”. The clustering coefficient measures the degree to which a node’s neighbors are themselves neighbors. A node’s *clustering coefficient* is the ratio of the number of closed triplets in the node’s neighborhood over the total number of triplets in the neighborhood.

There are two main motivating reasons for why clustering coefficients are important for graph analysis: A high clustering coefficient indicates very closely knit communities, that is a group of entities where most are of the form A associated with B and C implies B and C are also associated. Such communities are interesting for many applications like target marketing, social recommendations, etc. In a complementary way, a low clustering coefficient can signal a structure called as a “structural hole”, a vertex that is well connected to many communities that are not otherwise connected to each other. Such vertices can act as bridges between communities and are useful for applications that need to identify influential nodes and also where information propagation is needed.

11.8.2 Triangle Counting Algorithms

The brute force method and obvious way to count triangles in a graph is simply checking every group of three vertices and check if they form a triangle or not. For implementing this algorithm, we need to be able to answer queries of the form “given a pair $u; v$ of vertices, is (u, v) an edge?” This algorithm will take $O(n^3)$ time where n is the number of vertices in the graph. The major disadvantage of this method in addition to its high computation cost is that it takes the same amount of time to identify that a graph does not possess a triangle as it takes to find triangles.

A smarter approach is to first list all two-edge paths that are formed in the graph. Such paths of size 2 are called “wedges”. Thus instead of attempting to find all vertex triples, we only attempt to check vertex triples in which we already know that two edges are present. For every edge (x, y) in the graph, check if (x, y, z) forms a triangle and if so add one to the count of triangles. This process has to be repeated for every vertex z distinct from vertices x and y in the graph.

The analysis of the running time is straightforward, as the algorithm runs in time which is $O\left(\sum_{v \in V} \text{degree}_v^2\right)$. It can be proved that for graphs of constant degree this is a linear time algorithm. But the existence of even one high degree vertex will make this algorithm quadratic. Since it is highly likely to come across such high degree nodes in massive social graphs, this algorithm too is not practical. Further, this algorithm counts each triangle $\{x, y, z\}$ six times (once each as $\{x, y, z\}$, $\{x, z, y\}$, $\{y, x, z\}$, $\{y, z, x\}$, $\{z, x, y\}$ and $\{z, y, x\}$).

One optimization is to count each triangle only once. We can use another trick for further making the algorithm efficient. The key idea to use is that “only the lowest-degree vertex of a triangle is responsible for counting it”. Further, we also use an ordering of vertices which starts from the most likely vertices to form triangles to the ones with least probability.

The algorithm can be described as follows: Identify “Massive” vertices in a graph. Let the social graph have n vertices and m edges. We call a vertex massive if its degree is at least \sqrt{m} . If a triangle

has all its vertices as massive, then it is a massive triangle. The algorithm identifies massive triangles and non-massive triangles separately. We can easily see that the maximum number of massive vertices a graph can have is $2\sqrt{m}$. Now represent a graph as a list of its m edges.

1. Compute the degree of each vertex. Examine each edge and add 1 to the count of each of its two end vertices. The total time required is $O(m)$.
2. Create an index on edges using a hash table, with its vertex pair as a key. So we can check whether an edge exists given a pair of vertices in constant $O(1)$ time.
3. Create one more hash table for the edges key being a single vertex. Given a vertex, we can easily identify all vertices adjacent to it.
4. Number the vertices and order them in the ascending order of their degree. Lower degree vertices first followed by higher degree vertices. If two vertices have same degree then order them based on their number.

To find massive triangles we notice the following. There can be only $O(\sqrt{m})$ massive vertices and so if we check all possible three subsets of these set of vertices, we have $O(m^{3/2})$ possible massive triangles. To check the existence of all three edges we need only constant time because of the hash tables. Thus this implementation is only $O(m^{3/2})$.

To find the non-massive triangles we adopt the following procedure:

1. Consider each edge (v_1, v_2) . Ignore this edge when both v_1 and v_2 are massive.
2. Suppose v_1 is not massive and v_1 appears before v_2 in the vertex ordering. We enumerate all vertices adjacent to v_1 and call them u_1, u_2, \dots, u_k , k will be less than \sqrt{m} . It is easy to find all the u 's because of the second hash table which has single vertices as keys ($O(1)$ time).
3. Now for each u_i we check if edge (u_i, v_2) exists and if v_1 appears before u_i in the list. We count this triangle. (Thus we avoid counting the same triangle several times.) This operation is benefited because of the first hash table with key being edges.

Thus, the time to process all the nodes adjacent to v_1 is $O(\sqrt{m})$. Since there are m edges, the total time spent counting other triangles is $O(m^{3/2})$.

11.8.3 Counting Triangles Using MapReduce

For massive social graphs the methods described above will still be too expensive. This motivates us to find a parallel implementation of the triangle counting algorithms. We present a brief outline of how the MapReduce paradigm can be used for the triangle counting exercise. It is left to the reader to fill up the finer details of the MR implementation.

Let G be a social graph. We number the set of vertices V of a G as $1, 2, \dots, n$. Then we can construct a relation E on $(V \times V)$ to represent edges. To avoid representing each edge twice, we declare that $E(A, B)$ is a tuple of this relation if an edge exists between A and B and as integers, we have $A < B$.

Now the triangles are nothing but the tuples in the three-way join of E , which we can represent as

$$E(X, Y) \bowtie E(X, Z) \bowtie E(Y, Z)$$

The equation above represents a three-way join of the relation E with itself. We have to give different names to the elements of the three relations. Convince yourself that each triangle appears only once in this join. The triangle consisting of nodes A , B and C can be generated only when $X < Y < Z$. This is because if $A < B < C$, then X can only be A , Y is B and Z is C .

The Map tasks divide the relation E into as many parts as there are the Map tasks. Suppose one Map task is given the tuple $E(u, v)$ to send to certain Reduce tasks. Let (u, v) be a tuple of the join term $E(X, Y)$. We can hash u and v to get the bucket numbers for X and Y , but we do not know the bucket to which Z hashes. Notice that if we hash vertices to b buckets, then there will be b^3 reducers, each one associated with one sequence of three bucket numbers (x, y, z) . Thus, we must send $E(u, v)$ to all Reducer tasks that correspond to a sequence of three bucket numbers, $(h(u), h(v), z)$, where $h(x)$ indicates the hashing function. Similarly, we also have to send it to all reducers corresponding to $(h(u), y, h(v))$ for any y , and also to all reducers corresponding to a triple $(x, h(u), h(v))$ for any x .

The total communication required is thus $3b$ key-value pairs for each of the m tuples of the edge relation E . So if the communication cost is $O(mb)$, each of the b^3 reducers will receive about $O(m/b^2)$ edges. Using the analysis of the previous section, the total computation at each reducer will be $O((m/b^2)^{3/2}$ or $O(m^{3/2}/b^3)$. For b^3 Reducers the total cost is $O(m^{3/2})$.

Further optimizations to these algorithms are possible. There are also several graph partition algorithms available for counting triangles efficiently.

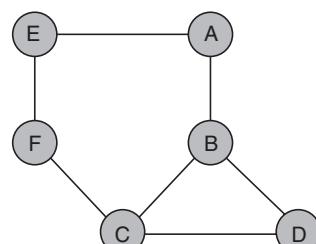
Summary

- The availability of massive quantities of data from very large social networks like blogs, social networking sites, newsgroups, chat rooms, etc. motivate their analysis. Mining these structures for patterns yields knowledge that can effectively be used for predictions and decision-making.
- When the objective is to mine a social network for patterns, a natural way to represent a social network is by a graph. The graph is typically very large, with nodes corresponding to objects and edges corresponding to links representing relationships or interactions between objects. One node indicates one person or a group of persons.
- A social network can be considered as a heterogeneous and multi-relational dataset represented by a graph. Both nodes and edges can have attributes. Objects may have class labels. There is at least one relationship between entities of the network.
- Social networks exhibit the property of non-randomness often called locality. Locality is the property of social networks that says nodes and edges of the graph tend to cluster in communities.

- Social networks can be of different types like:
 - **Collaboration Graphs:** Collaboration graphs display interactions which can be termed collaborations in a specific setting.
 - **Who-Talks-to-Whom Graphs:** These model a type of social network also called “Communication” network.
 - **Information Linkage Graphs:** Snapshots of the Web are central examples of network datasets; nodes are Web pages and directed edges represent links from one page to another.
 - **Heterogeneous Social Networks:** A social network may have heterogeneous nodes and links. This means the network may consist of different types of entities (multi-mode network) and relationships (multi-relational network).
- The discovery of “communities” in a social network is one fundamental requirement in several social network applications. Typically in a social network we can define a “community” as a collection of individuals with dense relationship patterns within a group and sparse links outside the group.
- In a social network, individuals (nodes) normally belong to several communities, and thus standard clustering techniques with traditional distance measures are not sufficient for community detection.
- One way to separate nodes into communities is to measure the betweenness of edges, which is the sum over all pairs of nodes of the fraction of shortest paths between those nodes that go through the given edge. Communities are formed by deleting the edges whose betweenness is above a given threshold. Girvan-Newman algorithm is one popular community detection algorithm based on this concept.
- Several graph-based models like finding k -cliques are also useful in direct discovery of communities.
- When the social network graph consists of nodes of several type we use a technique called “SimRank” to analyze them. SimRank is a measure of node similarity between nodes of the same type. It uses an algorithm simulating random surfers similar to the computation of PageRank. Computational costs limit the sizes of graphs that can be analyzed using SimRank.
- The number of triangles in a social graph is a useful measure of the closeness of a community in the network. We can list and count the number of triangles in a social graph using an algorithm that uses $O(m^{3/2})$ time. The MapReduce version can also be used.

Exercises

1. Identify five social networks in popular use. What type of social network do they represent? Attempt to capture their essence using a social graph.
2. For the graph given below use betweenness factor and find all communities.



3. For graph of Fig. 11.6 show how the clique percolation method will find cliques.
4. Try and describe a procedure to compute SimRank using some example social graph.
5. Using the algorithms for counting triangles find triangles for the different social graphs depicted in the chapter.

Programming Assignments

1. Implement Girvan–Newman Algorithm on any social graph using MapReduce.
2. Use the above code and implement a community detection algorithm on any dataset from the SNAP dataset (link given in the dataset list in the Appendix).
3. Implement the Clique Percolation method on the same dataset to find communities.
4. Implement triangle counting algorithm using MapReduce on any large social graph.

References

1. S. Kelley, M.K. Goldberg, K. Mertsalov, M. Magdon-Ismail, W.A. Wallace (2011). Overlapping Communities in Social Networks. *IJSCCPs*, 1(2): 135–159.
2. G. Palla, I. Derenyi, I. Farkas *et al.* (2005). Uncovering the Overlapping Community Structure of Complex Networks in Nature and Society. *Nature*, 435:814.
3. Stanford Network Analysis Platform, <http://snap.stanford.edu>.
4. S. Suri and S. Vassilivitskii (2011). Counting Triangles and the Curse of the Last Reducer. *Proc. WWW Conference*.
5. H. Tong, C. Faloutsos, J.-Y. Pan (2006). Fast Random Walk with Restart and its Applications. *ICDM*, pp. 613–622.
6. C.E. Tsourakakis, U. Kang, G.L. Miller *et al.* (2009). DOULION: Counting Triangles in Massive Graphs with a Coin. *Proc. Fifteenth ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*.
7. D. Easley and J. Kleinberg (2010). *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Cambridge University Press, New York, NY. Complete preprint on-line at <http://www.cs.cornell.edu/home/kleinber/networks-book/>
8. A. Rajaraman and J. D. Ullman (2011). *Mining of Massive Datasets*. Cambridge University Press, New York, NY.
9. D. Cai, Z. Shao, X. He *et al.* (2005). *Mining Hidden Communities in Heterogeneous Social Networks, LinkKDD*.
10. S. Parthasarathy, Y. Ruan, V Satuluri (2011). Community Discovery in Social Networks: Applications, Methods and Emerging Trends. *Social Network Data Analytics*. Springer US, 79–113.
11. S. Fortunato (2010). Community Detection in Graphs. *Phys. Rep.*, 486:75–174.

Appendix

Some Standard Useful Datasets Available on the Web

Please be sure to acknowledge the websites when you use data from these websites:

1. <http://law.di.unimi.it/datasets.php> – This website provides numerous datasets and useful open source software. It is created and maintained by The Laboratory for Web Algorithmics (LAW) Computer Science Department of the Università degli studi di Milano.
2. http://www.yelp.com/dataset_challenge – This dataset maintained by Yelp gives information about 1.6M reviews and 500K tips by 366K users for 61K businesses. The dataset contains a Social network of 366K users for a total of 2.9M social edges.
3. <http://snap.stanford.edu/data/index.html> – Stanford Large Network Dataset Collection, maintained by Stanford Network Analysis Project contains numerous graph datasets that can be used for various mining applications.
4. <http://grouplens.org/datasets/movielens> – GroupLens Research, an analytics organization, has collected and made available rating data sets from the MovieLens website. These datasets were collected over various periods of time (<http://movielens.org>), depending on the size of the set. Before using these data sets, please review usage details at the MovieLens website.
5. www.kaggle.com – Kaggle is an organization of data scientists from top universities and enterprises all over the world. Kaggle is dedicated to the dissemination of analytics knowledge through competitions. They make available large datasets for non-commercial purposes.
6. <http://www.kdnuggets.com/datasets/index.html> – KD Nuggets is a one stop portal for anything related to Data Mining, Analytics, Big Data, and Data Science. They have links to several software and data sources for big data.
7. The following are links to public and open big datasets
 - (a) <http://usgovxml.com>
 - (b) <http://aws.amazon.com/datasets>
 - (c) <http://databib.org>

- (d) <http://datacite.org>
- (e) <http://figshare.com>
- (f) <http://linkeddata.org>
- (g) <http://reddit.com/r/datasets>
- (h) <http://thewebminer.com/>
- (i) <http://thedatahub.org> alias <http://ckan.net>
- (j) <http://quandl.com>
- (k) <http://enigma.io>
- (l) <http://www.ufindthem.com/>

Index

A

ACID (Atomicity, Consistency, Isolation and Durability) properties, 6
ad-hoc queries, 133
Ad servers/Ad serving, 48
advertiser keyword suggestions, 105
AllegroGraph, 49
ALTER TABLE command, 40
Amazon.com, 112
Amazon DynamoDB, 42–43, 46
Amazon Redshift Integration, 43
Amazon S3 (simple storage service), 46
analytics of NoSQL, 42
Apache Cassandra, 41, 46–47
Apache Software Foundation (ASF), 11
Apache Spark, 5
Apache Sqoop. *See* Sqoop (“SQL-to-Hadoop”)
Apache ZooKeeper. *See* ZooKeeper
ApplicationMaster (AM) per application, 18
Apriori algorithm, 196, 211–215
association rule mining, 199–204
atomicity, consistency, isolation, durability (ACID) properties, 38, 41, 53
authorities, 183
automated storage scaling, 43
auto-sharding, 57
AWS Identity and Access Management, 43
AWS Management Console, 43
Azure Table Storage (ATS), 46

B

bags, 20
biased reservoir sampling, 137
big data
applications for text based similarity of, 110–111
case study of big data solutions, 7–8
characteristics of, 2–3
components of, 3

defined, 1
enterprise architecture for, 41
scalability of, 5
storage requirements of, 5
technologies available for, 6
type of, 3–4
volume of data, 2–3
Big Data Analytics, 1
advantages of, 5–6
cloud computing infrastructure, 22
major trends in computing, 2
big data approach, 5
BigTable, 43
Binary JSON (BSON) format, 49
blocking query operator, 135–136
blocks, 71
Bloom, Burton Howard, 139
Bloom filter, 139–142
Brewer’s theorem. *See* Consistency, Availability, Partition tolerance (CAP) theorem
built-in fault tolerance, 43
business intelligence of NoSQL, 42

C
C++, 23
Cafarella, Mike, 11
catalog management, 52
Chubby Lock Service, 43
chunk server, 71
Clickstream Analytics, 7–8
Clique Percolation Method (CPM), 291–292
cloaking, 160–161
cloud computing, 2
cluster computing, 70
clustering
applications of, 242–243
basics of, 239–241
curse of dimensionality, 244–245
definition of, 239
hierarchical, 245–248
partition-based, 248–254
of social graphs, 285–290
strategies of, 243–244
streams, 258–261
Clustering Using Representatives (CURE) algorithm, 254–258
collaboration graphs, 284
collaborative filtering, 105, 110, 266–269
as a similar-sets problem, 112–114
columnar databases, 4
column family store/wide column store, 46–48
combiners, 176
communities in a social graph, 290–294
concept drift, 130
concise sampling, 137–138
Consistency, Availability, Partition tolerance (CAP) theorem, 38
consistent hashing, 55–56
consumer life of data per day, 1
content-based image retrieval, 109
content-based recommendations, 269–275
content management, 52
continuous queries, 132
correlated aggregate queries, 134
Cosine Distance, 120–121
CouchBase, 48
CouchDB, 48
count-distinct problem, 143
CRM (Customer Relationship Management) applications, 2
cross-document co-reference resolution, 112
Cutting, Doug, 11

D
data
extracted, transformed and loaded (ETL process), 3
marts, 4
migration of, 3–4
speed of, 4
warehouses, 4

database management system
 (DBMS), 5, 127
 data stream model, 128–129
 data mining algorithms, 3
 DataNode, 70–73
 DataNodes, 15
 Datar–Gionis–Indyk–Motwani
 algorithm, 147–152
 data sources for business needs, 1
 data stream mining, 130–131
 data streams, 4, 127
 applications, 128, 131–132
 in batch processing, 135
 counting distinct elements in,
 143–146
 counting frequent items in, 231–234
 filtering, 138–142
 financial applications, 132
 issues in query processing, 133–136
 network traffic analysis using,
 131–132
 querying on windows, 146–152
 sampling in, 135–138
 sliding windows of data, 134
 space requirements, 146
 stream queries, 132–133
 synopsis or sketch of data, 135
 unbounded memory requirements,
 133–134
 dead ends, 168, 170
 360-degree view of customers, 52
 DELETE statement, 40
 Directed Acyclic Graph (DAG), 20
 disjoint communities, 290
 distributed file systems (DFS), 70–71
 document clustering, 112
 document path, 48
 Document similarity, 110
 document similarity, 110–112
 document store, 48–49
 “Doorway” pages, 161
 downward-closure property, 210

E

E-commerce, 113–114
 Edit Distance, 122
 Euclidean Distance, 118–119
 Euclidean spaces, 118
 extracted, transformed and loaded
 (ETL process), 3

F

Facebook, 1–2, 49, 284
 Flajolet–Martin (FM) algorithm,
 143–146

Flickr, 49
 fraud detection, 53
 frequent-itemset mining, 196–197,
 204–211

in decaying window, 233–234
 full-text index search engines, 160

G

Girvan–Newman algorithm, 289–290
 global Resource Manager (RM), 18
 Google, 1, 6
 Google+, 49
 Google BigTable, 19, 37, 43
 Google Chrome, 139
 Google File System, 43
 Google file system (GFS), 71
 Google News, 112
 graph store, 49–50

H

Hadoop, 5, 11, 55, 72
 assumptions, 13
 common packages for, 14
 compatible file system of, 22
 components, 13–18
 defined, 11–12
 degree of fault tolerance, 12
 economics of, 13
 goals of, 12–13
 handling of hardware failures, 13
 latency of, 13
 limitations, 23–24
 in Linux and other Unix systems, 23
 network bandwidth and storage, 72
 physical architecture, 21–23
 potential stability issues, 24
 programming languages, 24
 purpose of, 12
 scalability of, 12
 security concerns, 23
 small data and, 24
 in virtualized environments, 21

Hadoop cluster, 21
 Hadoop-compatible file system, 21
 Hadoop database (HBase), 6
 Hadoop Distributed File System
 (HDFS), 13, 19, 39, 41, 71
 advantage of using, 23
 components of, 14–16
 creation of replicas, 14
 DataNodes, 15
 file system, 23
 NameNode, 14–15
 non-POSIX operations, 23
 replication of data, 21

storing of files, 23
 Hadoop distributed file system
 (HDFS), 6

Hadoop ecosystem, 18–21
 Hamming Distance, 122–123
 hashing, 55–56
 Haveliwala, Taher H., 177
 HBase, 18–19, 39
 HCatalog, 19–20
 heterogeneous social networks, 284–285
 Hewlett-Packard, 112
 hierarchical clustering, 245–248
 Hive, 18–20
 HiveQL, 6, 19–20
 Horowitz, Eliot, 44
 HTTP, 23
 hubs, 183
 hyperlink-induced topic search
 (HITS), 183–189
 hypervisor layer, 71

I

IBM and W3C consortiums, 3
 information linkage graphs, 284
 InputFormat, 79
 InputSplit, 79
 INSERT statement, 40
 Internet of Things (IoT), 52
 item-based collaborative filtering, 114
 item-based recommenders, 114
 “Iterative Scan” (IS) method, 293
 iThenticate, 111

J

Jaccard Distance, 120
 Jaccard Similarity, 107–108, 114–115,
 120
 Java archive (JAR) files and scripts, 14
 Java Hibernate, 40
 Java map-reduce, 20
 Java programs and shell scripts, 19–20,
 23

Java Runtime Environment (JRE)
 1.6, 14
 JobClient, 80
 JobHistoryServer, 17
 JobScheduler, 15
 job scheduling and monitoring, 18
 JobTracker, 15–17, 70, 78, 80
 JSON (JavaScript Object Notation)
 format, 20, 44, 48

K

keyspace, 46, 48
 key-value store, 42–43

- dynamic component of, 46
examples, 46
static component of, 46
uses of, 46
weakness of, 45
K-means algorithm, 249–254
- L**
LinkedIn, 37, 49, 284
Link Spam, 160, 179–183
Linux, 23
literals, 200
 L^∞ -norm, 119
 L_1 -norm, 118
 L_2 -norm, 118
 L_r norm, 118–119
- M**
Mahout, 6, 19–21
main memory
 counting, 206–210
 handling larger datasets, 215–224
Manhattan Distance, 118
map process, 73
MapReduce, 6, 13–14, 37, 39, 43
 algorithms using, 81–91
 combiners, 76–77
 components of, 17
 coping with node failures, 80
 difference operation, 86–87
 distributed cache, 80
 distributed file systems (DFS), 70–71
 drivers of, 79
 execution pipeline, 79–80
 grouping and aggregation by, 88–89
 grouping by key, 76
 HDFS and, 17
 intersection operation, 85–86
 JobHistoryServer, 17
 job structure, 90–91
 JobTracker, 17
 mappers of, 79
 map phase of, 16
 map process, 73
 map tasks, 76
 matrix multiplication of large
 matrices, 89–90
 matrix-vector multiplication by, 82
 natural join, computing, 87–88
 output of Reduce task, 76
 PageRank implementation using, 174
 physical organization of compute
 nodes, 71–75
 process pipeline, 80
 projection, computing, 84
reduce phase of, 16
reduce process, 74
reducers of, 79
relational operators and, 83
retrieve the output of a job, 73
run-time coordination in, 78–80
Savasere, Omiecienski and Navathe
 (SON) Algorithm and, 228
scheduling, monitoring and rescheduling of failed tasks, 78
selections, computing, 83–84
shuffling and sorting, 79
tasks of, 17
TaskTrackers, 17
union operation, 85
word count using, 77
- MapReduce 1.0, 18
“marker-basket” model of data, 195–204
MasterNode, 69
master–slave replication, 54–55
MemcacheDB, 37, 46
memory bandwidth, 13
metadata information, 71
migration of data, 3–4
Minkowski measure, 118
min sup, 201
mobile applications of NoSQL, 52
mobile computing, 2
modern day transaction analysis, 41
MongoDB, 44, 48–49
MongoDb, 41
MovieLens, 115
Multihash algorithm, 223–224
Multistage algorithm, 221–223
MySql, 44
- N**
NameNode, 14–16, 21, 23, 70–73
Nearest Neighbor (NN) Search,
 106–110
 common applications of, 109–110
 problem formulation, 107–108
nearest-neighbor technique, 266–267
Neo4j, 41, 44
 features of, 50
 relationships in, 49
NetFlix, 37, 115
network traffic analysis, 131–132
news aggregators, 112
NodeManager, 18
NoSQL
 agility of, 40
 analytics and business intelligence, 42
 business drivers, 38–42
 case studies, 42–44
catalog management, 52
companies using, 37
consistent hashing data on a
 cluster, 55–56
content management, 52
data architectural patterns, 45–50
data availability of, 40
database environments, 40
dealing with online queries, 39
360-degree view of customers, 52
distribution models, 54–55
distribution of queries to
 DataNodes, 57
enterprise architecture for big data,
 41
fixed schemas or JOIN operations
 of, 37
fraud detection, 53
handling and managing big data,
 51–57
location transparency or location
 independence of, 40
master–slave replication in, 54–55
mobile applications, 52
modern day transaction analysis
 in, 41
on-board GPU memory, 39
overview, 37–38
peer-to-peer replication in, 54–55
querying large datasets, 57
rationale for, 38
reasons for developing, 37
replication of data, 56–57
scale of data sources, 41
scale-out architecture of, 57
schema-free flexible data model of,
 40–41
shared RAM, 54
speed, 41
storage models, 41
variability in processing, 40
variations of architectural patterns,
 50
velocity of processing, 40–41
volume of processing, 40
Not only SQL, 37. *See also* NoSQL
Nutch, 11
- O**
one-time queries, 132
on-line retailers, 113–114
Oozie, 19–20
open-source software framework for
 storing and processing big data.
 See Hadoop

open-source web search engine, 11
 Optical Character Recognition (OCR), 109
 Oracle, 44
 overlapping communities, 290

P

PageRank, 162–173
 avoiding spider traps, 171–172
 computation, 164–166, 173–176
 dealing with dead ends, 170
 definition, 163–164
 modified, 169–172
 representation of transition matrices, 173
 in search engine, 172–173
 structure of web and, 167–169
 topic-sensitive, 176–179
 Park–Chen–Yu (PCY) algorithm, 216–221
 Pearson Correlation Coefficient, 267
 peer-to-peer replication, 54–55
 personal computers (PCs), data storage capacity, 1
 PHP, 23
 Pig, 19–20
 Pig Latin, 20
 Pinterest, 2
 plagiarism detection, 111
 pre-defined query, 133
 primary NameNode, 23
 Property Graph Model, 44
 Pythagoras theorem, 118
 Python, 23

R

randomized sampling algorithm, 224–226
 RDF data model, 4
 real-time processing, 4
 rebalancing, 57
 recommender system
 application of, 265
 commonly seen, 265
 content-based, 269–275
 model, 266
 use of, 265
 RecordReader, 79
 RecordWriter, 79
 Redis, 46
 reduce process, 74
 relational database management systems (RDBMS), 5, 38–39, 55
 relational databases, 37, 44, 52

replication of data, 54, 56–57
 reservoir sampling, 136–137
 resource consumption monitoring, 43
 resource management, 18
 REST APIs, 19
 Riak, 46
 Ruby, 23

S

Savasere, Omiecinski and Navathe (SON) Algorithm, 226–228
 scalability of big data, 5, 43
 Scheduler, 43
 schema-free NoSQL data model, 40–41
 Search Engine Optimization (SEO), 160
 secondary NameNode, 16, 21, 23, 71–73
 Secure Shell (Ssh), 14
 SELECT statement, 40
 sensor networks, 131
 server virtualization, 71
 sharding, 54, 57
 “shared-nothing” concept, 57
 similarity

 applications for text based similarity of big data, 110–111
 distance measures and, 116–123
 of documents, 110–112
 user ratings and, 115
 SimpleDB, 37
 SimRank, 294–295
 Single Point of Failure (SPOF), 18
 single point of failure (SPOF), 54–55
 singleton, 138
 SlaveNode, 70
 sliding windows of data, 134
 smart-city, 7
 social graphs

 clustering of, 285–290
 counting triangles in, 295–298
 social network mining
 applications of, 280
 graph of, 280–283
 social networks, types of, 283–285
 spam, 160–162
 Spam farm, 180–182
 spam mass, 183
 spammers, 160
 speed of data, 4
 spider traps, 168, 171–172
 Sqoop (“SQL-to-Hadoop”), 19–20
 Strongly Connected Component (SCC), 167
 synopsis or sketch of data, 135

T

TaskTrackers, 16–17, 70, 78, 80
 taxonomies-based search engines, 160
 tendrils, 167
 TeradataAster, 49
 threshold parameter, 138
 Thrift, 19
 Toivonen’s algorithm, 229–231
 Topic-Sensitive PageRank (TSPR), 176–179
 traditional data management, 4
 transaction log analysis, 132
 TripAdvisor, 115
 TrustRank, 160, 183
 tubes, 167
 Turnitin, 111
 Twitter, 37, 49, 284

U

Unix systems, 23
 UPDATE SQL statement, 40
 user-based recommenders, 114
 user-defined functions (UDFs), 19
 user ratings, 115
 Userspace (FUSE) virtual file system, 23

V

value, 3
 variety, 2–3
 velocity, 2–3, 40–41, 130
 veracity, 3
 volatility, 130
 volume, 2–3, 130

W

web search, 105
 Web search engines, 159–162
 who-talks-to-whom graphs, 284

X

XML data model, 4, 20

Y

Yahoo, 11
 Yelp, 115
 Yet Another Resource Negotiator (YARN), 14
 function of, 18
 problems addressed, 18
 YouTube, 49

Z

znodes, 21
 ZooKeeper, 19, 21

About the Book

The goal of this book is to cover foundational techniques and tools required for *Big Data Analytics*. It focuses on concepts, principles, and techniques applicable to any technology environment and industry and establishes a baseline that can be enhanced further by additional real-world experience. This book aims to be a ready reckoner to either a novice or a professional working in the field.

Topics covered include Hadoop, MapReduce, Association Rules, Large-Scale Supervised Machine Learning, Data Streams, Clustering, NoSQL systems (Pig, Hive), and Applications including Recommendation Systems, Web and Security.

Precise Series

- Precise contents as per syllabi.
- Excellent presentation in a clear, logical and concise manner.
- Learning objectives at the beginning of each chapter.
- Focus on explanation of concepts.
- Sufficient examples for easy comprehension.
- Important points and formulas at the end of each chapter to quickly review the concepts.
- Optimum balance of qualitative and quantitative problem sets.

Highlights of the Book

- Fills the gap in the literature available on *Big Data Analytics*.
- Combines *Big Data Management and Analytics* in the same book.
- Provides overview of tools available for BDA: *NOSQL, MapReduce, Hadoop*.
- Covers many real-life examples.
- Provides links to several real-life datasets in the Appendix.
- Includes comprehensive *Lab Manual*.

follow us on



facebook.com/wileyindia



twitter.com/wileyindiapl



linkedin.com/in/wileyindia



google.com/+wileyindia

READER LEVEL
Undergraduate/Graduate

SHELVING CATEGORY
Engineering

WILEY

Wiley India Pvt. Ltd.

4435-36/7, Ansari Road, Daryaganj
New Delhi-110 002
Customer Care +91 11 43630000
Fax +91 11 23275895
csupport@wiley.com
www.wileyindia.com

ISBN: 978-81-265-5865-0

