Third Edition

DATA MINING

Concepts and Techniques

Jiawei Han | Micheline Kamber | Jian Pei

MORGAN KAUFMANN

Data Mining

Third Edition

The Morgan Kaufmann Series in Data Management Systems (Selected Titles)

Joe Celko's Data, Measurements, and Standards in SQL
Joe Celko

Information Modeling and Relational Databases, 2*nd Edition*
Terry Halpin, Tony Morgan

Joe Celko's Thinking in Sets
Joe Celko

Business Metadata
Bill Inmon, Bonnie O'Neil, Lowell Fryman

Unleashing Web 2.0
Gottfried Vossen, Stephan Hagemann

Enterprise Knowledge Management
David Loshin

The Practitioner's Guide to Data Quality Improvement
David Loshin

Business Process Change, 2*nd Edition*
Paul Harmon

IT Manager's Handbook, 2*nd Edition*
Bill Holtsnider, Brian Jaffe

Joe Celko's Puzzles and Answers, 2*nd Edition*
Joe Celko

Architecture and Patterns for IT Service Management, 2*nd Edition, Resource Planning and Governance*
Charles Betz

Joe Celko's Analytics and OLAP in SQL
Joe Celko

Data Preparation for Data Mining Using SAS
Mamdouh Refaat

Querying XML: XQuery, XPath, and SQL/ XML in Context
Jim Melton, Stephen Buxton

Data Mining: Concepts and Techniques, 3*rd Edition*
Jiawei Han, Micheline Kamber, Jian Pei

Database Modeling and Design: Logical Design, 5*th Edition*
Toby J. Teorey, Sam S. Lightstone, Thomas P. Nadeau, H. V. Jagadish

Foundations of Multidimensional and Metric Data Structures
Hanan Samet

Joe Celko's SQL for Smarties: Advanced SQL Programming, 4*th Edition*
Joe Celko

Moving Objects Databases
Ralf Hartmut Güting, Markus Schneider

Joe Celko's SQL Programming Style
Joe Celko

Fuzzy Modeling and Genetic Algorithms for Data Mining and Exploration
Earl Cox

Data Modeling Essentials, 3*rd Edition*
Graeme C. Simsion, Graham C. Witt

Developing High Quality Data Models
Matthew West

Location-Based Services
Jochen Schiller, Agnes Voisard

Managing Time in Relational Databases: How to Design, Update, and Query Temporal Data
Tom Johnston, Randall Weis

Database Modeling with Microsoft® Visio for Enterprise Architects
Terry Halpin, Ken Evans, Patrick Hallock, Bill Maclean

Designing Data-Intensive Web Applications
Stephano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, Maristella Matera

Mining the Web: Discovering Knowledge from Hypertext Data
Soumen Chakrabarti

Advanced SQL: 1999—Understanding Object-Relational and Other Advanced Features
Jim Melton

Database Tuning: Principles, Experiments, and Troubleshooting Techniques
Dennis Shasha, Philippe Bonnet

SQL: 1999—Understanding Relational Language Components
Jim Melton, Alan R. Simon

Information Visualization in Data Mining and Knowledge Discovery
Edited by Usama Fayyad, Georges G. Grinstein, Andreas Wierse

Transactional Information Systems
Gerhard Weikum, Gottfried Vossen

Spatial Databases
Philippe Rigaux, Michel Scholl, and Agnes Voisard

Managing Reference Data in Enterprise Databases
Malcolm Chisholm

Understanding SQL and Java Together
Jim Melton, Andrew Eisenberg

Database: Principles, Programming, and Performance, 2*nd Edition*
Patrick and Elizabeth O'Neil

The Object Data Standard
Edited by R. G. G. Cattell, Douglas Barry

Data on the Web: From Relations to Semistructured Data and XML
Serge Abiteboul, Peter Buneman, Dan Suciu

Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations, 3*rd Edition*
Ian Witten, Eibe Frank, Mark A. Hall

Joe Celko's Data and Databases: Concepts in Practice
Joe Celko

Developing Time-Oriented Database Applications in SQL
Richard T. Snodgrass

Web Farming for the Data Warehouse
Richard D. Hackathorn

Management of Heterogeneous and Autonomous Database Systems
Edited by Ahmed Elmagarmid, Marek Rusinkiewicz, Amit Sheth

Object-Relational DBMSs, 2nd Edition
Michael Stonebraker, Paul Brown, with Dorothy Moore

Universal Database Management: A Guide to Object/Relational Technology
Cynthia Maro Saracco

Readings in Database Systems, 3rd Edition
Edited by Michael Stonebraker, Joseph M. Hellerstein

Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM
Jim Melton

Principles of Multimedia Database Systems
V. S. Subrahmanian

Principles of Database Query Processing for Advanced Applications
Clement T. Yu, Weiyi Meng

Advanced Database Systems
Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, Roberto Zicari

Principles of Transaction Processing, 2nd Edition
Philip A. Bernstein, Eric Newcomer

Using the New DB2: IBM's Object-Relational Database System
Don Chamberlin

Distributed Algorithms
Nancy A. Lynch

Active Database Systems: Triggers and Rules for Advanced Database Processing
Edited by Jennifer Widom, Stefano Ceri

Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach
Michael L. Brodie, Michael Stonebraker

Atomic Transactions
Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete

Query Processing for Advanced Database Systems
Edited by Johann Christoph Freytag, David Maier, Gottfried Vossen

Transaction Processing
Jim Gray, Andreas Reuter

Database Transaction Models for Advanced Applications
Edited by Ahmed K. Elmagarmid

A Guide to Developing Client/Server SQL Applications
Setrag Khoshafian, Arvola Chan, Anna Wong, Harry K. T. Wong

Data Mining
Concepts and Techniques

Third Edition

Jiawei Han
University of Illinois at Urbana–Champaign
Micheline Kamber
Jian Pei
Simon Fraser University

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann is an imprint of Elsevier

Morgan Kaufmann Publishers is an imprint of Elsevier.
225 Wyman Street, Waltham, MA 02451, USA

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data

Han, Jiawei.
Data mining : concepts and techniques / Jiawei Han, Micheline Kamber, Jian Pei. – 3rd ed.
p. cm.
ISBN 978-0-12-381479-1
1. Data mining. I. Kamber, Micheline. II. Pei, Jian. III. Title.
QA76.9.D343H36 2011
006.3′12–dc22 2011010635

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

For information on all Morgan Kaufmann publications, visit our
Web site at *www.mkp.com* or *www.elsevierdirect.com*

Printed in the United States of America
11 12 13 14 15 10 9 8 7 6 5 4 3 2 1

To Y. Dora and Lawrence for your love and encouragement

J.H.

To Erik, Kevan, Kian, and Mikael for your love and inspiration

M.K.

To my wife, Jennifer, and daughter, Jacqueline

J.P.

This page intentionally left blank

Contents

Foreword

Analyzing large amounts of data is a necessity. Even popular science books, like "super crunchers," give compelling cases where large amounts of data yield discoveries and intuitions that surprise even experts. Every enterprise benefits from collecting and analyzing its data: Hospitals can spot trends and anomalies in their patient records, search engines can do better ranking and ad placement, and environmental and public health agencies can spot patterns and abnormalities in their data. The list continues, with cybersecurity and computer network intrusion detection; monitoring of the energy consumption of household appliances; pattern analysis in bioinformatics and pharmaceutical data; financial and business intelligence data; spotting trends in blogs, Twitter, and many more. Storage is inexpensive and getting even less so, as are data sensors. Thus, collecting and storing data is easier than ever before.

The problem then becomes *how to analyze* the data. This is exactly the focus of this Third Edition of the book. Jiawei, Micheline, and Jian give encyclopedic coverage of all the related methods, from the classic topics of clustering and classification, to database methods (e.g., association rules, data cubes) to more recent and advanced topics (e.g., SVD/PCA, wavelets, support vector machines).

The exposition is extremely accessible to beginners and advanced readers alike. The book gives the fundamental material first and the more advanced material in follow-up chapters. It also has numerous rhetorical questions, which I found extremely helpful for maintaining focus.

We have used the first two editions as textbooks in data mining courses at Carnegie Mellon and plan to continue to do so with this Third Edition. The new version has significant additions: Notably, it has more than 100 citations to works from 2006 onward, focusing on more recent material such as graphs and social networks, sensor networks, and outlier detection. This book has a new section for visualization, has expanded outlier detection into a whole chapter, and has separate chapters for advanced

methods—for example, pattern mining with top-k patterns and more and clustering methods with biclustering and graph clustering.

Overall, it is an excellent book on classic and modern data mining methods, and it is ideal not only for teaching but also as a reference book.

Christos Faloutsos
Carnegie Mellon University

Foreword to Second Edition

We are deluged by data—scientific data, medical data, demographic data, financial data, and marketing data. People have no time to look at this data. Human attention has become the precious resource. So, we must find ways to automatically analyze the data, to automatically classify it, to automatically summarize it, to automatically discover and characterize trends in it, and to automatically flag anomalies. This is one of the most active and exciting areas of the database research community. Researchers in areas including statistics, visualization, artificial intelligence, and machine learning are contributing to this field. The breadth of the field makes it difficult to grasp the extraordinary progress over the last few decades.

Six years ago, Jiawei Han's and Micheline Kamber's seminal textbook organized and presented Data Mining. It heralded a golden age of innovation in the field. This revision of their book reflects that progress; more than half of the references and historical notes are to recent work. The field has matured with many new and improved algorithms, and has broadened to include many more datatypes: streams, sequences, graphs, time-series, geospatial, audio, images, and video. We are certainly not at the end of the golden age—indeed research and commercial interest in data mining continues to grow—but we are all fortunate to have this modern compendium.

The book gives quick introductions to database and data mining concepts with particular emphasis on data analysis. It then covers in a chapter-by-chapter tour the concepts and techniques that underlie classification, prediction, association, and clustering. These topics are presented with examples, a tour of the best algorithms for each problem class, and with pragmatic rules of thumb about when to apply each technique. The Socratic presentation style is both very readable and very informative. I certainly learned a lot from reading the first edition and got re-educated and updated in reading the second edition.

Jiawei Han and Micheline Kamber have been leading contributors to data mining research. This is the text they use with their students to bring them up to speed on

the field. The field is evolving very rapidly, but this book is a quick way to learn the basic ideas, and to understand where the field is today. I found it very informative and stimulating, and believe you will too.

Jim Gray
In his memory

Preface

The computerization of our society has substantially enhanced our capabilities for both generating and collecting data from diverse sources. A tremendous amount of data has flooded almost every aspect of our lives. This explosive growth in stored or transient data has generated an urgent need for new techniques and automated tools that can intelligently assist us in transforming the vast amounts of data into useful information and knowledge. This has led to the generation of a promising and flourishing frontier in computer science called *data mining*, and its various applications. Data mining, also popularly referred to as *knowledge discovery from data (KDD)*, is the automated or convenient extraction of patterns representing knowledge implicitly stored or captured in large databases, data warehouses, the Web, other massive information repositories, or data streams.

This book explores the concepts and techniques of *knowledge discovery* and *data mining*. As a multidisciplinary field, data mining draws on work from areas including statistics, machine learning, pattern recognition, database technology, information retrieval, network science, knowledge-based systems, artificial intelligence, high-performance computing, and data visualization. We focus on issues relating to the feasibility, usefulness, effectiveness, and scalability of techniques for the discovery of patterns hidden in *large data sets*. As a result, this book is not intended as an introduction to statistics, machine learning, database systems, or other such areas, although we do provide some background knowledge to facilitate the reader's comprehension of their respective roles in data mining. Rather, the book is a comprehensive introduction to data mining. It is useful for computing science students, application developers, and business professionals, as well as researchers involved in any of the disciplines previously listed.

Data mining emerged during the late 1980s, made great strides during the 1990s, and continues to flourish into the new millennium. This book presents an overall picture of the field, introducing interesting data mining techniques and systems and discussing applications and research directions. An important motivation for writing this book was the need to build an organized framework for the study of data mining—a challenging task, owing to the extensive multidisciplinary nature of this fast-developing field. We hope that this book will encourage people with different backgrounds and experiences to exchange their views regarding data mining so as to contribute toward the further promotion and shaping of this exciting and dynamic field.

Organization of the Book

Since the publication of the first two editions of this book, great progress has been made in the field of data mining. Many new data mining methodologies, systems, and applications have been developed, especially for handling new kinds of data, including information networks, graphs, complex structures, and data streams, as well as text, Web, multimedia, time-series, and spatiotemporal data. Such fast development and rich, new technical contents make it difficult to cover the full spectrum of the field in a single book. Instead of continuously expanding the coverage of this book, we have decided to cover the core material in sufficient scope and depth, and leave the handling of complex data types to a separate forthcoming book.

The third edition substantially revises the first two editions of the book, with numerous enhancements and a reorganization of the technical contents. The core technical material, which handles mining on general data types, is expanded and substantially enhanced. Several individual chapters for topics from the second edition (e.g., data preprocessing, frequent pattern mining, classification, and clustering) are now augmented and each split into two chapters for this new edition. For these topics, one chapter encapsulates the basic concepts and techniques while the other presents advanced concepts and methods.

Chapters from the second edition on mining complex data types (e.g., stream data, sequence data, graph-structured data, social network data, and multirelational data, as well as text, Web, multimedia, and spatiotemporal data) are now reserved for a new book that will be dedicated to *advanced topics in data mining*. Still, to support readers in learning such advanced topics, we have placed an electronic version of the relevant chapters from the second edition onto the book's web site as companion material for the third edition.

The chapters of the third edition are described briefly as follows, with emphasis on the new material.

Chapter 1 provides an *introduction* to the multidisciplinary field of data mining. It discusses the evolutionary path of information technology, which has led to the need for data mining, and the importance of its applications. It examines the data types to be mined, including relational, transactional, and data warehouse data, as well as complex data types such as time-series, sequences, data streams, spatiotemporal data, multimedia data, text data, graphs, social networks, and Web data. The chapter presents a general classification of data mining tasks, based on the kinds of knowledge to be mined, the kinds of technologies used, and the kinds of applications that are targeted. Finally, major challenges in the field are discussed.

Chapter 2 introduces the *general data features*. It first discusses data objects and attribute types and then introduces typical measures for basic statistical data descriptions. It overviews data visualization techniques for various kinds of data. In addition to methods of numeric data visualization, methods for visualizing text, tags, graphs, and multidimensional data are introduced. Chapter 2 also introduces ways to measure similarity and dissimilarity for various kinds of data.

Chapter 3 introduces *techniques for data preprocessing*. It first introduces the concept of data quality and then discusses methods for data cleaning, data integration, data reduction, data transformation, and data discretization.

Chapters 4 and 5 provide a solid introduction to *data warehouses*, *OLAP* (online analytical processing), and *data cube technology*. **Chapter 4** introduces the basic concepts, modeling, design architectures, and general implementations of data warehouses and OLAP, as well as the relationship between data warehousing and other data generalization methods. **Chapter 5** takes an in-depth look at data cube technology, presenting a detailed study of methods of data cube computation, including Star-Cubing and high-dimensional OLAP methods. Further explorations of data cube and OLAP technologies are discussed, such as sampling cubes, ranking cubes, prediction cubes, multifeature cubes for complex analysis queries, and discovery-driven cube exploration.

Chapters 6 and 7 present methods for *mining frequent patterns, associations*, and *correlations* in large data sets. **Chapter 6** introduces fundamental concepts, such as market basket analysis, with many techniques for frequent itemset mining presented in an organized way. These range from the basic Apriori algorithm and its variations to more advanced methods that improve efficiency, including the frequent pattern growth approach, frequent pattern mining with vertical data format, and mining closed and max frequent itemsets. The chapter also discusses pattern evaluation methods and introduces measures for mining correlated patterns. **Chapter 7** is on advanced pattern mining methods. It discusses methods for pattern mining in multilevel and multidimensional space, mining rare and negative patterns, mining colossal patterns and high-dimensional data, constraint-based pattern mining, and mining compressed or approximate patterns. It also introduces methods for pattern exploration and application, including semantic annotation of frequent patterns.

Chapters 8 and 9 describe methods for *data classification*. Due to the importance and diversity of classification methods, the contents are partitioned into two chapters. **Chapter 8** introduces basic concepts and methods for classification, including decision tree induction, Bayes classification, and rule-based classification. It also discusses model evaluation and selection methods and methods for improving classification accuracy, including ensemble methods and how to handle imbalanced data. **Chapter 9** discusses advanced methods for classification, including Bayesian belief networks, the neural network technique of backpropagation, support vector machines, classification using frequent patterns, *k*-nearest-neighbor classifiers, case-based reasoning, genetic algorithms, rough set theory, and fuzzy set approaches. Additional topics include multiclass classification, semi-supervised classification, active learning, and transfer learning.

Cluster analysis forms the topic of Chapters 10 and 11. **Chapter 10** introduces the basic concepts and methods for data clustering, including an overview of basic cluster analysis methods, partitioning methods, hierarchical methods, density-based methods, and grid-based methods. It also introduces methods for the evaluation of clustering. **Chapter 11** discusses advanced methods for clustering, including probabilistic model-based clustering, clustering high-dimensional data, clustering graph and network data, and clustering with constraints.

Chapter 12 is dedicated to *outlier detection*. It introduces the basic concepts of outliers and outlier analysis and discusses various outlier detection methods from the view of degree of supervision (i.e., supervised, semi-supervised, and unsupervised methods), as well as from the view of approaches (i.e., statistical methods, proximity-based methods, clustering-based methods, and classification-based methods). It also discusses methods for mining contextual and collective outliers, and for outlier detection in high-dimensional data.

Finally, in **Chapter 13**, we discuss *trends, applications*, and *research frontiers* in data mining. We briefly cover mining complex data types, including mining sequence data (e.g., time series, symbolic sequences, and biological sequences), mining graphs and networks, and mining spatial, multimedia, text, and Web data. In-depth treatment of data mining methods for such data is left to a book on advanced topics in data mining, the writing of which is in progress. The chapter then moves ahead to cover other data mining methodologies, including statistical data mining, foundations of data mining, visual and audio data mining, as well as data mining applications. It discusses data mining for financial data analysis, for industries like retail and telecommunication, for use in science and engineering, and for intrusion detection and prevention. It also discusses the relationship between data mining and recommender systems. Because data mining is present in many aspects of daily life, we discuss issues regarding data mining and society, including ubiquitous and invisible data mining, as well as privacy, security, and the social impacts of data mining. We conclude our study by looking at data mining trends.

Throughout the text, *italic* font is used to emphasize terms that are defined, while **bold** font is used to highlight or summarize main ideas. Sans serif font is used for reserved words. Bold italic font is used to represent multidimensional quantities.

This book has several strong features that set it apart from other texts on data mining. It presents a very broad yet in-depth coverage of the principles of data mining. The chapters are written to be as self-contained as possible, so they may be read in order of interest by the reader. Advanced chapters offer a larger-scale view and may be considered optional for interested readers. All of the major methods of data mining are presented. The book presents important topics in data mining regarding multidimensional OLAP analysis, which is often overlooked or minimally treated in other data mining books. The book also maintains web sites with a number of online resources to aid instructors, students, and professionals in the field. These are described further in the following.

To the Instructor

This book is designed to give a broad, yet detailed overview of the data mining field. It can be used to teach an introductory course on data mining at an advanced undergraduate level or at the first-year graduate level. Sample course syllabi are provided on the book's web sites (*www.cs.uiuc.edu/~hanj/bk3* and *www.booksite.mkp.com/datamining3e*) in addition to extensive teaching resources such as lecture slides, instructors' manuals, and reading lists (see p. xxix).

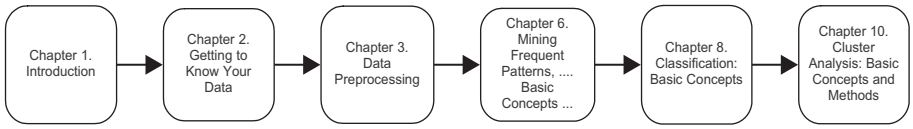


Figure P.1 A suggested sequence of chapters for a short introductory course.

Depending on the length of the instruction period, the background of students, and your interests, you may select subsets of chapters to teach in various sequential orderings. For example, if you would like to give only a short introduction to students on data mining, you may follow the suggested sequence in Figure P.1. Notice that depending on the need, you can also omit some sections or subsections in a chapter if desired.

Depending on the length of the course and its technical scope, you may choose to selectively add more chapters to this preliminary sequence. For example, instructors who are more interested in advanced classification methods may first add "Chapter 9. Classification: Advanced Methods"; those more interested in pattern mining may choose to include "Chapter 7. Advanced Pattern Mining"; whereas those interested in OLAP and data cube technology may like to add "Chapter 4. Data Warehousing and Online Analytical Processing" and "Chapter 5. Data Cube Technology."

Alternatively, you may choose to teach the whole book in a two-course sequence that covers all of the chapters in the book, plus, when time permits, some advanced topics such as graph and network mining. Material for such advanced topics may be selected from the companion chapters available from the book's web site, accompanied with a set of selected research papers.

Individual chapters in this book can also be used for tutorials or for special topics in related courses, such as machine learning, pattern recognition, data warehousing, and intelligent data analysis.

Each chapter ends with a set of exercises, suitable as assigned homework. The exercises are either short questions that test basic mastery of the material covered, longer questions that require analytical thinking, or implementation projects. Some exercises can also be used as research discussion topics. The bibliographic notes at the end of each chapter can be used to find the research literature that contains the origin of the concepts and methods presented, in-depth treatment of related topics, and possible extensions.

To the Student

We hope that this textbook will spark your interest in the young yet fast-evolving field of data mining. We have attempted to present the material in a clear manner, with careful explanation of the topics covered. Each chapter ends with a summary describing the main points. We have included many figures and illustrations throughout the text to make the book more enjoyable and reader-friendly. Although this book was designed as a textbook, we have tried to organize it so that it will also be useful to you as a reference

book or handbook, should you later decide to perform in-depth research in the related fields or pursue a career in data mining.

What do you need to know to read this book?

- You should have some knowledge of the concepts and terminology associated with statistics, database systems, and machine learning. However, we do try to provide enough background of the basics, so that if you are not so familiar with these fields or your memory is a bit rusty, you will not have trouble following the discussions in the book.
- You should have some programming experience. In particular, you should be able to read pseudocode and understand simple data structures such as multidimensional arrays.

To the Professional

This book was designed to cover a wide range of topics in the data mining field. As a result, it is an excellent handbook on the subject. Because each chapter is designed to be as standalone as possible, you can focus on the topics that most interest you. The book can be used by application programmers and information service managers who wish to learn about the key ideas of data mining on their own. The book would also be useful for technical data analysis staff in banking, insurance, medicine, and retailing industries who are interested in applying data mining solutions to their businesses. Moreover, the book may serve as a comprehensive survey of the data mining field, which may also benefit researchers who would like to advance the state-of-the-art in data mining and extend the scope of data mining applications.

The techniques and algorithms presented are of practical utility. Rather than selecting algorithms that perform well on small "toy" data sets, the algorithms described in the book are geared for the discovery of patterns and knowledge hidden in large, real data sets. Algorithms presented in the book are illustrated in pseudocode. The pseudocode is similar to the C programming language, yet is designed so that it should be easy to follow by programmers unfamiliar with C or C++. If you wish to implement any of the algorithms, you should find the translation of our pseudocode into the programming language of your choice to be a fairly straightforward task.

Book Web Sites with Resources

The book has a web site at *www.cs.uiuc.edu/~hanj/bk3* and another with Morgan Kaufmann Publishers at *www.booksite.mkp.com/datamining3e*. These web sites contain many supplemental materials for readers of this book or anyone else with an interest in data mining. The resources include the following:

- **Slide presentations for each chapter**. Lecture notes in Microsoft PowerPoint slides are available for each chapter.

- **Companion chapters on advanced data mining**. Chapters 8 to 10 of the second edition of the book, which cover mining complex data types, are available on the book's web sites for readers who are interested in learning more about such advanced topics, beyond the themes covered in this book.
- **Instructors' manual**. This complete set of answers to the exercises in the book is available only to instructors from the publisher's web site.
- **Course syllabi and lecture plans**. These are given for undergraduate and graduate versions of introductory and advanced courses on data mining, which use the text and slides.
- **Supplemental reading lists with hyperlinks**. Seminal papers for supplemental reading are organized per chapter.
- **Links to data mining data sets and software**. We provide a set of links to data mining data sets and sites that contain interesting data mining software packages, such as IlliMine from the University of Illinois at Urbana-Champaign (*http://illimine.cs.uiuc.edu*).
- **Sample assignments, exams, and course projects**. A set of sample assignments, exams, and course projects is available to instructors from the publisher's web site.
- **Figures from the book**. This may help you to make your own slides for your classroom teaching.
- **Contents** of the book in PDF format.
- **Errata on the different printings of the book**. We encourage you to point out any errors in this book. Once the error is confirmed, we will update the errata list and include acknowledgment of your contribution.

Comments or suggestions can be sent to *hanj@cs.uiuc.edu*. We would be happy to hear from you.

This page intentionally left blank

Acknowledgments

Third Edition of the Book

We would like to express our grateful thanks to all of the previous and current members of the Data Mining Group at UIUC, the faculty and students in the Data and Information Systems (DAIS) Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign, and many friends and colleagues, whose constant support and encouragement have made our work on this edition a rewarding experience. We would also like to thank students in CS412 and CS512 classes at UIUC of the 2010–2011 academic year, who carefully went through the early drafts of this book, identified many errors, and suggested various improvements.

We also wish to thank David Bevans and Rick Adams at Morgan Kaufmann Publishers, for their enthusiasm, patience, and support during our writing of this edition of the book. We thank Marilyn Rash, the Project Manager, and her team members, for keeping us on schedule.

We are also grateful for the invaluable feedback from all of the reviewers. Moreover, we would like to thank U.S. National Science Foundation, NASA, U.S. Air Force Office of Scientific Research, U.S. Army Research Laboratory, and Natural Science and Engineering Research Council of Canada (NSERC), as well as IBM Research, Microsoft Research, Google, Yahoo! Research, Boeing, HP Labs, and other industry research labs for their support of our research in the form of research grants, contracts, and gifts. Such research support deepens our understanding of the subjects discussed in this book. Finally, we thank our families for their wholehearted support throughout this project.

Second Edition of the Book

We would like to express our grateful thanks to all of the previous and current members of the Data Mining Group at UIUC, the faculty and students in the Data and

Information Systems (DAIS) Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign, and many friends and colleagues, whose constant support and encouragement have made our work on this edition a rewarding experience. These include Gul Agha, Rakesh Agrawal, Loretta Auvil, Peter Bajcsy, Geneva Belford, Deng Cai, Y. Dora Cai, Roy Cambell, Kevin C.-C. Chang, Surajit Chaudhuri, Chen Chen, Yixin Chen, Yuguo Chen, Hong Cheng, David Cheung, Shengnan Cong, Gerald DeJong, AnHai Doan, Guozhu Dong, Charios Ermopoulos, Martin Ester, Christos Faloutsos, Wei Fan, Jack C. Feng, Ada Fu, Michael Garland, Johannes Gehrke, Hector Gonzalez, Mehdi Harandi, Thomas Huang, Wen Jin, Chulyun Kim, Sangkyum Kim, Won Kim, Won-Young Kim, David Kuck, Young-Koo Lee, Harris Lewin, Xiaolei Li, Yifan Li, Chao Liu, Han Liu, Huan Liu, Hongyan Liu, Lei Liu, Ying Lu, Klara Nahrstedt, David Padua, Jian Pei, Lenny Pitt, Daniel Reed, Dan Roth, Bruce Schatz, Zheng Shao, Marc Snir, Zhaohui Tang, Bhavani M. Thuraisingham, Josep Torrellas, Peter Tzvetkov, Benjamin W. Wah, Haixun Wang, Jianyong Wang, Ke Wang, Muyuan Wang, Wei Wang, Michael Welge, Marianne Winslett, Ouri Wolfson, Andrew Wu, Tianyi Wu, Dong Xin, Xifeng Yan, Jiong Yang, Xiaoxin Yin, Hwanjo Yu, Jeffrey X. Yu, Philip S. Yu, Maria Zemankova, ChengXiang Zhai, Yuanyuan Zhou, and Wei Zou.

Deng Cai and ChengXiang Zhai have contributed to the text mining and Web mining sections, Xifeng Yan to the graph mining section, and Xiaoxin Yin to the multirelational data mining section. Hong Cheng, Charios Ermopoulos, Hector Gonzalez, David J. Hill, Chulyun Kim, Sangkyum Kim, Chao Liu, Hongyan Liu, Kasif Manzoor, Tianyi Wu, Xifeng Yan, and Xiaoxin Yin have contributed to the proofreading of the individual chapters of the manuscript.

We also wish to thank Diane Cerra, our Publisher at Morgan Kaufmann Publishers, for her constant enthusiasm, patience, and support during our writing of this book. We are indebted to Alan Rose, the book Production Project Manager, for his tireless and ever-prompt communications with us to sort out all details of the production process. We are grateful for the invaluable feedback from all of the reviewers. Finally, we thank our families for their wholehearted support throughout this project.

First Edition of the Book

We would like to express our sincere thanks to all those who have worked or are currently working with us on data mining–related research and/or the DBMiner project, or have provided us with various support in data mining. These include Rakesh Agrawal, Stella Atkins, Yvan Bedard, Binay Bhattacharya, (Yandong) Dora Cai, Nick Cercone, Surajit Chaudhuri, Sonny H. S. Chee, Jianping Chen, Ming-Syan Chen, Qing Chen, Qiming Chen, Shan Cheng, David Cheung, Shi Cong, Son Dao, Umeshwar Dayal, James Delgrande, Guozhu Dong, Carole Edwards, Max Egenhofer, Martin Ester, Usama Fayyad, Ling Feng, Ada Fu, Yongjian Fu, Daphne Gelbart, Randy Goebel, Jim Gray, Robert Grossman, Wan Gong, Yike Guo, Eli Hagen, Howard Hamilton, Jing He, Larry Henschen, Jean Hou, Mei-Chun Hsu, Kan Hu, Haiming Huang, Yue Huang, Julia Itskevitch, Wen Jin, Tiko Kameda, Hiroyuki Kawano, Rizwan Kheraj, Eddie Kim, Won Kim, Krzysztof Koperski, Hans-Peter Kriegel, Vipin Kumar, Laks V. S. Lakshmanan, Joyce

Man Lam, James Lau, Deyi Li, George (Wenmin) Li, Jin Li, Ze-Nian Li, Nancy Liao, Gang Liu, Junqiang Liu, Ling Liu, Alan (Yijun) Lu, Hongjun Lu, Tong Lu, Wei Lu, Xuebin Lu, Wo-Shun Luk, Heikki Mannila, Runying Mao, Abhay Mehta, Gabor Melli, Alberto Mendelzon, Tim Merrett, Harvey Miller, Drew Miners, Behzad Mortazavi-Asl, Richard Muntz, Raymond T. Ng, Vicent Ng, Shojiro Nishio, Beng-Chin Ooi, Tamer Ozsu, Jian Pei, Gregory Piatetsky-Shapiro, Helen Pinto, Fred Popowich, Amynmohamed Rajan, Peter Scheuermann, Shashi Shekhar, Wei-Min Shen, Avi Silberschatz, Evangelos Simoudis, Nebojsa Stefanovic, Yin Jenny Tam, Simon Tang, Zhaohui Tang, Dick Tsur, Anthony K. H. Tung, Ke Wang, Wei Wang, Zhaoxia Wang, Tony Wind, Lara Winstone, Ju Wu, Betty (Bin) Xia, Cindy M. Xin, Xiaowei Xu, Qiang Yang, Yiwen Yin, Clement Yu, Jeffrey Yu, Philip S. Yu, Osmar R. Zaiane, Carlo Zaniolo, Shuhua Zhang, Zhong Zhang, Yvonne Zheng, Xiaofang Zhou, and Hua Zhu.

We are also grateful to Jean Hou, Helen Pinto, Lara Winstone, and Hua Zhu for their help with some of the original figures in this book, and to Eugene Belchev for his careful proofreading of each chapter.

We also wish to thank Diane Cerra, our Executive Editor at Morgan Kaufmann Publishers, for her enthusiasm, patience, and support during our writing of this book, as well as Howard Severson, our Production Editor, and his staff for their conscientious efforts regarding production. We are indebted to all of the reviewers for their invaluable feedback. Finally, we thank our families for their wholehearted support throughout this project.

This page intentionally left blank

About the Authors

Jiawei Han is a Bliss Professor of Engineering in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He has received numerous awards for his contributions on research into knowledge discovery and data mining, including ACM SIGKDD Innovation Award (2004), IEEE Computer Society Technical Achievement Award (2005), and IEEE W. Wallace McDowell Award (2009). He is a Fellow of ACM and IEEE. He served as founding Editor-in-Chief of *ACM Transactions on Knowledge Discovery from Data* (2006–2011) and as an editorial board member of several journals, including *IEEE Transactions on Knowledge and Data Engineering* and *Data Mining and Knowledge Discovery*.

Micheline Kamber has a master's degree in computer science (specializing in artificial intelligence) from Concordia University in Montreal, Quebec. She was an NSERC Scholar and has worked as a researcher at McGill University, Simon Fraser University, and in Switzerland. Her background in data mining and passion for writing in easy-to-understand terms help make this text a favorite of professionals, instructors, and students.

Jian Pei is currently an associate professor at the School of Computing Science, Simon Fraser University in British Columbia. He received a Ph.D. degree in computing science from Simon Fraser University in 2002 under Dr. Jiawei Han's supervision. He has published prolifically in the premier academic forums on data mining, databases, Web searching, and information retrieval and actively served the academic community. His publications have received thousands of citations and several prestigious awards. He is an associate editor of several data mining and data analytics journals.

This page intentionally left blank

Introduction

This book is an introduction to the young and fast-growing field of *data mining* (also known as *knowledge discovery from data*, or *KDD* for short). The book focuses on fundamental data mining concepts and techniques for discovering interesting patterns from data in various applications. In particular, we emphasize prominent techniques for developing effective, efficient, and scalable data mining tools.

This chapter is organized as follows. In Section 1.1, you will learn why data mining is in high demand and how it is part of the natural evolution of information technology. Section 1.2 defines data mining with respect to the knowledge discovery process. Next, you will learn about data mining from many aspects, such as the kinds of data that can be mined (Section 1.3), the kinds of knowledge to be mined (Section 1.4), the kinds of technologies to be used (Section 1.5), and targeted applications (Section 1.6). In this way, you will gain a multidimensional view of data mining. Finally, Section 1.7 outlines major data mining research and development issues.

1.1 Why Data Mining?

Necessity, who is the mother of invention. – Plato

We live in a world where vast amounts of data are collected daily. Analyzing such data is an important need. Section 1.1.1 looks at how data mining can meet this need by providing tools to discover knowledge from data. In Section 1.1.2, we observe how data mining can be viewed as a result of the natural evolution of information technology.

1.1.1 Moving toward the Information Age

"*We are living in the information age*" is a popular saying; however, *we are actually living in the data age*. Terabytes or petabytes[1] of data pour into our computer networks, the World Wide Web (WWW), and various data storage devices every day from business,

[1] A petabyte is a unit of information or computer storage equal to 1 quadrillion bytes, or a thousand terabytes, or 1 million gigabytes.

society, science and engineering, medicine, and almost every other aspect of daily life. This explosive growth of available data volume is a result of the computerization of our society and the fast development of powerful data collection and storage tools. Businesses worldwide generate gigantic data sets, including sales transactions, stock trading records, product descriptions, sales promotions, company profiles and performance, and customer feedback. For example, large stores, such as Wal-Mart, handle hundreds of millions of transactions per week at thousands of branches around the world. Scientific and engineering practices generate high orders of petabytes of data in a continuous manner, from remote sensing, process measuring, scientific experiments, system performance, engineering observations, and environment surveillance.

Global backbone telecommunication networks carry tens of petabytes of data traffic every day. The medical and health industry generates tremendous amounts of data from medical records, patient monitoring, and medical imaging. Billions of Web searches supported by search engines process tens of petabytes of data daily. Communities and social media have become increasingly important data sources, producing digital pictures and videos, blogs, Web communities, and various kinds of social networks. The list of sources that generate huge amounts of data is endless.

This explosively growing, widely available, and gigantic body of data makes our time truly the data age. Powerful and versatile tools are badly needed to automatically uncover valuable information from the tremendous amounts of data and to transform such data into organized knowledge. This necessity has led to the birth of data mining. The field is young, dynamic, and promising. Data mining has and will continue to make great strides in our journey from the data age toward the coming information age.

Example 1.1 **Data mining turns a large collection of data into knowledge.** A search engine (e.g., Google) receives hundreds of millions of queries every day. Each query can be viewed as a transaction where the user describes her or his information need. What novel and useful knowledge can a search engine learn from such a huge collection of queries collected from users over time? Interestingly, some patterns found in user search queries can disclose invaluable knowledge that cannot be obtained by reading individual data items alone. For example, Google's *Flu Trends* uses specific search terms as indicators of flu activity. It found a close relationship between the number of people who search for flu-related information and the number of people who actually have flu symptoms. A pattern emerges when all of the search queries related to flu are aggregated. Using aggregated Google search data, *Flu Trends* can estimate flu activity up to two weeks faster than traditional systems can.[2] This example shows how data mining can turn a large collection of data into knowledge that can help meet a current global challenge. ■

1.1.2 Data Mining as the Evolution of Information Technology

Data mining can be viewed as a result of the natural evolution of information technology. The database and data management industry evolved in the development of

[2]This is reported in [GMP+09].

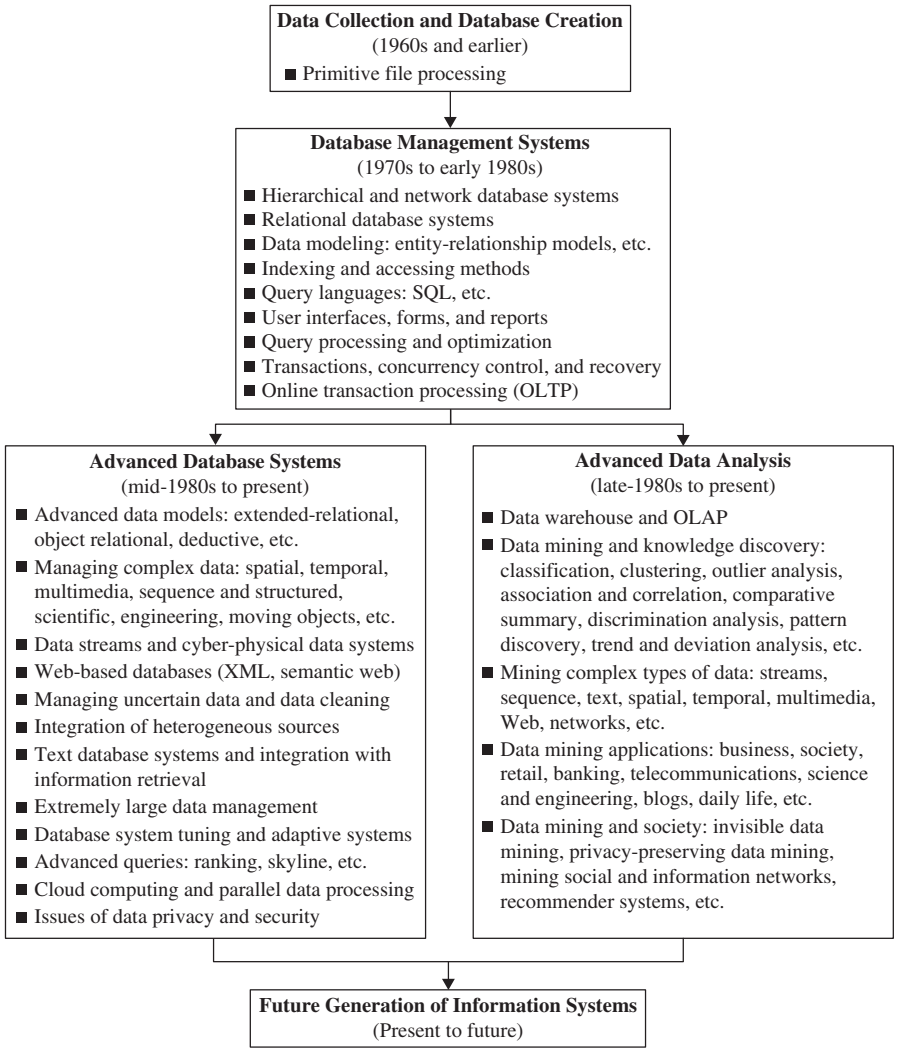


Figure 1.1 The evolution of database system technology.

several critical functionalities (Figure 1.1): *data collection and database creation, data management* (including data storage and retrieval and database transaction processing), and *advanced data analysis* (involving data warehousing and data mining). The early development of data collection and database creation mechanisms served as a prerequisite for the later development of effective mechanisms for data storage and retrieval, as well as query and transaction processing. Nowadays numerous database systems offer query and transaction processing as common practice. Advanced data analysis has naturally become the next step.

Since the 1960s, database and information technology has evolved systematically from primitive file processing systems to sophisticated and powerful database systems. The research and development in database systems since the 1970s progressed from early hierarchical and network database systems to relational database systems (where data are stored in relational table structures; see Section 1.3.1), data modeling tools, and indexing and accessing methods. In addition, users gained convenient and flexible data access through query languages, user interfaces, query optimization, and transaction management. Efficient methods for online transaction processing (OLTP), where a query is viewed as a read-only transaction, contributed substantially to the evolution and wide acceptance of relational technology as a major tool for efficient storage, retrieval, and management of large amounts of data.

After the establishment of database management systems, database technology moved toward the development of *advanced database systems*, *data warehousing*, and *data mining* for advanced data analysis and *web-based databases*. Advanced database systems, for example, resulted from an upsurge of research from the mid-1980s onward. These systems incorporate new and powerful data models such as extended-relational, object-oriented, object-relational, and deductive models. Application-oriented database systems have flourished, including spatial, temporal, multimedia, active, stream and sensor, scientific and engineering databases, knowledge bases, and office information bases. Issues related to the distribution, diversification, and sharing of data have been studied extensively.

Advanced data analysis sprang up from the late 1980s onward. The steady and dazzling progress of computer hardware technology in the past three decades led to large supplies of powerful and affordable computers, data collection equipment, and storage media. This technology provides a great boost to the database and information industry, and it enables a huge number of databases and information repositories to be available for transaction management, information retrieval, and data analysis. Data can now be stored in many different kinds of databases and information repositories.

One emerging data repository architecture is the **data warehouse** (Section 1.3.2). This is a repository of multiple heterogeneous data sources organized under a unified schema at a single site to facilitate management decision making. Data warehouse technology includes data cleaning, data integration, and online analytical processing (OLAP)—that is, analysis techniques with functionalities such as summarization, consolidation, and aggregation, as well as the ability to view information from different angles. Although OLAP tools support multidimensional analysis and decision making, additional data analysis tools are required for in-depth analysis—for example, data mining tools that provide data classification, clustering, outlier/anomaly detection, and the characterization of changes in data over time.

Huge volumes of data have been accumulated beyond databases and data warehouses. During the 1990s, the World Wide Web and web-based databases (e.g., XML databases) began to appear. Internet-based global information bases, such as the WWW and various kinds of interconnected, heterogeneous databases, have emerged and play a vital role in the information industry. The effective and efficient analysis of data from such different forms of data by integration of information retrieval, data mining, and information network analysis technologies is a challenging task.

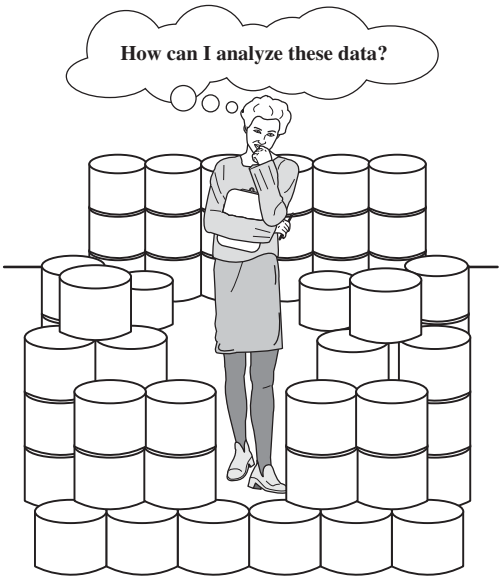


Figure 1.2 The world is data rich but information poor.

In summary, the abundance of data, coupled with the need for powerful data analysis tools, has been described as a *data rich but information poor* situation (Figure 1.2). The fast-growing, tremendous amount of data, collected and stored in large and numerous data repositories, has far exceeded our human ability for comprehension without powerful tools. As a result, data collected in large data repositories become "data tombs"—data archives that are seldom visited. Consequently, important decisions are often made based not on the information-rich data stored in data repositories but rather on a decision maker's intuition, simply because the decision maker does not have the tools to extract the valuable knowledge embedded in the vast amounts of data. Efforts have been made to develop expert system and knowledge-based technologies, which typically rely on users or domain experts to *manually* input knowledge into knowledge bases. Unfortunately, however, the manual knowledge input procedure is prone to biases and errors and is extremely costly and time consuming. The widening gap between data and information calls for the systematic development of *data mining tools* that can turn data tombs into "golden nuggets" of knowledge.

1.2 What Is Data Mining?

It is no surprise that data mining, as a truly interdisciplinary subject, can be defined in many different ways. Even the term *data mining* does not really present all the major components in the picture. To refer to the mining of gold from rocks or sand, we say *gold mining* instead of rock or sand mining. Analogously, data mining should have been more

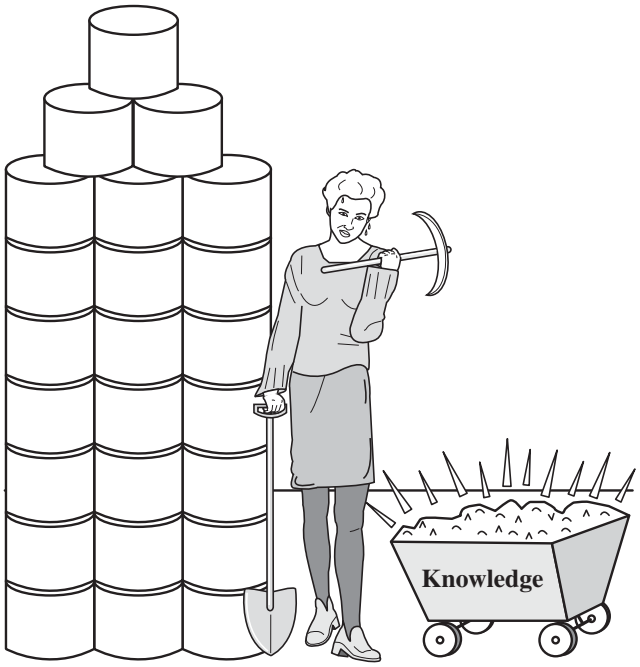


Figure 1.3 Data mining—searching for knowledge (interesting patterns) in data.

appropriately named "knowledge mining from data," which is unfortunately somewhat long. However, the shorter term, *knowledge mining* may not reflect the emphasis on mining from large amounts of data. Nevertheless, mining is a vivid term characterizing the process that finds a small set of precious nuggets from a great deal of raw material (Figure 1.3). Thus, such a misnomer carrying both "data" and "mining" became a popular choice. In addition, many other terms have a similar meaning to data mining—for example, *knowledge mining from data*, *knowledge extraction*, *data/pattern analysis*, *data archaeology*, and *data dredging*.

Many people treat data mining as a synonym for another popularly used term, **knowledge discovery from data**, or **KDD**, while others view data mining as merely an essential step in the process of knowledge discovery. The knowledge discovery process is shown in Figure 1.4 as an iterative sequence of the following steps:

1. **Data cleaning** (to remove noise and inconsistent data)
2. **Data integration** (where multiple data sources may be combined)[3]

[3]A popular trend in the information industry is to perform data cleaning and data integration as a preprocessing step, where the resulting data are stored in a data warehouse.

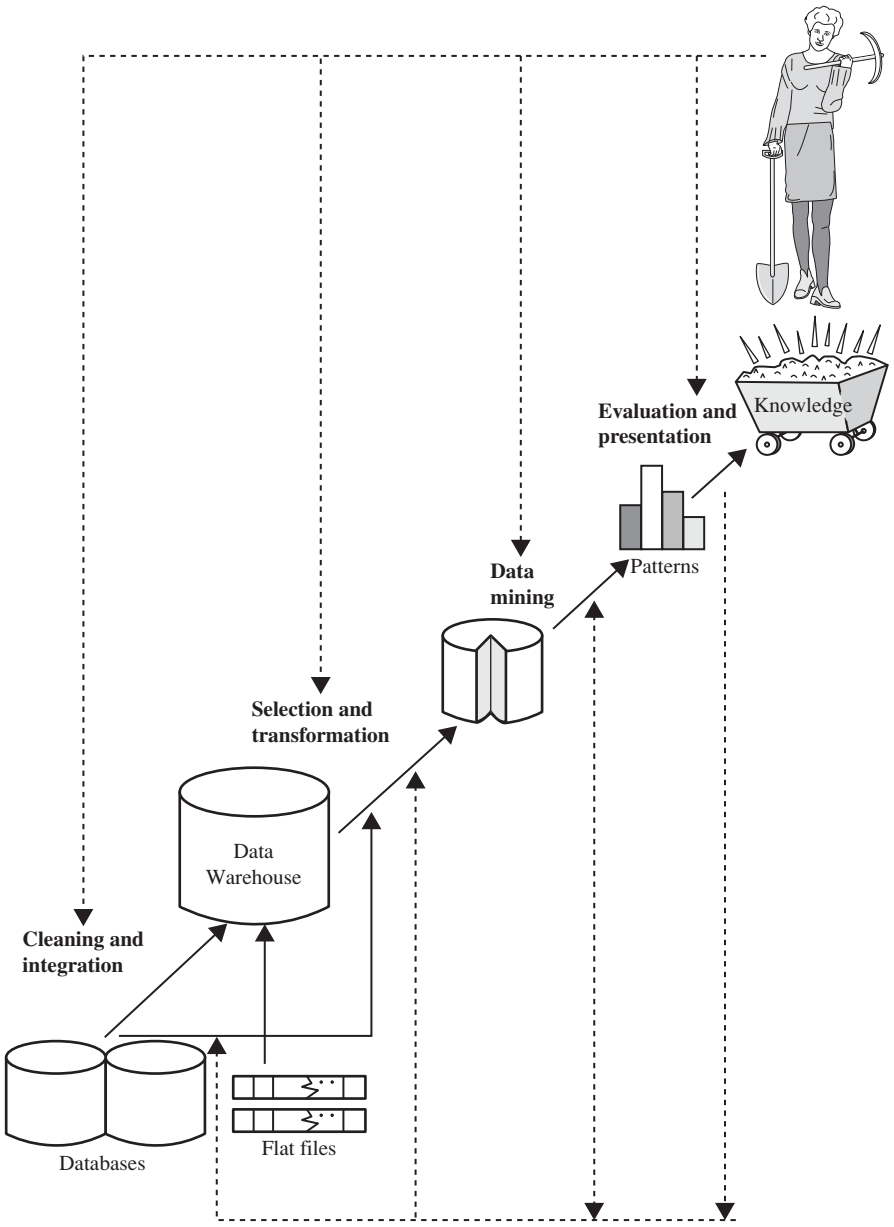


Figure 1.4 Data mining as a step in the process of knowledge discovery.

3. **Data selection** (where data relevant to the analysis task are retrieved from the database)
4. **Data transformation** (where data are transformed and consolidated into forms appropriate for mining by performing summary or aggregation operations)[4]
5. **Data mining** (an essential process where intelligent methods are applied to extract data patterns)
6. **Pattern evaluation** (to identify the truly interesting patterns representing knowledge based on *interestingness measures*—see Section 1.4.6)
7. **Knowledge presentation** (where visualization and knowledge representation techniques are used to present mined knowledge to users)

Steps 1 through 4 are different forms of data preprocessing, where data are prepared for mining. The data mining step may interact with the user or a knowledge base. The interesting patterns are presented to the user and may be stored as new knowledge in the knowledge base.

The preceding view shows data mining as one step in the knowledge discovery process, albeit an essential one because it uncovers hidden patterns for evaluation. However, in industry, in media, and in the research milieu, the term *data mining* is often used to refer to the entire knowledge discovery process (perhaps because the term is shorter than *knowledge discovery from data*). Therefore, we adopt a broad view of data mining functionality: **Data mining** is the *process* of discovering interesting patterns and knowledge from *large* amounts of data. The data sources can include databases, data warehouses, the Web, other information repositories, or data that are streamed into the system dynamically.

1.3 What Kinds of Data Can Be Mined?

As a general technology, data mining can be applied to any kind of data as long as the data are meaningful for a target application. The most basic forms of data for mining applications are database data (Section 1.3.1), data warehouse data (Section 1.3.2), and transactional data (Section 1.3.3). The concepts and techniques presented in this book focus on such data. Data mining can also be applied to other forms of data (e.g., data streams, ordered/sequence data, graph or networked data, spatial data, text data, multimedia data, and the WWW). We present an overview of such data in Section 1.3.4. Techniques for mining of these kinds of data are briefly introduced in Chapter 13. In-depth treatment is considered an advanced topic. Data mining will certainly continue to embrace new data types as they emerge.

[4]Sometimes data transformation and consolidation are performed before the data selection process, particularly in the case of data warehousing. *Data reduction* may also be performed to obtain a smaller representation of the original data without sacrificing its integrity.

1.3.1 Database Data

A database system, also called a **database management system** (**DBMS**), consists of a collection of interrelated data, known as a **database**, and a set of software programs to manage and access the data. The software programs provide mechanisms for defining database structures and data storage; for specifying and managing concurrent, shared, or distributed data access; and for ensuring consistency and security of the information stored despite system crashes or attempts at unauthorized access.

A **relational database** is a collection of **tables**, each of which is assigned a unique name. Each table consists of a set of **attributes** (*columns* or *fields*) and usually stores a large set of **tuples** (*records* or *rows*). Each tuple in a relational table represents an object identified by a unique *key* and described by a set of attribute values. A semantic data model, such as an **entity-relationship (ER)** data model, is often constructed for relational databases. An ER data model represents the database as a set of entities and their relationships.

Example 1.2 A relational database for *AllElectronics*. The fictitious *AllElectronics* store is used to illustrate concepts throughout this book. The company is described by the following relation tables: *customer, item, employee*, and *branch*. The headers of the tables described here are shown in Figure 1.5. (A header is also called the *schema* of a relation.)

- The relation *customer* consists of a set of attributes describing the customer information, including a unique customer identity number (*cust_ID*), customer name, address, age, occupation, annual income, credit information, and category.
- Similarly, each of the relations *item*, *employee*, and *branch* consists of a set of attributes describing the properties of these entities.
- Tables can also be used to represent the relationships between or among multiple entities. In our example, these include *purchases* (customer purchases items, creating a sales transaction handled by an employee), *items_sold* (lists items sold in a given transaction), and *works_at* (employee works at a branch of *AllElectronics*). ■

customer	(*cust_ID, name, address, age, occupation, annual_income, credit_information, category*, ...)
item	(*item_ID, brand, category, type, price, place_made, supplier, cost*, ...)
employee	(*empl_ID, name, category, group, salary, commission*, ...)
branch	(*branch_ID, name, address*, ...)
purchases	(*trans_ID, cust_ID, empl_ID, date, time, method_paid, amount*)
items_sold	(*trans_ID, item_ID, qty*)
works_at	(*empl_ID, branch_ID*)

Figure 1.5 Relational schema for a relational database, *AllElectronics*.

Relational data can be accessed by **database queries** written in a relational query language (e.g., SQL) or with the assistance of graphical user interfaces. A given query is transformed into a set of relational operations, such as join, selection, and projection, and is then optimized for efficient processing. A query allows retrieval of specified subsets of the data. Suppose that your job is to analyze the *AllElectronics* data. Through the use of relational queries, you can ask things like, "*Show me a list of all items that were sold in the last quarter.*" Relational languages also use aggregate functions such as sum, avg (average), count, max (maximum), and min (minimum). Using aggregates allows you to ask: "*Show me the total sales of the last month, grouped by branch,*" or "*How many sales transactions occurred in the month of December?*" or "*Which salesperson had the highest sales?*"

When **mining relational databases**, we can go further by *searching for trends* or *data patterns*. For example, data mining systems can analyze customer data to predict the credit risk of new customers based on their income, age, and previous credit information. Data mining systems may also detect deviations—that is, items with sales that are far from those expected in comparison with the previous year. Such deviations can then be further investigated. For example, data mining may discover that there has been a change in packaging of an item or a significant increase in price.

Relational databases are one of the most commonly available and richest information repositories, and thus they are a major data form in the study of data mining.

1.3.2 Data Warehouses

Suppose that *AllElectronics* is a successful international company with branches around the world. Each branch has its own set of databases. The president of *AllElectronics* has asked you to provide an analysis of the company's sales per item type per branch for the third quarter. This is a difficult task, particularly since the relevant data are spread out over several databases physically located at numerous sites.

If *AllElectronics* had a data warehouse, this task would be easy. A **data warehouse** is a repository of information collected from multiple sources, stored under a unified schema, and usually residing at a single site. Data warehouses are constructed via a process of data cleaning, data integration, data transformation, data loading, and periodic data refreshing. This process is discussed in Chapters 3 and 4. Figure 1.6 shows the typical framework for construction and use of a data warehouse for *AllElectronics*.

To facilitate decision making, the data in a data warehouse are organized around *major subjects* (e.g., customer, item, supplier, and activity). The data are stored to provide information from a *historical perspective*, such as in the past 6 to 12 months, and are typically *summarized*. For example, rather than storing the details of each sales transaction, the data warehouse may store a summary of the transactions per item type for each store or, summarized to a higher level, for each sales region.

A data warehouse is usually modeled by a multidimensional data structure, called a **data cube**, in which each **dimension** corresponds to an attribute or a set of attributes in the schema, and each **cell** stores the value of some aggregate measure such as *count*

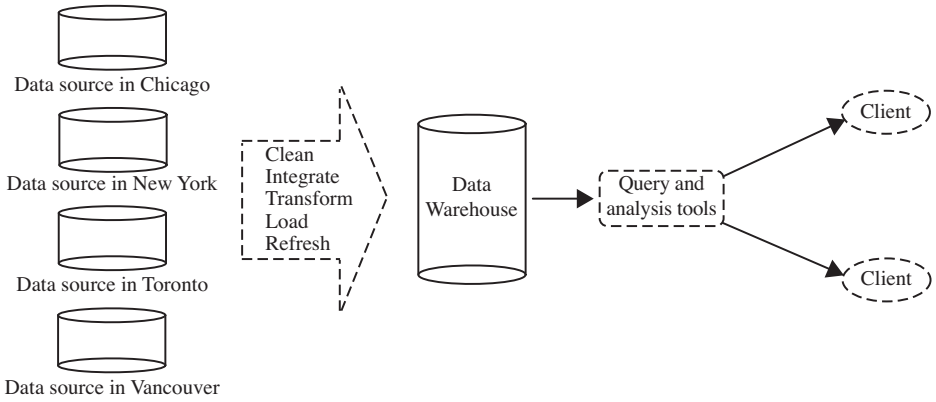


Figure 1.6 Typical framework of a data warehouse for *AllElectronics*.

or *sum*(*sales_amount*). A data cube provides a multidimensional view of data and allows the precomputation and fast access of summarized data.

Example 1.3 **A data cube for *AllElectronics*.** A data cube for summarized sales data of *AllElectronics* is presented in Figure 1.7(a). The cube has three dimensions: *address* (with city values *Chicago, New York, Toronto, Vancouver*), *time* (with quarter values *Q1, Q2, Q3, Q4*), and *item* (with item type values *home entertainment, computer, phone, security*). The aggregate value stored in each cell of the cube is *sales_amount* (in thousands). For example, the total sales for the first quarter, *Q1*, for the items related to security systems in Vancouver is $400,000, as stored in cell ⟨*Vancouver, Q1, security*⟩. Additional cubes may be used to store aggregate sums over each dimension, corresponding to the aggregate values obtained using different SQL group-bys (e.g., the total sales amount per city and quarter, or per city and item, or per quarter and item, or per each individual dimension). ■

By providing multidimensional data views and the precomputation of summarized data, data warehouse systems can provide inherent support for OLAP. Online analytical processing operations make use of background knowledge regarding the domain of the data being studied to allow the presentation of data at *different levels of abstraction*. Such operations accommodate different user viewpoints. Examples of OLAP operations include **drill-down** and **roll-up**, which allow the user to view the data at differing degrees of summarization, as illustrated in Figure 1.7(b). For instance, we can drill down on sales data summarized by *quarter* to see data summarized by *month*. Similarly, we can roll up on sales data summarized by *city* to view data summarized by *country*.

Although data warehouse tools help support data analysis, additional tools for data mining are often needed for in-depth analysis. **Multidimensional data mining** (also called **exploratory multidimensional data mining**) performs data mining in

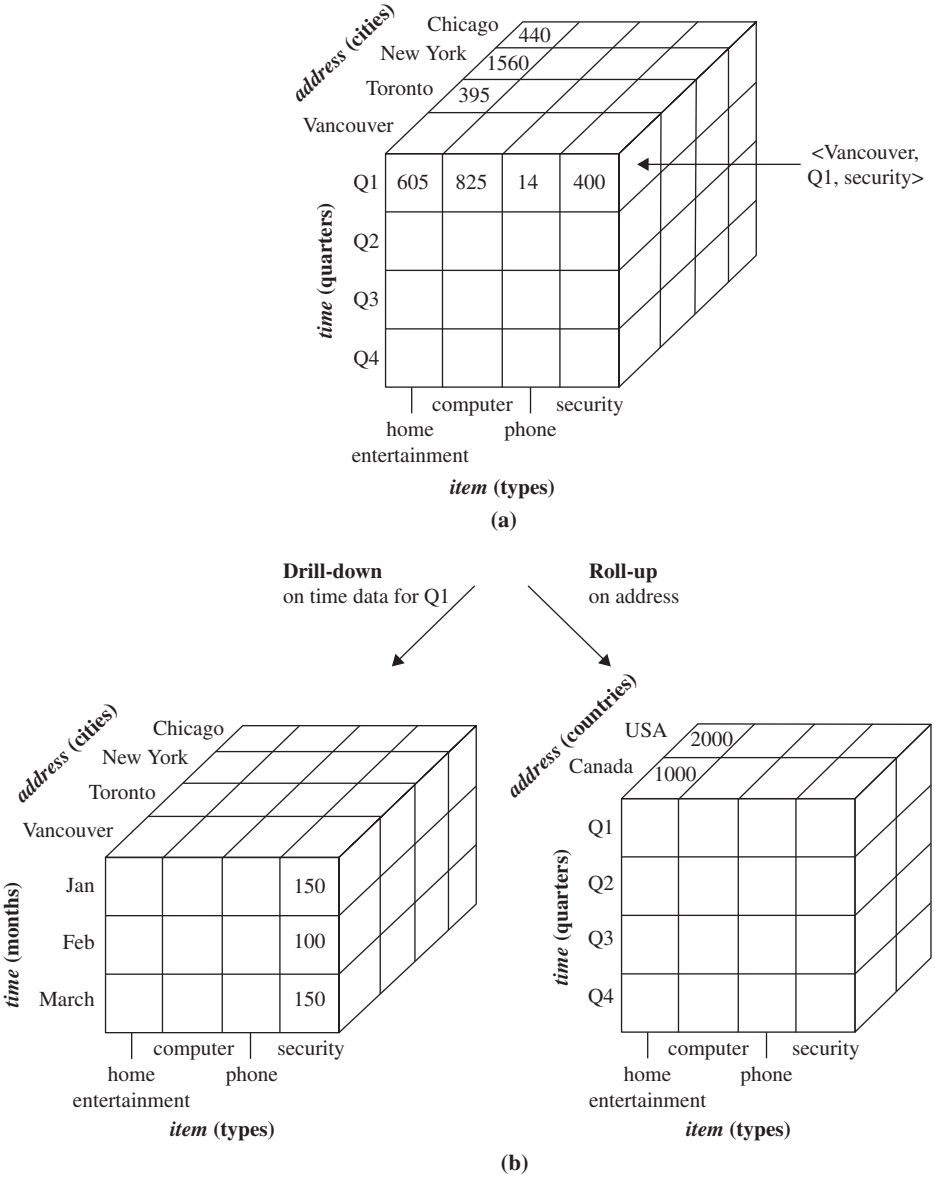


Figure 1.7 A multidimensional data cube, commonly used for data warehousing, (a) showing summarized data for *AllElectronics* and (b) showing summarized data resulting from drill-down and roll-up operations on the cube in (a). For improved readability, only some of the cube cell values are shown.

multidimensional space in an OLAP style. That is, it allows the exploration of multiple combinations of dimensions at varying levels of granularity in data mining, and thus has greater potential for discovering interesting patterns representing knowledge. An overview of data warehouse and OLAP technology is provided in Chapter 4. Advanced issues regarding data cube computation and multidimensional data mining are discussed in Chapter 5.

1.3.3 Transactional Data

In general, each record in a **transactional database** captures a transaction, such as a customer's purchase, a flight booking, or a user's clicks on a web page. A transaction typically includes a unique transaction identity number (*trans_ID*) and a list of the **items** making up the transaction, such as the items purchased in the transaction. A transactional database may have additional tables, which contain other information related to the transactions, such as item description, information about the salesperson or the branch, and so on.

Example 1.4 **A transactional database for *AllElectronics*.** Transactions can be stored in a table, with one record per transaction. A fragment of a transactional database for *AllElectronics* is shown in Figure 1.8. From the relational database point of view, the *sales* table in the figure is a nested relation because the attribute *list_of_item_IDs* contains a set of *items*. Because most relational database systems do not support nested relational structures, the transactional database is usually either stored in a flat file in a format similar to the table in Figure 1.8 or unfolded into a standard relation in a format similar to the *items_sold* table in Figure 1.5. ■

As an analyst of *AllElectronics*, you may ask, "*Which items sold well together?*" This kind of *market basket data analysis* would enable you to bundle groups of items together as a strategy for boosting sales. For example, given the knowledge that printers are commonly purchased together with computers, you could offer certain printers at a steep discount (or even for free) to customers buying selected computers, in the hopes of selling more computers (which are often more expensive than printers). A traditional database system is not able to perform market basket data analysis. Fortunately, data mining on transactional data can do so by mining *frequent itemsets*, that is, sets

trans_ID	*list_of_item_IDs*
T100	I1, I3, I8, I16
T200	I2, I8
...	...

Figure 1.8 Fragment of a transactional database for sales at *AllElectronics*.

of items that are frequently sold together. The mining of such frequent patterns from transactional data is discussed in Chapters 6 and 7.

1.3.4 Other Kinds of Data

Besides relational database data, data warehouse data, and transaction data, there are many other kinds of data that have versatile forms and structures and rather different semantic meanings. Such kinds of data can be seen in many applications: time-related or sequence data (e.g., historical records, stock exchange data, and time-series and biological sequence data), data streams (e.g., video surveillance and sensor data, which are continuously transmitted), spatial data (e.g., maps), engineering design data (e.g., the design of buildings, system components, or integrated circuits), hypertext and multimedia data (including text, image, video, and audio data), graph and networked data (e.g., social and information networks), and the Web (a huge, widely distributed information repository made available by the Internet). These applications bring about new challenges, like how to handle data carrying special structures (e.g., sequences, trees, graphs, and networks) and specific semantics (such as ordering, image, audio and video contents, and connectivity), and how to mine patterns that carry rich structures and semantics.

Various kinds of knowledge can be mined from these kinds of data. Here, we list just a few. Regarding temporal data, for instance, we can mine banking data for changing trends, which may aid in the scheduling of bank tellers according to the volume of customer traffic. Stock exchange data can be mined to uncover trends that could help you plan investment strategies (e.g., the best time to purchase *AllElectronics* stock). We could mine computer network data streams to detect intrusions based on the anomaly of message flows, which may be discovered by clustering, dynamic construction of stream models or by comparing the current frequent patterns with those at a previous time. With spatial data, we may look for patterns that describe changes in metropolitan poverty rates based on city distances from major highways. The relationships among a set of spatial objects can be examined in order to discover which subsets of objects are spatially autocorrelated or associated. By mining text data, such as literature on data mining from the past ten years, we can identify the evolution of hot topics in the field. By mining user comments on products (which are often submitted as short text messages), we can assess customer sentiments and understand how well a product is embraced by a market. From multimedia data, we can mine images to identify objects and classify them by assigning semantic labels or tags. By mining video data of a hockey game, we can detect video sequences corresponding to goals. *Web mining* can help us learn about the distribution of information on the WWW in general, characterize and classify web pages, and uncover web dynamics and the association and other relationships among different web pages, users, communities, and web-based activities.

It is important to keep in mind that, in many applications, multiple types of data are present. For example, in web mining, there often exist text data and multimedia data (e.g., pictures and videos) on web pages, graph data like web graphs, and map data on some web sites. In bioinformatics, genomic sequences, biological networks, and

3-D spatial structures of genomes may coexist for certain biological objects. Mining multiple data sources of complex data often leads to fruitful findings due to the mutual enhancement and consolidation of such multiple sources. On the other hand, it is also challenging because of the difficulties in data cleaning and data integration, as well as the complex interactions among the multiple sources of such data.

While such data require sophisticated facilities for efficient storage, retrieval, and updating, they also provide fertile ground and raise challenging research and implementation issues for data mining. Data mining on such data is an advanced topic. The methods involved are extensions of the basic techniques presented in this book.

1.4 What Kinds of Patterns Can Be Mined?

We have observed various types of data and information repositories on which data mining can be performed. Let us now examine the kinds of patterns that can be mined.

There are a number of ***data mining functionalities***. These include characterization and discrimination (Section 1.4.1); the mining of frequent patterns, associations, and correlations (Section 1.4.2); classification and regression (Section 1.4.3); clustering analysis (Section 1.4.4); and outlier analysis (Section 1.4.5). Data mining functionalities are used to specify the kinds of patterns to be found in data mining tasks. In general, such tasks can be classified into two categories: **descriptive** and **predictive**. Descriptive mining tasks characterize properties of the data in a target data set. Predictive mining tasks perform induction on the current data in order to make predictions.

Data mining functionalities, and the kinds of patterns they can discover, are described below. In addition, Section 1.4.6 looks at what makes a pattern interesting. Interesting patterns represent *knowledge*.

1.4.1 Class/Concept Description: Characterization and Discrimination

Data entries can be associated with classes or concepts. For example, in the *AllElectronics* store, classes of items for sale include *computers* and *printers*, and concepts of customers include *bigSpenders* and *budgetSpenders*. It can be useful to describe individual classes and concepts in summarized, concise, and yet precise terms. Such descriptions of a class or a concept are called **class/concept descriptions**. These descriptions can be derived using (1) *data characterization*, by summarizing the data of the class under study (often called the **target class**) in general terms, or (2) *data discrimination*, by comparison of the target class with one or a set of comparative classes (often called the **contrasting classes**), or (3) both data characterization and discrimination.

Data characterization is a summarization of the general characteristics or features of a target class of data. The data corresponding to the user-specified class are typically collected by a query. For example, to study the characteristics of software products with sales that increased by 10% in the previous year, the data related to such products can be collected by executing an SQL query on the sales database.

There are several methods for effective data summarization and characterization. Simple data summaries based on statistical measures and plots are described in Chapter 2. The data cube-based OLAP roll-up operation (Section 1.3.2) can be used to perform user-controlled data summarization along a specified dimension. This process is further detailed in Chapters 4 and 5, which discuss data warehousing. An *attribute-oriented induction* technique can be used to perform data generalization and characterization without step-by-step user interaction. This technique is also described in Chapter 4.

The output of data characterization can be presented in various forms. Examples include **pie charts**, **bar charts**, **curves**, **multidimensional data cubes**, and **multidimensional tables**, including crosstabs. The resulting descriptions can also be presented as **generalized relations** or in rule form (called **characteristic rules**).

Example 1.5 **Data characterization.** A customer relationship manager at *AllElectronics* may order the following data mining task: *Summarize the characteristics of customers who spend more than $5000 a year at AllElectronics.* The result is a general profile of these customers, such as that they are 40 to 50 years old, employed, and have excellent credit ratings. The data mining system should allow the customer relationship manager to drill down on any dimension, such as on *occupation* to view these customers according to their type of employment. ■

Data discrimination is a comparison of the general features of the target class data objects against the general features of objects from one or multiple contrasting classes. The target and contrasting classes can be specified by a user, and the corresponding data objects can be retrieved through database queries. For example, a user may want to compare the general features of software products with sales that increased by 10% last year against those with sales that decreased by at least 30% during the same period. The methods used for data discrimination are similar to those used for data characterization.

"*How are discrimination descriptions output?*" The forms of output presentation are similar to those for characteristic descriptions, although discrimination descriptions should include comparative measures that help to distinguish between the target and contrasting classes. Discrimination descriptions expressed in the form of rules are referred to as **discriminant rules**.

Example 1.6 **Data discrimination.** A customer relationship manager at *AllElectronics* may want to compare two groups of customers—those who shop for computer products regularly (e.g., more than twice a month) and those who rarely shop for such products (e.g., less than three times a year). The resulting description provides a general comparative profile of these customers, such as that 80% of the customers who frequently purchase computer products are between 20 and 40 years old and have a university education, whereas 60% of the customers who infrequently buy such products are either seniors or youths, and have no university degree. Drilling down on a dimension like *occupation*, or adding a new dimension like *income_level*, may help to find even more discriminative features between the two classes. ■

Concept description, including characterization and discrimination, is described in Chapter 4.

1.4.2 Mining Frequent Patterns, Associations, and Correlations

Frequent patterns, as the name suggests, are patterns that occur frequently in data. There are many kinds of frequent patterns, including frequent itemsets, frequent subsequences (also known as sequential patterns), and frequent substructures. A *frequent itemset* typically refers to a set of items that often appear together in a transactional data set—for example, milk and bread, which are frequently bought together in grocery stores by many customers. A frequently occurring subsequence, such as the pattern that customers, tend to purchase first a laptop, followed by a digital camera, and then a memory card, is a (*frequent*) *sequential pattern*. A substructure can refer to different structural forms (e.g., graphs, trees, or lattices) that may be combined with itemsets or subsequences. If a substructure occurs frequently, it is called a (*frequent*) *structured pattern*. Mining frequent patterns leads to the discovery of interesting associations and correlations within data.

Example 1.7 **Association analysis.** Suppose that, as a marketing manager at *AllElectronics*, you want to know which items are frequently purchased together (i.e., within the same transaction). An example of such a rule, mined from the *AllElectronics* transactional database, is

$$buys(X, \text{“}computer\text{”}) \Rightarrow buys(X, \text{“}software\text{”}) \; [support = 1\%, confidence = 50\%],$$

where X is a variable representing a customer. A **confidence**, or certainty, of 50% means that if a customer buys a computer, there is a 50% chance that she will buy software as well. A 1% **support** means that 1% of all the transactions under analysis show that computer and software are purchased together. This association rule involves a single attribute or predicate (i.e., *buys*) that repeats. Association rules that contain a single predicate are referred to as **single-dimensional association rules**. Dropping the predicate notation, the rule can be written simply as “*computer* $\Rightarrow$ *software* [1%, 50%].”

Suppose, instead, that we are given the *AllElectronics* relational database related to purchases. A data mining system may find association rules like

$$age(X, \text{“20..29”}) \wedge income(X, \text{“40K..49K”}) \Rightarrow buys(X, \text{“laptop”})$$
$$[support = 2\%, confidence = 60\%].$$

The rule indicates that of the *AllElectronics* customers under study, 2% are 20 to 29 years old with an income of $40,000 to $49,000 and have purchased a laptop (computer) at *AllElectronics*. There is a 60% probability that a customer in this age and income group will purchase a laptop. Note that this is an association involving more than one attribute or predicate (i.e., *age*, *income*, and *buys*). Adopting the terminology used in multidimensional databases, where each attribute is referred to as a dimension, the above rule can be referred to as a **multidimensional association rule**. ■

Typically, association rules are discarded as uninteresting if they do not satisfy both a **minimum support threshold** and a **minimum confidence threshold**. Additional analysis can be performed to uncover interesting statistical **correlations** between associated attribute–value pairs.

Frequent itemset mining is a fundamental form of frequent pattern mining. The mining of frequent patterns, associations, and correlations is discussed in Chapters 6 and 7, where particular emphasis is placed on efficient algorithms for frequent itemset mining. Sequential pattern mining and structured pattern mining are considered advanced topics.

1.4.3 Classification and Regression for Predictive Analysis

Classification is the process of finding a **model** (or function) that describes and distinguishes data classes or concepts. The model are derived based on the analysis of a set of **training data** (i.e., data objects for which the class labels are known). The model is used to predict the class label of objects for which the the class label is unknown.

"How is the derived model presented?" The derived model may be represented in various forms, such as *classification rules* (i.e., *IF-THEN rules*), *decision trees*, *mathematical formulae*, or *neural networks* (Figure 1.9). A **decision tree** is a flowchart-like tree structure, where each node denotes a test on an attribute value, each branch represents an outcome of the test, and tree leaves represent classes or class distributions. Decision trees can easily

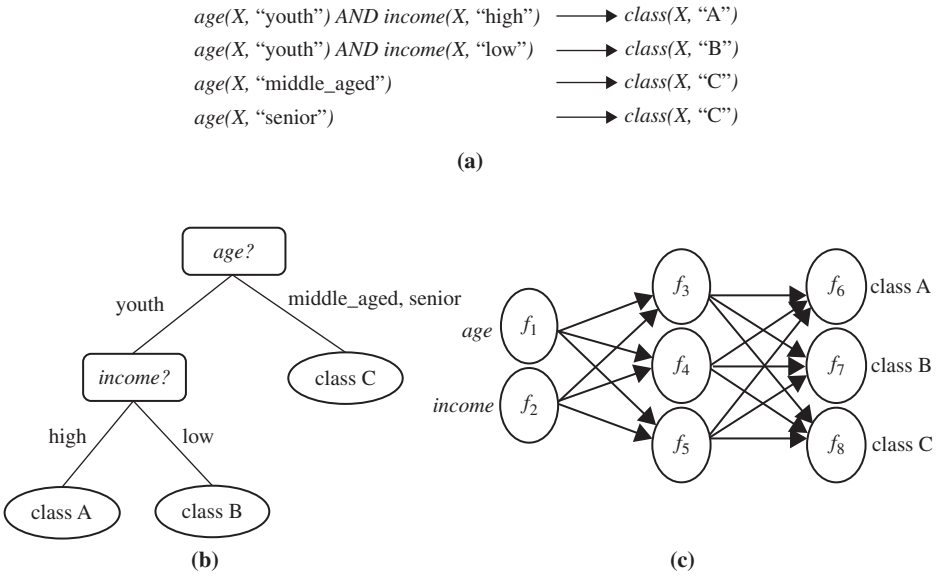


Figure 1.9 A classification model can be represented in various forms: (a) IF-THEN rules, (b) a decision tree, or (c) a neural network.

be converted to classification rules. A **neural network**, when used for classification, is typically a collection of neuron-like processing units with weighted connections between the units. There are many other methods for constructing classification models, such as naïve Bayesian classification, support vector machines, and *k*-nearest-neighbor classification.

Whereas classification predicts categorical (discrete, unordered) labels, **regression** models continuous-valued functions. That is, regression is used to predict missing or unavailable *numerical data values* rather than (discrete) class labels. The term *prediction* refers to both numeric prediction and class label prediction. **Regression analysis** is a statistical methodology that is most often used for numeric prediction, although other methods exist as well. Regression also encompasses the identification of distribution *trends* based on the available data.

Classification and regression may need to be preceded by **relevance analysis**, which attempts to identify attributes that are significantly relevant to the classification and regression process. Such attributes will be selected for the classification and regression process. Other attributes, which are irrelevant, can then be excluded from consideration.

Example 1.8 **Classification and regression.** Suppose as a sales manager of *AllElectronics* you want to classify a large set of items in the store, based on three kinds of responses to a sales campaign: *good response*, *mild response* and *no response*. You want to derive a model for each of these three classes based on the descriptive features of the items, such as *price*, *brand*, *place_made*, *type*, and *category*. The resulting classification should maximally distinguish each class from the others, presenting an organized picture of the data set.

Suppose that the resulting classification is expressed as a decision tree. The decision tree, for instance, may identify *price* as being the single factor that best distinguishes the three classes. The tree may reveal that, in addition to *price*, other features that help to further distinguish objects of each class from one another include *brand* and *place_made*. Such a decision tree may help you understand the impact of the given sales campaign and design a more effective campaign in the future.

Suppose instead, that rather than predicting categorical response labels for each store item, you would like to predict the amount of revenue that each item will generate during an upcoming sale at *AllElectronics*, based on the previous sales data. This is an example of regression analysis because the regression model constructed will predict a continuous function (or ordered value.) ■

Chapters 8 and 9 discuss classification in further detail. Regression analysis is beyond the scope of this book. Sources for further information are given in the bibliographic notes.

1.4.4 Cluster Analysis

Unlike classification and regression, which analyze class-labeled (training) data sets, **clustering** analyzes data objects without consulting class labels. In many cases, class-labeled data may simply not exist at the beginning. Clustering can be used to generate

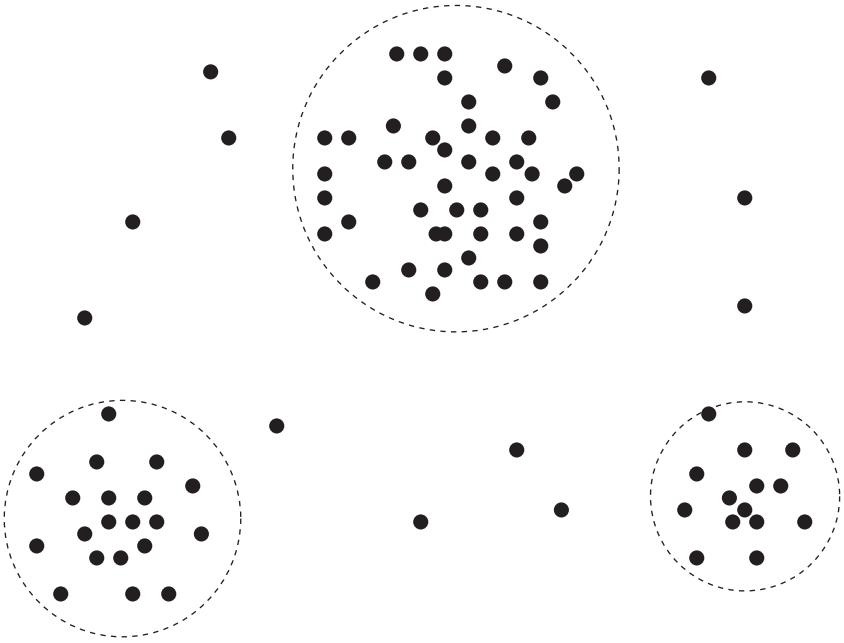


Figure 1.10 A 2-D plot of customer data with respect to customer locations in a city, showing three data clusters.

class labels for a group of data. The objects are clustered or grouped based on the principle of *maximizing the intraclass similarity and minimizing the interclass similarity*. That is, clusters of objects are formed so that objects within a cluster have high similarity in comparison to one another, but are rather dissimilar to objects in other clusters. Each cluster so formed can be viewed as a class of objects, from which rules can be derived. Clustering can also facilitate **taxonomy formation**, that is, the organization of observations into a hierarchy of classes that group similar events together.

Example 1.9 **Cluster analysis.** Cluster analysis can be performed on *AllElectronics* customer data to identify homogeneous subpopulations of customers. These clusters may represent individual target groups for marketing. Figure 1.10 shows a 2-D plot of customers with respect to customer locations in a city. Three clusters of data points are evident. ■

Cluster analysis forms the topic of Chapters 10 and 11.

1.4.5 Outlier Analysis

A data set may contain objects that do not comply with the general behavior or model of the data. These data objects are **outliers**. Many data mining methods discard outliers as noise or exceptions. However, in some applications (e.g., fraud detection) the rare

events can be more interesting than the more regularly occurring ones. The analysis of outlier data is referred to as **outlier analysis** or **anomaly mining**.

Outliers may be detected using statistical tests that assume a distribution or probability model for the data, or using distance measures where objects that are remote from any other cluster are considered outliers. Rather than using statistical or distance measures, density-based methods may identify outliers in a local region, although they look normal from a global statistical distribution view.

Example 1.10 **Outlier analysis.** Outlier analysis may uncover fraudulent usage of credit cards by detecting purchases of unusually large amounts for a given account number in comparison to regular charges incurred by the same account. Outlier values may also be detected with respect to the locations and types of purchase, or the purchase frequency. ■

Outlier analysis is discussed in Chapter 12.

1.4.6 Are All Patterns Interesting?

A data mining system has the potential to generate thousands or even millions of patterns, or rules.

You may ask, *"Are all of the patterns interesting?"* Typically, the answer is no—only a small fraction of the patterns potentially generated would actually be of interest to a given user.

This raises some serious questions for data mining. You may wonder, *"What makes a pattern interesting? Can a data mining system generate all of the interesting patterns? Or, Can the system generate only the interesting ones?"*

To answer the first question, a pattern is **interesting** if it is (1) *easily understood* by humans, (2) *valid* on new or test data with some degree of *certainty*, (3) potentially *useful*, and (4) *novel*. A pattern is also interesting if it validates a hypothesis that the user *sought to confirm*. An interesting pattern represents **knowledge**.

Several **objective measures of pattern interestingness** exist. These are based on the structure of discovered patterns and the statistics underlying them. An objective measure for association rules of the form $X \Rightarrow Y$ is rule **support**, representing the percentage of transactions from a transaction database that the given rule satisfies. This is taken to be the probability $P(X \cup Y)$, where $X \cup Y$ indicates that a transaction contains both X and Y, that is, the union of itemsets X and Y. Another objective measure for association rules is **confidence**, which assesses the degree of certainty of the detected association. This is taken to be the conditional probability $P(Y|X)$, that is, the probability that a transaction containing X also contains Y. More formally, support and confidence are defined as

$$support(X \Rightarrow Y) = P(X \cup Y),$$
$$confidence(X \Rightarrow Y) = P(Y|X).$$

In general, each interestingness measure is associated with a threshold, which may be controlled by the user. For example, rules that do not satisfy a confidence threshold of,

say, 50% can be considered uninteresting. Rules below the threshold likely reflect noise, exceptions, or minority cases and are probably of less value.

Other objective interestingness measures include *accuracy* and *coverage* for classification (IF-THEN) rules. In general terms, accuracy tells us the percentage of data that are correctly classified by a rule. Coverage is similar to support, in that it tells us the percentage of data to which a rule applies. Regarding understandability, we may use simple objective measures that assess the complexity or length in bits of the patterns mined.

Although objective measures help identify interesting patterns, they are often insufficient unless combined with subjective measures that reflect a particular user's needs and interests. For example, patterns describing the characteristics of customers who shop frequently at *AllElectronics* should be interesting to the marketing manager, but may be of little interest to other analysts studying the same database for patterns on employee performance. Furthermore, many patterns that are interesting by objective standards may represent common sense and, therefore, are actually uninteresting.

Subjective interestingness measures are based on user beliefs in the data. These measures find patterns interesting if the patterns are **unexpected** (contradicting a user's belief) or offer strategic information on which the user can act. In the latter case, such patterns are referred to as **actionable**. For example, patterns like "a large earthquake often follows a cluster of small quakes" may be highly actionable if users can act on the information to save lives. Patterns that are **expected** can be interesting if they confirm a hypothesis that the user wishes to validate or they resemble a user's hunch.

The second question—"*Can a data mining system generate* all *of the interesting patterns?*"—refers to the **completeness** of a data mining algorithm. It is often unrealistic and inefficient for data mining systems to generate all possible patterns. Instead, user-provided constraints and interestingness measures should be used to focus the search. For some mining tasks, such as association, this is often sufficient to ensure the completeness of the algorithm. Association rule mining is an example where the use of constraints and interestingness measures can ensure the completeness of mining. The methods involved are examined in detail in Chapter 6.

Finally, the third question—"*Can a data mining system generate only interesting patterns?*"—is an optimization problem in data mining. It is highly desirable for data mining systems to generate only interesting patterns. This would be efficient for users and data mining systems because neither would have to search through the patterns generated to identify the truly interesting ones. Progress has been made in this direction; however, such optimization remains a challenging issue in data mining.

Measures of pattern interestingness are essential for the efficient discovery of patterns by target users. Such measures can be used after the data mining step to rank the discovered patterns according to their interestingness, filtering out the uninteresting ones. More important, such measures can be used to guide and constrain the discovery process, improving the search efficiency by pruning away subsets of the pattern space that do not satisfy prespecified interestingness constraints. Examples of such a constraint-based mining process are described in Chapter 7 (with respect to pattern discovery) and Chapter 11 (with respect to clustering).

Methods to assess pattern interestingness, and their use to improve data mining efficiency, are discussed throughout the book with respect to each kind of pattern that can be mined.

1.5 Which Technologies Are Used?

As a highly application-driven domain, data mining has incorporated many techniques from other domains such as statistics, machine learning, pattern recognition, database and data warehouse systems, information retrieval, visualization, algorithms, high-performance computing, and many application domains (Figure 1.11). The interdisciplinary nature of data mining research and development contributes significantly to the success of data mining and its extensive applications. In this section, we give examples of several disciplines that strongly influence the development of data mining methods.

1.5.1 Statistics

Statistics studies the collection, analysis, interpretation or explanation, and presentation of data. Data mining has an inherent connection with statistics.

A **statistical model** is a set of mathematical functions that describe the behavior of the objects in a target class in terms of random variables and their associated probability distributions. Statistical models are widely used to model data and data classes. For example, in data mining tasks like data characterization and classification, statistical

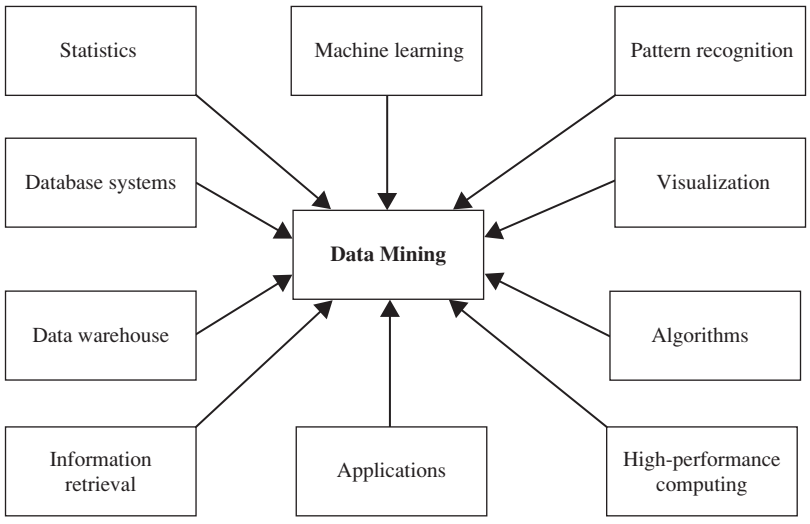


Figure 1.11 Data mining adopts techniques from many domains.

models of target classes can be built. In other words, such statistical models can be the outcome of a data mining task. Alternatively, data mining tasks can be built on top of statistical models. For example, we can use statistics to model noise and missing data values. Then, when mining patterns in a large data set, the data mining process can use the model to help identify and handle noisy or missing values in the data.

Statistics research develops tools for prediction and forecasting using data and statistical models. Statistical methods can be used to summarize or describe a collection of data. Basic **statistical descriptions** of data are introduced in Chapter 2. Statistics is useful for mining various patterns from data as well as for understanding the underlying mechanisms generating and affecting the patterns. **Inferential statistics** (or **predictive statistics**) models data in a way that accounts for randomness and uncertainty in the observations and is used to draw inferences about the process or population under investigation.

Statistical methods can also be used to verify data mining results. For example, after a classification or prediction model is mined, the model should be verified by statistical hypothesis testing. A **statistical hypothesis test** (sometimes called *confirmatory data analysis*) makes statistical decisions using experimental data. A result is called *statistically significant* if it is unlikely to have occurred by chance. If the classification or prediction model holds true, then the descriptive statistics of the model increases the soundness of the model.

Applying statistical methods in data mining is far from trivial. Often, a serious challenge is how to scale up a statistical method over a large data set. Many statistical methods have high complexity in computation. When such methods are applied on large data sets that are also distributed on multiple logical or physical sites, algorithms should be carefully designed and tuned to reduce the computational cost. This challenge becomes even tougher for online applications, such as online query suggestions in search engines, where data mining is required to continuously handle fast, real-time data streams.

1.5.2 Machine Learning

Machine learning investigates how computers can learn (or improve their performance) based on data. A main research area is for computer programs to *automatically* learn to recognize complex patterns and make intelligent decisions based on data. For example, a typical machine learning problem is to program a computer so that it can automatically recognize handwritten postal codes on mail after learning from a set of examples.

Machine learning is a fast-growing discipline. Here, we illustrate classic problems in machine learning that are highly related to data mining.

- **Supervised learning** is basically a synonym for classification. The supervision in the learning comes from the labeled examples in the training data set. For example, in the postal code recognition problem, a set of handwritten postal code images and their corresponding machine-readable translations are used as the training examples, which supervise the learning of the classification model.

- **Unsupervised learning** is essentially a synonym for clustering. The learning process is unsupervised since the input examples are not class labeled. Typically, we may use clustering to discover classes within the data. For example, an unsupervised learning method can take, as input, a set of images of handwritten digits. Suppose that it finds 10 clusters of data. These clusters may correspond to the 10 distinct digits of 0 to 9, respectively. However, since the training data are not labeled, the learned model cannot tell us the semantic meaning of the clusters found.
- **Semi-supervised learning** is a class of machine learning techniques that make use of both labeled and unlabeled examples when learning a model. In one approach, labeled examples are used to learn class models and unlabeled examples are used to refine the boundaries between classes. For a two-class problem, we can think of the set of examples belonging to one class as the *positive examples* and those belonging to the other class as the *negative examples*. In Figure 1.12, if we do not consider the unlabeled examples, the dashed line is the decision boundary that best partitions the positive examples from the negative examples. Using the unlabeled examples, we can refine the decision boundary to the solid line. Moreover, we can detect that the two positive examples at the top right corner, though labeled, are likely noise or outliers.
- **Active learning** is a machine learning approach that lets users play an active role in the learning process. An active learning approach can ask a user (e.g., a domain expert) to label an example, which may be from a set of unlabeled examples or synthesized by the learning program. The goal is to optimize the model quality by actively acquiring knowledge from human users, given a constraint on how many examples they can be asked to label.

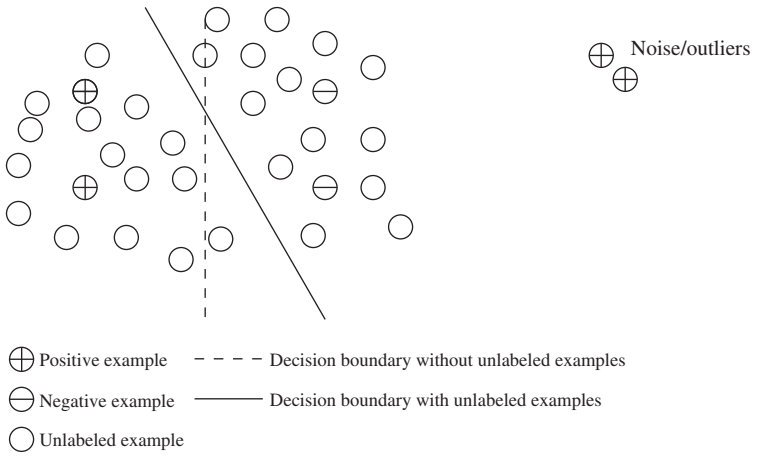


Figure 1.12 Semi-supervised learning.

You can see there are many similarities between data mining and machine learning. For classification and clustering tasks, machine learning research often focuses on the accuracy of the model. In addition to accuracy, data mining research places strong emphasis on the efficiency and scalability of mining methods on large data sets, as well as on ways to handle complex types of data and explore new, alternative methods.

1.5.3 Database Systems and Data Warehouses

Database systems research focuses on the creation, maintenance, and use of databases for organizations and end-users. Particularly, database systems researchers have established highly recognized principles in data models, query languages, query processing and optimization methods, data storage, and indexing and accessing methods. Database systems are often well known for their high scalability in processing very large, relatively structured data sets.

Many data mining tasks need to handle large data sets or even real-time, fast streaming data. Therefore, data mining can make good use of scalable database technologies to achieve high efficiency and scalability on large data sets. Moreover, data mining tasks can be used to extend the capability of existing database systems to satisfy advanced users' sophisticated data analysis requirements.

Recent database systems have built systematic data analysis capabilities on database data using data warehousing and data mining facilities. A **data warehouse** integrates data originating from multiple sources and various timeframes. It consolidates data in multidimensional space to form partially materialized data cubes. The data cube model not only facilitates OLAP in multidimensional databases but also promotes *multidimensional data mining* (see Section 1.3.2).

1.5.4 Information Retrieval

Information retrieval (**IR**) is the science of searching for documents or information in documents. Documents can be text or multimedia, and may reside on the Web. The differences between traditional information retrieval and database systems are twofold: Information retrieval assumes that (1) the data under search are unstructured; and (2) the queries are formed mainly by keywords, which do not have complex structures (unlike SQL queries in database systems).

The typical approaches in information retrieval adopt probabilistic models. For example, a text document can be regarded as a bag of words, that is, a multiset of words appearing in the document. The document's **language model** is the probability density function that generates the bag of words in the document. The similarity between two documents can be measured by the similarity between their corresponding language models.

Furthermore, a topic in a set of text documents can be modeled as a probability distribution over the vocabulary, which is called a **topic model**. A text document, which may involve one or multiple topics, can be regarded as a mixture of multiple topic models. By integrating information retrieval models and data mining techniques, we can find

the major topics in a collection of documents and, for each document in the collection, the major topics involved.

Increasingly large amounts of text and multimedia data have been accumulated and made available online due to the fast growth of the Web and applications such as digital libraries, digital governments, and health care information systems. Their effective search and analysis have raised many challenging issues in data mining. Therefore, text mining and multimedia data mining, integrated with information retrieval methods, have become increasingly important.

1.6 Which Kinds of Applications Are Targeted?

Where there are data, there are data mining applications

As a highly application-driven discipline, data mining has seen great successes in many applications. It is impossible to enumerate all applications where data mining plays a critical role. Presentations of data mining in knowledge-intensive application domains, such as bioinformatics and software engineering, require more in-depth treatment and are beyond the scope of this book. To demonstrate the importance of applications as a major dimension in data mining research and development, we briefly discuss two highly successful and popular application examples of data mining: *business intelligence* and *search engines*.

1.6.1 Business Intelligence

It is critical for businesses to acquire a better understanding of the commercial context of their organization, such as their customers, the market, supply and resources, and competitors. **Business intelligence** (**BI**) technologies provide historical, current, and predictive views of business operations. Examples include reporting, online analytical processing, business performance management, competitive intelligence, benchmarking, and predictive analytics.

"*How important is business intelligence?*" Without data mining, many businesses may not be able to perform effective market analysis, compare customer feedback on similar products, discover the strengths and weaknesses of their competitors, retain highly valuable customers, and make smart business decisions.

Clearly, data mining is the core of business intelligence. Online analytical processing tools in business intelligence rely on data warehousing and multidimensional data mining. Classification and prediction techniques are the core of predictive analytics in business intelligence, for which there are many applications in analyzing markets, supplies, and sales. Moreover, clustering plays a central role in customer relationship management, which groups customers based on their similarities. Using characterization mining techniques, we can better understand features of each customer group and develop customized customer reward programs.

1.6.2 Web Search Engines

A **Web search engine** is a specialized computer server that searches for information on the Web. The search results of a user query are often returned as a list (sometimes called *hits*). The hits may consist of web pages, images, and other types of files. Some search engines also search and return data available in public databases or open directories. Search engines differ from **web directories** in that web directories are maintained by human editors whereas search engines operate algorithmically or by a mixture of algorithmic and human input.

Web search engines are essentially very large data mining applications. Various data mining techniques are used in all aspects of search engines, ranging from *crawling*[5] (e.g., deciding which pages should be crawled and the crawling frequencies), indexing (e.g., selecting pages to be indexed and deciding to which extent the index should be constructed), and searching (e.g., deciding how pages should be ranked, which advertisements should be added, and how the search results can be personalized or made "context aware").

Search engines pose grand challenges to data mining. First, they have to handle a huge and ever-growing amount of data. Typically, such data cannot be processed using one or a few machines. Instead, search engines often need to use *computer clouds*, which consist of thousands or even hundreds of thousands of computers that collaboratively mine the huge amount of data. Scaling up data mining methods over computer clouds and large distributed data sets is an area for further research.

Second, Web search engines often have to deal with online data. A search engine may be able to afford constructing a model offline on huge data sets. To do this, it may construct a query classifier that assigns a search query to predefined categories based on the query topic (i.e., whether the search query "apple" is meant to retrieve information about a fruit or a brand of computers). Whether a model is constructed offline, the application of the model online must be fast enough to answer user queries in real time.

Another challenge is maintaining and incrementally updating a model on fast-growing data streams. For example, a query classifier may need to be incrementally maintained continuously since new queries keep emerging and predefined categories and the data distribution may change. Most of the existing model training methods are offline and static and thus cannot be used in such a scenario.

Third, Web search engines often have to deal with queries that are asked only a very small number of times. Suppose a search engine wants to provide *context-aware* query recommendations. That is, when a user poses a query, the search engine tries to infer the context of the query using the user's profile and his query history in order to return more customized answers within a small fraction of a second. However, although the total number of queries asked can be huge, most of the queries may be asked only once or a few times. Such severely skewed data are challenging for many data mining and machine learning methods.

[5] A Web crawler is a computer program that browses the Web in a methodical, automated manner.

1.7 Major Issues in Data Mining

Life is short but art is long. – Hippocrates

Data mining is a dynamic and fast-expanding field with great strengths. In this section, we briefly outline the major issues in data mining research, partitioning them into five groups: *mining methodology, user interaction, efficiency and scalability, diversity of data types*, and *data mining and society*. Many of these issues have been addressed in recent data mining research and development *to a certain extent* and are now considered *data mining requirements*; others are still at the research stage. The issues continue to stimulate further investigation and improvement in data mining.

1.7.1 Mining Methodology

Researchers have been vigorously developing new data mining methodologies. This involves the investigation of new kinds of knowledge, mining in multidimensional space, integrating methods from other disciplines, and the consideration of semantic ties among data objects. In addition, mining methodologies should consider issues such as data uncertainty, noise, and incompleteness. Some mining methods explore how user-specified measures can be used to assess the interestingness of discovered patterns as well as guide the discovery process. Let's have a look at these various aspects of mining methodology.

- *Mining various and new kinds of knowledge:* Data mining covers a wide spectrum of data analysis and knowledge discovery tasks, from data characterization and discrimination to association and correlation analysis, classification, regression, clustering, outlier analysis, sequence analysis, and trend and evolution analysis. These tasks may use the same database in different ways and require the development of numerous data mining techniques. Due to the diversity of applications, new mining tasks continue to emerge, making data mining a dynamic and fast-growing field. For example, for effective knowledge discovery in information networks, integrated clustering and ranking may lead to the discovery of high-quality clusters and object ranks in large networks.
- *Mining knowledge in multidimensional space:* When searching for knowledge in large data sets, we can explore the data in multidimensional space. That is, we can search for interesting patterns among combinations of dimensions (attributes) at varying levels of abstraction. Such mining is known as *(exploratory) multidimensional data mining*. In many cases, data can be aggregated or viewed as a multidimensional data cube. Mining knowledge in cube space can substantially enhance the power and flexibility of data mining.
- *Data mining—an interdisciplinary effort:* The power of data mining can be substantially enhanced by integrating new methods from multiple disciplines. For example,

to mine data with natural language text, it makes sense to fuse data mining methods with methods of information retrieval and natural language processing. As another example, consider the mining of software bugs in large programs. This form of mining, known as *bug mining*, benefits from the incorporation of software engineering knowledge into the data mining process.

- *Boosting the power of discovery in a networked environment:* Most data objects reside in a linked or interconnected environment, whether it be the Web, database relations, files, or documents. Semantic links across multiple data objects can be used to advantage in data mining. Knowledge derived in one set of objects can be used to boost the discovery of knowledge in a "related" or semantically linked set of objects.
- *Handling uncertainty, noise, or incompleteness of data:* Data often contain noise, errors, exceptions, or uncertainty, or are incomplete. Errors and noise may confuse the data mining process, leading to the derivation of erroneous patterns. Data cleaning, data preprocessing, outlier detection and removal, and uncertainty reasoning are examples of techniques that need to be integrated with the data mining process.
- *Pattern evaluation and pattern- or constraint-guided mining:* Not all the patterns generated by data mining processes are interesting. What makes a pattern interesting may vary from user to user. Therefore, techniques are needed to assess the interestingness of discovered patterns based on subjective measures. These estimate the value of patterns with respect to a given user class, based on user beliefs or expectations. Moreover, by using interestingness measures or user-specified constraints to *guide* the discovery process, we may generate more interesting patterns and reduce the search space.

1.7.2 User Interaction

The user plays an important role in the data mining process. Interesting areas of research include *how to interact with a data mining system*, *how to incorporate a user's background knowledge in mining*, and *how to visualize and comprehend data mining results*. We introduce each of these here.

- *Interactive mining:* The data mining process should be highly *interactive*. Thus, it is important to build flexible user interfaces and an exploratory mining environment, facilitating the user's interaction with the system. A user may like to first sample a set of data, explore general characteristics of the data, and estimate potential mining results. Interactive mining should allow users to dynamically change the focus of a search, to refine mining requests based on returned results, and to drill, dice, and pivot through the data and knowledge space interactively, dynamically exploring "cube space" while mining.
- *Incorporation of background knowledge:* Background knowledge, constraints, rules, and other information regarding the domain under study should be incorporated

into the knowledge discovery process. Such knowledge can be used for pattern evaluation as well as to guide the search toward interesting patterns.

- *Ad hoc data mining and data mining query languages:* Query languages (e.g., SQL) have played an important role in flexible searching because they allow users to pose ad hoc queries. Similarly, high-level data mining query languages or other high-level flexible user interfaces will give users the freedom to define ad hoc data mining tasks. This should facilitate specification of the relevant sets of data for analysis, the domain knowledge, the kinds of knowledge to be mined, and the conditions and constraints to be enforced on the discovered patterns. Optimization of the processing of such flexible mining requests is another promising area of study.
- *Presentation and visualization of data mining results:* How can a data mining system present data mining results, vividly and flexibly, so that the discovered knowledge can be easily understood and directly usable by humans? This is especially crucial if the data mining process is interactive. It requires the system to adopt expressive knowledge representations, user-friendly interfaces, and visualization techniques.

1.7.3 Efficiency and Scalability

Efficiency and scalability are always considered when comparing data mining algorithms. As data amounts continue to multiply, these two factors are especially critical.

- *Efficiency and scalability of data mining algorithms:* Data mining algorithms must be efficient and scalable in order to effectively extract information from huge amounts of data in many data repositories or in dynamic data streams. In other words, the running time of a data mining algorithm must be predictable, short, and acceptable by applications. *Efficiency, scalability, performance, optimization*, and the ability to execute in *real time* are key criteria that drive the development of many new data mining algorithms.
- *Parallel, distributed, and incremental mining algorithms:* The humongous size of many data sets, the wide distribution of data, and the computational complexity of some data mining methods are factors that motivate the development of **parallel and distributed data-intensive mining algorithms**. Such algorithms first partition the data into "pieces." Each piece is processed, in parallel, by searching for patterns. The parallel processes may interact with one another. The patterns from each partition are eventually merged.

  Cloud computing and *cluster computing*, which use computers in a distributed and collaborative way to tackle very large-scale computational tasks, are also active research themes in parallel data mining. In addition, the high cost of some data mining processes and the incremental nature of input promote **incremental** data mining, which incorporates new data updates without having to mine the entire data "from scratch." Such methods perform knowledge modification incrementally to amend and strengthen what was previously discovered.

1.7.4 Diversity of Database Types

The wide diversity of database types brings about challenges to data mining. These include

- *Handling complex types of data:* Diverse applications generate a wide spectrum of new data types, from structured data such as relational and data warehouse data to semi-structured and unstructured data; from stable data repositories to dynamic data streams; from simple data objects to temporal data, biological sequences, sensor data, spatial data, hypertext data, multimedia data, software program code, Web data, and social network data. It is unrealistic to expect one data mining system to mine all kinds of data, given the diversity of data types and the different goals of data mining. Domain- or application-dedicated data mining systems are being constructed for in-depth mining of specific kinds of data. The construction of effective and efficient data mining tools for diverse applications remains a challenging and active area of research.
- *Mining dynamic, networked, and global data repositories:* Multiple sources of data are connected by the Internet and various kinds of networks, forming gigantic, distributed, and heterogeneous global information systems and networks. The discovery of knowledge from different sources of structured, semi-structured, or unstructured yet interconnected data with diverse data semantics poses great challenges to data mining. Mining such gigantic, interconnected information networks may help disclose many more patterns and knowledge in heterogeneous data sets than can be discovered from a small set of isolated data repositories. Web mining, multisource data mining, and information network mining have become challenging and fast-evolving data mining fields.

1.7.5 Data Mining and Society

How does data mining impact society? What steps can data mining take to preserve the privacy of individuals? Do we use data mining in our daily lives without even knowing that we do? These questions raise the following issues:

- *Social impacts of data mining:* With data mining penetrating our everyday lives, it is important to study the impact of data mining on society. How can we use data mining technology to benefit society? How can we guard against its misuse? The improper disclosure or use of data and the potential violation of individual privacy and data protection rights are areas of concern that need to be addressed.
- *Privacy-preserving data mining:* Data mining will help scientific discovery, business management, economy recovery, and security protection (e.g., the real-time discovery of intruders and cyberattacks). However, it poses the risk of disclosing an individual's personal information. Studies on privacy-preserving data publishing and data mining are ongoing. The philosophy is to observe data sensitivity and preserve people's privacy while performing successful data mining.

- *Invisible data mining:* We cannot expect everyone in society to learn and master data mining techniques. More and more systems should have data mining functions built within so that people can perform data mining or use data mining results simply by mouse clicking, without any knowledge of data mining algorithms. Intelligent search engines and Internet-based stores perform such *invisible data mining* by incorporating data mining into their components to improve their functionality and performance. This is done often unbeknownst to the user. For example, when purchasing items online, users may be unaware that the store is likely collecting data on the buying patterns of its customers, which may be used to recommend other items for purchase in the future.

These issues and many additional ones relating to the research, development, and application of data mining are discussed throughout the book.

1.8 Summary

- *Necessity is the mother of invention.* With the mounting growth of data in every application, data mining meets the imminent need for effective, scalable, and flexible data analysis in our society. Data mining can be considered as a natural evolution of information technology and a confluence of several related disciplines and application domains.
- **Data mining** is the process of discovering interesting patterns from massive amounts of data. As a *knowledge discovery process*, it typically involves data cleaning, data integration, data selection, data transformation, pattern discovery, pattern evaluation, and knowledge presentation.
- A pattern is *interesting* if it is valid on test data with some degree of certainty, novel, potentially useful (e.g., can be acted on or validates a hunch about which the user was curious), and easily understood by humans. Interesting patterns represent **knowledge**. Measures of **pattern interestingness**, either *objective* or *subjective*, can be used to guide the discovery process.
- We present a **multidimensional view** of data mining. The major dimensions are **data**, **knowledge**, **technologies**, and **applications**.
- Data mining can be conducted on any kind of **data** as long as the data are meaningful for a target application, such as database data, data warehouse data, transactional data, and advanced data types. Advanced data types include time-related or sequence data, data streams, spatial and spatiotemporal data, text and multimedia data, graph and networked data, and Web data.
- A **data warehouse** is a repository for long-term storage of data from multiple sources, organized so as to facilitate management decision making. The data are stored under a unified schema and are typically summarized. Data warehouse systems provide multidimensional data analysis capabilities, collectively referred to as **online analytical processing**.

- **Multidimensional data mining** (also called **exploratory multidimensional data mining**) integrates core data mining techniques with OLAP-based multidimensional analysis. It searches for interesting patterns among multiple combinations of dimensions (attributes) at varying levels of abstraction, thereby exploring multidimensional data space.
- **Data mining functionalities** are used to specify the kinds of patterns or **knowledge** to be found in data mining tasks. The functionalities include characterization and discrimination; the mining of frequent patterns, associations, and correlations; classification and regression; cluster analysis; and outlier detection. As new types of data, new applications, and new analysis demands continue to emerge, there is no doubt we will see more and more novel data mining tasks in the future.
- Data mining, as a highly application-driven domain, has incorporated **technologies** from many other domains. These include statistics, machine learning, database and data warehouse systems, and information retrieval. The **interdisciplinary nature of data mining research and development** contributes significantly to the success of data mining and its extensive applications.
- Data mining has many successful **applications**, such as business intelligence, Web search, bioinformatics, health informatics, finance, digital libraries, and digital governments.
- There are many challenging **issues in data mining research**. Areas include mining methodology, user interaction, efficiency and scalability, and dealing with diverse data types. Data mining research has strongly impacted society and will continue to do so in the future.

1.9 Exercises

1.1 What is *data mining*? In your answer, address the following:

(a) Is it another hype?

(b) Is it a simple transformation or application of technology developed from *databases*, *statistics*, *machine learning*, and *pattern recognition*?

(c) We have presented a view that data mining is the result of the evolution of *database technology*. Do you think that data mining is also the result of the evolution of *machine learning research*? Can you present such views based on the historical progress of this discipline? Address the same for the fields of *statistics* and *pattern recognition*.

(d) Describe the steps involved in data mining when viewed as a process of knowledge discovery.

1.2 How is a *data warehouse* different from a *database*? How are they similar?

1.3 Define each of the following *data mining functionalities*: characterization, discrimination, association and correlation analysis, classification, regression, clustering, and

outlier analysis. Give examples of each data mining functionality, using a real-life database that you are familiar with.

1.4 Present an example where data mining is crucial to the success of a business. What *data mining functionalities* does this business need (e.g., think of the kinds of patterns that could be mined)? Can such patterns be generated alternatively by data query processing or simple statistical analysis?

1.5 Explain the difference and similarity between discrimination and classification, between characterization and clustering, and between classification and regression.

1.6 Based on your observations, describe another possible kind of knowledge that needs to be discovered by data mining methods but has not been listed in this chapter. Does it require a mining methodology that is quite different from those outlined in this chapter?

1.7 *Outliers* are often discarded as noise. However, one person's garbage could be another's treasure. For example, exceptions in credit card transactions can help us detect the fraudulent use of credit cards. Using fraudulence detection as an example, propose two methods that can be used to detect outliers and discuss which one is more reliable.

1.8 Describe three challenges to data mining regarding *data mining methodology* and *user interaction issues.*

1.9 What are the major challenges of mining a huge amount of data (e.g., billions of tuples) in comparison with mining a small amount of data (e.g., data set of a few hundred tuple)?

1.10 Outline the major research challenges of data mining in one specific application domain, such as stream/sensor data analysis, spatiotemporal data analysis, or bioinformatics.

1.10 Bibliographic Notes

The book *Knowledge Discovery in Databases*, edited by Piatetsky-Shapiro and Frawley [P-SF91], is an early collection of research papers on knowledge discovery from data. The book *Advances in Knowledge Discovery and Data Mining*, edited by Fayyad, Piatetsky-Shapiro, Smyth, and Uthurusamy [FPSS+96], is a collection of later research results on knowledge discovery and data mining. There have been many data mining books published in recent years, including *The Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman [HTF09]; *Introduction to Data Mining* by Tan, Steinbach, and Kumar [TSK05]; *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations* by Witten, Frank, and Hall [WFH11]; *Predictive Data Mining* by Weiss and Indurkhya [WI98]; *Mastering Data Mining: The Art and Science of Customer Relationship Management* by Berry and Linoff [BL99]; *Principles of Data Mining (Adaptive Computation and Machine Learning)* by Hand, Mannila, and Smyth [HMS01]; *Mining the Web: Discovering Knowledge from Hypertext Data* by Chakrabarti [Cha03a]; *Web Data Mining: Exploring Hyperlinks, Contents, and Usage*

Data by Liu [Liu06]; *Data Mining: Introductory and Advanced Topics* by Dunham [Dun03]; and *Data Mining: Multimedia, Soft Computing, and Bioinformatics* by Mitra and Acharya [MA03].

There are also books that contain collections of papers or chapters on particular aspects of knowledge discovery—for example, *Relational Data Mining* edited by Dzeroski and Lavrac [De01]; *Mining Graph Data* edited by Cook and Holder [CH07]; *Data Streams: Models and Algorithms* edited by Aggarwal [Agg06]; *Next Generation of Data Mining* edited by Kargupta, Han, Yu, et al. [KHY+08]; *Multimedia Data Mining: A Systematic Introduction to Concepts and Theory* edited by Z. Zhang and R. Zhang [ZZ09]; *Geographic Data Mining and Knowledge Discovery* edited by Miller and Han [MH09]; and *Link Mining: Models, Algorithms and Applications* edited by Yu, Han, and Faloutsos [YHF10]. There are many tutorial notes on data mining in major databases, data mining, machine learning, statistics, and Web technology conferences.

KDNuggets is a regular electronic newsletter containing information relevant to knowledge discovery and data mining, moderated by Piatetsky-Shapiro since 1991. The Internet site *KDNuggets* (*www.kdnuggets.com*) contains a good collection of KDD-related information.

The data mining community started its first international conference on knowledge discovery and data mining in 1995. The conference evolved from the four international workshops on knowledge discovery in databases, held from 1989 to 1994. ACM-SIGKDD, a Special Interest Group on Knowledge Discovery in Databases was set up under ACM in 1998 and has been organizing the international conferences on knowledge discovery and data mining since 1999. IEEE Computer Science Society has organized its annual data mining conference, International Conference on Data Mining (ICDM), since 2001. SIAM (Society on Industrial and Applied Mathematics) has organized its annual data mining conference, SIAM Data Mining Conference (SDM), since 2002. A dedicated journal, *Data Mining and Knowledge Discovery*, published by Kluwers Publishers, has been available since 1997. An ACM journal, *ACM Transactions on Knowledge Discovery from Data*, published its first volume in 2007.

ACM-SIGKDD also publishes a bi-annual newsletter, *SIGKDD Explorations*. There are a few other international or regional conferences on data mining, such as the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), and the International Conference on Data Warehousing and Knowledge Discovery (DaWaK).

Research in data mining has also been published in books, conferences, and journals on databases, statistics, machine learning, and data visualization. References to such sources are listed at the end of the book.

Popular textbooks on database systems include *Database Systems: The Complete Book* by Garcia-Molina, Ullman, and Widom [GMUW08]; *Database Management Systems* by Ramakrishnan and Gehrke [RG03]; *Database System Concepts* by Silberschatz, Korth, and Sudarshan [SKS10]; and *Fundamentals of Database Systems* by Elmasri and Navathe [EN10]. For an edited collection of seminal articles on database systems, see *Readings in Database Systems* by Hellerstein and Stonebraker [HS05].

There are also many books on data warehouse technology, systems, and applications, such as *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling* by Kimball and Ross [KR02]; *The Data Warehouse Lifecycle Toolkit* by Kimball, Ross, Thornthwaite, and Mundy [KRTM08]; *Mastering Data Warehouse Design: Relational and Dimensional Techniques* by Imhoff, Galemmo, and Geiger [IGG03]; and *Building the Data Warehouse* by Inmon [Inm96]. A set of research papers on materialized views and data warehouse implementations were collected in *Materialized Views: Techniques, Implementations, and Applications* by Gupta and Mumick [GM99]. Chaudhuri and Dayal [CD97] present an early comprehensive overview of data warehouse technology.

Research results relating to data mining and data warehousing have been published in the proceedings of many international database conferences, including the ACM-SIGMOD International Conference on Management of Data (SIGMOD), the International Conference on Very Large Data Bases (VLDB), the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), the International Conference on Data Engineering (ICDE), the International Conference on Extending Database Technology (EDBT), the International Conference on Database Theory (ICDT), the International Conference on Information and Knowledge Management (CIKM), the International Conference on Database and Expert Systems Applications (DEXA), and the International Symposium on Database Systems for Advanced Applications (DASFAA). Research in data mining is also published in major database journals, such as *IEEE Transactions on Knowledge and Data Engineering (TKDE), ACM Transactions on Database Systems (TODS), Information Systems, The VLDB Journal, Data and Knowledge Engineering, International Journal of Intelligent Information Systems (JIIS)*, and *Knowledge and Information Systems (KAIS)*.

Many effective data mining methods have been developed by statisticians and introduced in a rich set of textbooks. An overview of classification from a statistical pattern recognition perspective can be found in *Pattern Classification* by Duda, Hart, and Stork [DHS01]. There are also many textbooks covering regression and other topics in statistical analysis, such as *Mathematical Statistics: Basic Ideas and Selected Topics* by Bickel and Doksum [BD01]; *The Statistical Sleuth: A Course in Methods of Data Analysis* by Ramsey and Schafer [RS01]; *Applied Linear Statistical Models* by Neter, Kutner, Nachtsheim, and Wasserman [NKNW96]; *An Introduction to Generalized Linear Models* by Dobson [Dob90]; *Applied Statistical Time Series Analysis* by Shumway [Shu88]; and *Applied Multivariate Statistical Analysis* by Johnson and Wichern [JW92].

Research in statistics is published in the proceedings of several major statistical conferences, including Joint Statistical Meetings, International Conference of the Royal Statistical Society and Symposium on the Interface: Computing Science and Statistics. Other sources of publication include the *Journal of the Royal Statistical Society*, *The Annals of Statistics*, the *Journal of American Statistical Association*, *Technometrics*, and *Biometrika*.

Textbooks and reference books on machine learning and pattern recognition include *Machine Learning* by Mitchell [Mit97]; *Pattern Recognition and Machine Learning* by Bishop [Bis06]; *Pattern Recognition* by Theodoridis and Koutroumbas [TK08]; *Introduction to Machine Learning* by Alpaydin [Alp11]; *Probabilistic Graphical Models: Principles*

and Techniques by Koller and Friedman [KF09]; and *Machine Learning: An Algorithmic Perspective* by Marsland [Mar09]. For an edited collection of seminal articles on machine learning, see *Machine Learning, An Artificial Intelligence Approach*, Volumes 1 through 4, edited by Michalski et al. [MCM83, MCM86, KM90, MT94], and *Readings in Machine Learning* by Shavlik and Dietterich [SD90].

Machine learning and pattern recognition research is published in the proceedings of several major machine learning, artificial intelligence, and pattern recognition conferences, including the International Conference on Machine Learning (ML), the ACM Conference on Computational Learning Theory (COLT), the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), the International Conference on Pattern Recognition (ICPR), the International Joint Conference on Artificial Intelligence (IJCAI), and the American Association of Artificial Intelligence Conference (AAAI). Other sources of publication include major machine learning, artificial intelligence, pattern recognition, and knowledge system journals, some of which have been mentioned before. Others include *Machine Learning (ML), Pattern Recognition (PR), Artificial Intelligence Journal (AI)*, *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, and *Cognitive Science*.

Textbooks and reference books on information retrieval include *Introduction to Information Retrieval* by Manning, Raghavan, and Schutz [MRS08]; *Information Retrieval: Implementing and Evaluating Search Engines* by Büttcher, Clarke, and Cormack [BCC10]; *Search Engines: Information Retrieval in Practice* by Croft, Metzler, and Strohman [CMS09]; *Modern Information Retrieval: The Concepts and Technology Behind Search* by Baeza-Yates and Ribeiro-Neto [BYRN11]; and *Information Retrieval: Algorithms and Heuristics* by Grossman and Frieder [GR04].

Information retrieval research is published in the proceedings of several information retrieval and Web search and mining conferences, including the International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR), the International World Wide Web Conference (WWW), the ACM International Conference on Web Search and Data Mining (WSDM), the ACM Conference on Information and Knowledge Management (CIKM), the European Conference on Information Retrieval (ECIR), the Text Retrieval Conference (TREC), and the ACM/IEEE Joint Conference on Digital Libraries (JCDL). Other sources of publication include major information retrieval, information systems, and Web journals, such as *Journal of Information Retrieval, ACM Transactions on Information Systems (TOIS)*, *Information Processing and Management*, *Knowledge and Information Systems (KAIS)*, and *IEEE Transactions on Knowledge and Data Engineering (TKDE)*.

2 Getting to Know Your Data

It's tempting to jump straight into mining, but first, we need to get the data ready. This involves having a closer look at attributes and data values. Real-world data are typically noisy, enormous in volume (often several gigabytes or more), and may originate from a hodgepodge of heterogenous sources. This chapter is about getting familiar with your data. Knowledge about your data is useful for data preprocessing (see Chapter 3), the first major task of the data mining process. You will want to know the following: What are the types of *attributes* or fields that make up your data? What kind of values does each attribute have? Which attributes are discrete, and which are continuous-valued? What do the data *look like*? How are the values distributed? Are there ways we can visualize the data to get a better sense of it all? Can we spot any outliers? Can we measure the similarity of some data objects with respect to others? Gaining such insight into the data will help with the subsequent analysis.

"So what can we learn about our data that's helpful in data preprocessing?" We begin in Section 2.1 by studying the various attribute types. These include nominal attributes, binary attributes, ordinal attributes, and numeric attributes. Basic *statistical descriptions* can be used to learn more about each attribute's values, as described in Section 2.2. Given a *temperature* attribute, for example, we can determine its **mean** (average value), **median** (middle value), and **mode** (most common value). These are **measures of central tendency**, which give us an idea of the "middle" or center of distribution.

Knowing such basic statistics regarding each attribute makes it easier to fill in missing values, smooth noisy values, and spot outliers during data preprocessing. Knowledge of the attributes and attribute values can also help in fixing inconsistencies incurred during data integration. Plotting the measures of central tendency shows us if the data are symmetric or skewed. Quantile plots, histograms, and scatter plots are other graphic displays of basic statistical descriptions. These can all be useful during data preprocessing and can provide insight into areas for mining.

The field of data visualization provides many additional techniques for viewing data through graphical means. These can help identify relations, trends, and biases "hidden" in unstructured data sets. Techniques may be as simple as scatter-plot matrices (where

two attributes are mapped onto a 2-D grid) to more sophisticated methods such as treemaps (where a hierarchical partitioning of the screen is displayed based on the attribute values). Data visualization techniques are described in Section 2.3.

Finally, we may want to examine how similar (or dissimilar) data objects are. For example, suppose we have a database where the data objects are patients, described by their symptoms. We may want to find the similarity or dissimilarity between individual patients. Such information can allow us to find clusters of like patients within the data set. The similarity/dissimilarity between objects may also be used to detect outliers in the data, or to perform nearest-neighbor classification. (Clustering is the topic of Chapters 10 and 11, while nearest-neighbor classification is discussed in Chapter 9.) There are many measures for assessing similarity and dissimilarity. In general, such measures are referred to as proximity measures. Think of the proximity of two objects as a function of the *distance* between their attribute values, although proximity can also be calculated based on probabilities rather than actual distance. Measures of data proximity are described in Section 2.4.

In summary, by the end of this chapter, you will know the different attribute types and basic statistical measures to describe the central tendency and dispersion (spread) of attribute data. You will also know techniques to visualize attribute distributions and how to compute the similarity or dissimilarity between objects.

2.1 Data Objects and Attribute Types

Data sets are made up of data objects. A **data object** represents an entity—in a sales database, the objects may be customers, store items, and sales; in a medical database, the objects may be patients; in a university database, the objects may be students, professors, and courses. Data objects are typically described by attributes. Data objects can also be referred to as *samples, examples, instances, data points*, or *objects*. If the data objects are stored in a database, they are *data tuples*. That is, the rows of a database correspond to the data objects, and the columns correspond to the attributes. In this section, we define attributes and look at the various attribute types.

2.1.1 What Is an Attribute?

An **attribute** is a data field, representing a characteristic or feature of a data object. The nouns *attribute*, *dimension*, *feature*, and *variable* are often used interchangeably in the literature. The term *dimension* is commonly used in data warehousing. Machine learning literature tends to use the term *feature*, while statisticians prefer the term *variable*. Data mining and database professionals commonly use the term *attribute*, and we do here as well. Attributes describing a customer object can include, for example, *customer_ID*, *name*, and *address*. Observed values for a given attribute are known as *observations*. A set of attributes used to describe a given object is called an *attribute vector* (or *feature vector*). The distribution of data involving one attribute (or variable) is called *univariate*. A *bivariate* distribution involves two attributes, and so on.

The **type** of an attribute is determined by the set of possible values—nominal, binary, ordinal, or numeric—the attribute can have. In the following subsections, we introduce each type.

2.1.2 Nominal Attributes

Nominal means "relating to names." The values of a **nominal attribute** are symbols or *names of things*. Each value represents some kind of category, code, or state, and so nominal attributes are also referred to as **categorical**. The values do not have any meaningful order. In computer science, the values are also known as *enumerations*.

Example 2.1 **Nominal attributes.** Suppose that *hair_color* and *marital_status* are two attributes describing *person* objects. In our application, possible values for *hair_color* are *black*, *brown*, *blond*, *red*, *auburn*, *gray*, and *white*. The attribute *marital_status* can take on the values *single, married, divorced*, and *widowed*. Both *hair_color* and *marital_status* are nominal attributes. Another example of a nominal attribute is *occupation*, with the values *teacher, dentist, programmer, farmer*, and so on. ■

Although we said that the values of a nominal attribute are symbols or "names of things," it is possible to represent such symbols or "names" with numbers. With *hair_color*, for instance, we can assign a code of 0 for *black*, 1 for *brown*, and so on. Another example is *customor_ID*, with possible values that are all numeric. However, in such cases, the numbers are not intended to be used quantitatively. That is, mathematical operations on values of nominal attributes are not meaningful. It makes no sense to subtract one customer ID number from another, unlike, say, subtracting an age value from another (where *age* is a numeric attribute). Even though a nominal attribute may have integers as values, it is not considered a numeric attribute because the integers are not meant to be used quantitatively. We will say more on numeric attributes in Section 2.1.5.

Because nominal attribute values do not have any meaningful order about them and are not quantitative, it makes no sense to find the mean (average) value or median (middle) value for such an attribute, given a set of objects. One thing that is of interest, however, is the attribute's most commonly occurring value. This value, known as the *mode*, is one of the measures of central tendency. You will learn about measures of central tendency in Section 2.2.

2.1.3 Binary Attributes

A **binary attribute** is a nominal attribute with only two categories or states: 0 or 1, where 0 typically means that the attribute is absent, and 1 means that it is present. Binary attributes are referred to as **Boolean** if the two states correspond to *true* and *false*.

Example 2.2 **Binary attributes.** Given the attribute *smoker* describing a *patient* object, 1 indicates that the patient smokes, while 0 indicates that the patient does not. Similarly, suppose

the patient undergoes a medical test that has two possible outcomes. The attribute *medical_test* is binary, where a value of 1 means the result of the test for the patient is positive, while 0 means the result is negative. ■

A binary attribute is **symmetric** if both of its states are equally valuable and carry the same weight; that is, there is no preference on which outcome should be coded as 0 or 1. One such example could be the attribute *gender* having the states *male* and *female*.

A binary attribute is **asymmetric** if the outcomes of the states are not equally important, such as the *positive* and *negative* outcomes of a medical test for HIV. By convention, we code the most important outcome, which is usually the rarest one, by 1 (e.g., *HIV positive*) and the other by 0 (e.g., *HIV negative*).

2.1.4 Ordinal Attributes

An **ordinal attribute** is an attribute with possible values that have a meaningful order or *ranking* among them, but the magnitude between successive values is not known.

Example 2.3 **Ordinal attributes.** Suppose that *drink_size* corresponds to the size of drinks available at a fast-food restaurant. This nominal attribute has three possible values: *small, medium*, and *large*. The values have a meaningful sequence (which corresponds to increasing drink size); however, we cannot tell from the values *how much* bigger, say, a medium is than a large. Other examples of ordinal attributes include *grade* (e.g., *A*+, *A*, *A*−, *B*+, and so on) and *professional_rank*. Professional ranks can be enumerated in a sequential order: for example, *assistant*, *associate*, and *full* for professors, and *private, private first class, specialist, corporal, and sergeant* for army ranks.

Ordinal attributes are useful for registering subjective assessments of qualities that cannot be measured objectively; thus ordinal attributes are often used in surveys for ratings. In one survey, participants were asked to rate how satisfied they were as customers. Customer satisfaction had the following ordinal categories: *0: very dissatisfied, 1: somewhat dissatisfied, 2: neutral, 3: satisfied*, and *4: very satisfied.* ■

Ordinal attributes may also be obtained from the discretization of numeric quantities by splitting the value range into a finite number of ordered categories as described in Chapter 3 on data reduction.

The central tendency of an ordinal attribute can be represented by its mode and its median (the middle value in an ordered sequence), but the mean cannot be defined.

Note that nominal, binary, and ordinal attributes are *qualitative*. That is, they *describe* a feature of an object without giving an actual size or quantity. The values of such qualitative attributes are typically words representing categories. If integers are used, they represent computer codes for the categories, as opposed to measurable quantities (e.g., 0 for *small* drink size, 1 for *medium*, and 2 for *large*). In the following subsection we look at numeric attributes, which provide *quantitative* measurements of an object.

2.1.5 Numeric Attributes

A **numeric attribute** is *quantitative*; that is, it is a measurable quantity, represented in integer or real values. Numeric attributes can be *interval-scaled* or *ratio-scaled*.

Interval-Scaled Attributes

Interval-scaled attributes are measured on a scale of equal-size units. The values of interval-scaled attributes have order and can be positive, 0, or negative. Thus, in addition to providing a ranking of values, such attributes allow us to compare and quantify the *difference* between values.

Example 2.4 **Interval-scaled attributes.** A *temperature* attribute is interval-scaled. Suppose that we have the outdoor *temperature* value for a number of different days, where each day is an object. By ordering the values, we obtain a ranking of the objects with respect to *temperature*. In addition, we can quantify the difference between values. For example, a temperature of 20°C is five degrees higher than a temperature of 15°C. Calendar dates are another example. For instance, the years 2002 and 2010 are eight years apart. ■

Temperatures in Celsius and Fahrenheit do not have a true zero-point, that is, neither 0°C nor 0°F indicates "no temperature." (On the Celsius scale, for example, the unit of measurement is 1/100 of the difference between the melting temperature and the boiling temperature of water in atmospheric pressure.) Although we can compute the *difference* between temperature values, we cannot talk of one temperature value as being a *multiple* of another. Without a true zero, we cannot say, for instance, that 10°C is twice as warm as 5°C. That is, we cannot speak of the values in terms of ratios. Similarly, there is no true zero-point for calendar dates. (The year 0 does not correspond to the beginning of time.) This brings us to ratio-scaled attributes, for which a true zero-point exits.

Because interval-scaled attributes are numeric, we can compute their mean value, in addition to the median and mode measures of central tendency.

Ratio-Scaled Attributes

A **ratio-scaled attribute** is a numeric attribute with an inherent zero-point. That is, if a measurement is ratio-scaled, we can speak of a value as being a multiple (or ratio) of another value. In addition, the values are ordered, and we can also compute the difference between values, as well as the mean, median, and mode.

Example 2.5 **Ratio-scaled attributes.** Unlike temperatures in Celsius and Fahrenheit, the Kelvin (K) temperature scale has what is considered a true zero-point ($0°K = -273.15°C$): It is the point at which the particles that comprise matter have zero kinetic energy. Other examples of ratio-scaled attributes include *count* attributes such as *years_of_experience* (e.g., the objects are employees) and *number_of_words* (e.g., the objects are documents). Additional examples include attributes to measure weight, height, latitude and longitude

coordinates (e.g., when clustering houses), and monetary quantities (e.g., you are 100 times richer with $100 than with $1). ■

2.1.6 Discrete versus Continuous Attributes

In our presentation, we have organized attributes into nominal, binary, ordinal, and numeric types. There are many ways to organize attribute types. The types are not mutually exclusive.

Classification algorithms developed from the field of machine learning often talk of attributes as being either *discrete* or *continuous*. Each type may be processed differently. A **discrete attribute** has a finite or countably infinite set of values, which may or may not be represented as integers. The attributes *hair_color*, *smoker*, *medical_test*, and *drink_size* each have a finite number of values, and so are discrete. Note that discrete attributes may have numeric values, such as 0 and 1 for binary attributes or, the values 0 to 110 for the attribute *age*. An attribute is *countably infinite* if the set of possible values is infinite but the values can be put in a one-to-one correspondence with natural numbers. For example, the attribute *customer_ID* is countably infinite. The number of customers can grow to infinity, but in reality, the actual set of values is countable (where the values can be put in one-to-one correspondence with the set of integers). Zip codes are another example.

If an attribute is not discrete, it is **continuous**. The terms *numeric attribute* and *continuous attribute* are often used interchangeably in the literature. (This can be confusing because, in the classic sense, continuous values are real numbers, whereas numeric values can be either integers or real numbers.) In practice, real values are represented using a finite number of digits. Continuous attributes are typically represented as floating-point variables.

2.2 Basic Statistical Descriptions of Data

For data preprocessing to be successful, it is essential to have an overall picture of your data. Basic statistical descriptions can be used to identify properties of the data and highlight which data values should be treated as noise or outliers.

This section discusses three areas of basic statistical descriptions. We start with *measures of central tendency* (Section 2.2.1), which measure the location of the middle or center of a data distribution. Intuitively speaking, given an attribute, where do most of its values fall? In particular, we discuss the mean, median, mode, and midrange.

In addition to assessing the central tendency of our data set, we also would like to have an idea of the *dispersion of the data*. That is, how are the data spread out? The most common data dispersion measures are the *range*, *quartiles*, and *interquartile range*; the *five-number summary* and *boxplots*; and the *variance* and *standard deviation* of the data These measures are useful for identifying outliers and are described in Section 2.2.2.

Finally, we can use many graphic displays of basic statistical descriptions to visually inspect our data (Section 2.2.3). Most statistical or graphical data presentation software

packages include bar charts, pie charts, and line graphs. Other popular displays of data summaries and distributions include *quantile plots*, *quantile–quantile plots*, *histograms*, and *scatter plots*.

2.2.1 Measuring the Central Tendency: Mean, Median, and Mode

In this section, we look at various ways to measure the central tendency of data. Suppose that we have some attribute X, like *salary*, which has been recorded for a set of objects. Let $x_1, x_2, \ldots, x_N$ be the set of N observed values or *observations* for X. Here, these values may also be referred to as the data set (for X). If we were to plot the observations for *salary*, where would most of the values fall? This gives us an idea of the central tendency of the data. Measures of central tendency include the mean, median, mode, and midrange.

The most common and effective numeric measure of the "center" of a set of data is the *(arithmetic) mean*. Let $x_1, x_2, \ldots, x_N$ be a set of N values or *observations*, such as for some numeric attribute X, like *salary*. The **mean** of this set of values is

$$\bar{x} = \frac{\sum_{i=1}^{N} x_i}{N} = \frac{x_1 + x_2 + \cdots + x_N}{N}. \tag{2.1}$$

This corresponds to the built-in aggregate function, *average* (**avg()** in SQL), provided in relational database systems.

Example 2.6 **Mean.** Suppose we have the following values for *salary* (in thousands of dollars), shown in increasing order: 30, 36, 47, 50, 52, 52, 56, 60, 63, 70, 70, 110. Using Eq. (2.1), we have

$$\bar{x} = \frac{30 + 36 + 47 + 50 + 52 + 52 + 56 + 60 + 63 + 70 + 70 + 110}{12}$$
$$= \frac{696}{12} = 58.$$

Thus, the mean salary is $58,000. ■

Sometimes, each value x_i in a set may be associated with a weight w_i for $i = 1, \ldots, N$. The weights reflect the significance, importance, or occurrence frequency attached to their respective values. In this case, we can compute

$$\bar{x} = \frac{\sum_{i=1}^{N} w_i x_i}{\sum_{i=1}^{N} w_i} = \frac{w_1 x_1 + w_2 x_2 + \cdots + w_N x_N}{w_1 + w_2 + \cdots + w_N}. \tag{2.2}$$

This is called the **weighted arithmetic mean** or the **weighted average**.

Although the mean is the singlemost useful quantity for describing a data set, it is not always the best way of measuring the center of the data. A major problem with the mean is its sensitivity to extreme (e.g., outlier) values. Even a small number of extreme values can corrupt the mean. For example, the mean salary at a company may be substantially pushed up by that of a few highly paid managers. Similarly, the mean score of a class in an exam could be pulled down quite a bit by a few very low scores. To offset the effect caused by a small number of extreme values, we can instead use the **trimmed mean**, which is the mean obtained after chopping off values at the high and low extremes. For example, we can sort the values observed for *salary* and remove the top and bottom 2% before computing the mean. We should avoid trimming too large a portion (such as 20%) at both ends, as this can result in the loss of valuable information.

For skewed (asymmetric) data, a better measure of the center of data is the **median**, which is the middle value in a set of ordered data values. It is the value that separates the higher half of a data set from the lower half.

In probability and statistics, the median generally applies to numeric data; however, we may extend the concept to ordinal data. Suppose that a given data set of N values for an attribute X is sorted in increasing order. If N is odd, then the median is the *middle value* of the ordered set. If N is even, then the median is not unique; it is the two middlemost values and any value in between. If X is a numeric attribute in this case, by convention, the median is taken as the average of the two middlemost values.

Example 2.7 **Median.** Let's find the median of the data from Example 2.6. The data are already sorted in increasing order. There is an even number of observations (i.e., 12); therefore, the median is not unique. It can be any value within the two middlemost values of 52 and 56 (that is, within the sixth and seventh values in the list). By convention, we assign the average of the two middlemost values as the median; that is, $\frac{52+56}{2} = \frac{108}{2} = 54$. Thus, the median is \$54,000.

Suppose that we had only the first 11 values in the list. Given an odd number of values, the median is the middlemost value. This is the sixth value in this list, which has a value of \$52,000. ■

The median is expensive to compute when we have a large number of observations. For numeric attributes, however, we can easily *approximate* the value. Assume that data are grouped in intervals according to their x_i data values and that the frequency (i.e., number of data values) of each interval is known. For example, employees may be grouped according to their annual salary in intervals such as \$10–20,000, \$20–30,000, and so on. Let the interval that contains the median frequency be the *median interval*. We can approximate the median of the entire data set (e.g., the median salary) by interpolation using the formula

$$median = L_1 + \left(\frac{N/2 - \left(\sum freq\right)_l}{freq_{median}} \right) width, \tag{2.3}$$

where L_1 is the lower boundary of the median interval, N is the number of values in the entire data set, $\left(\sum freq\right)_l$ is the sum of the frequencies of all of the intervals that are

lower than the median interval, $freq_{median}$ is the frequency of the median interval, and *width* is the width of the median interval.

The *mode* is another measure of central tendency. The **mode** for a set of data is the value that occurs most frequently in the set. Therefore, it can be determined for qualitative and quantitative attributes. It is possible for the greatest frequency to correspond to several different values, which results in more than one mode. Data sets with one, two, or three modes are respectively called **unimodal**, **bimodal**, and **trimodal**. In general, a data set with two or more modes is **multimodal**. At the other extreme, if each data value occurs only once, then there is no mode.

Example 2.8 **Mode.** The data from Example 2.6 are bimodal. The two modes are \$52,000 and \$70,000. ■

For unimodal numeric data that are moderately skewed (asymmetrical), we have the following empirical relation:

$$mean - mode \approx 3 \times (mean - median). \tag{2.4}$$

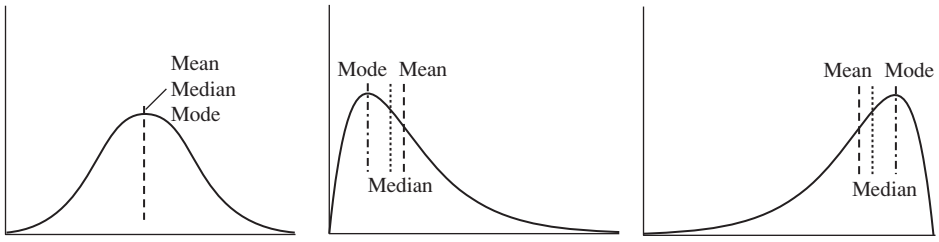
This implies that the mode for unimodal frequency curves that are moderately skewed can easily be approximated if the mean and median values are known.

The **midrange** can also be used to assess the central tendency of a numeric data set. It is the average of the largest and smallest values in the set. This measure is easy to compute using the SQL aggregate functions, max() and min().

Example 2.9 **Midrange.** The midrange of the data of Example 2.6 is $\frac{30{,}000+110{,}000}{2} = \$70{,}000$. ■

In a unimodal frequency curve with perfect **symmetric** data distribution, the mean, median, and mode are all at the same center value, as shown in Figure 2.1(a).

Data in most real applications are not symmetric. They may instead be either **positively skewed**, where the mode occurs at a value that is smaller than the median (Figure 2.1b), or **negatively skewed**, where the mode occurs at a value greater than the median (Figure 2.1c).



(a) Symmetric data **(b)** Positively skewed data **(c)** Negatively skewed data

Figure 2.1 Mean, median, and mode of symmetric versus positively and negatively skewed data.

2.2.2 Measuring the Dispersion of Data: Range, Quartiles, Variance, Standard Deviation, and Interquartile Range

We now look at measures to assess the dispersion or spread of numeric data. The measures include range, quantiles, quartiles, percentiles, and the interquartile range. The five-number summary, which can be displayed as a boxplot, is useful in identifying outliers. Variance and standard deviation also indicate the spread of a data distribution.

Range, Quartiles, and Interquartile Range

To start off, let's study the *range*, *quantiles*, *quartiles*, *percentiles*, and the *interquartile range* as measures of data dispersion.

Let $x_1, x_2, \ldots, x_N$ be a set of observations for some numeric attribute, X. The **range** of the set is the difference between the largest (max()) and smallest (min()) values.

Suppose that the data for attribute X are sorted in increasing numeric order. Imagine that we can pick certain data points so as to split the data distribution into equal-size consecutive sets, as in Figure 2.2. These data points are called *quantiles*. **Quantiles** are points taken at regular intervals of a data distribution, dividing it into essentially equal-size consecutive sets. (We say "essentially" because there may not be data values of X that divide the data into exactly equal-sized subsets. For readability, we will refer to them as equal.) The kth *q-quantile* for a given data distribution is the value x such that at most k/q of the data values are less than x and at most $(q-k)/q$ of the data values are more than x, where k is an integer such that $0 < k < q$. There are $q-1$ q-quantiles.

The 2-quantile is the data point dividing the lower and upper halves of the data distribution. It corresponds to the median. The 4-quantiles are the three data points that split the data distribution into four equal parts; each part represents one-fourth of the data distribution. They are more commonly referred to as **quartiles**. The 100-quantiles are more commonly referred to as **percentiles**; they divide the data distribution into 100 equal-sized consecutive sets. The median, quartiles, and percentiles are the most widely used forms of quantiles.

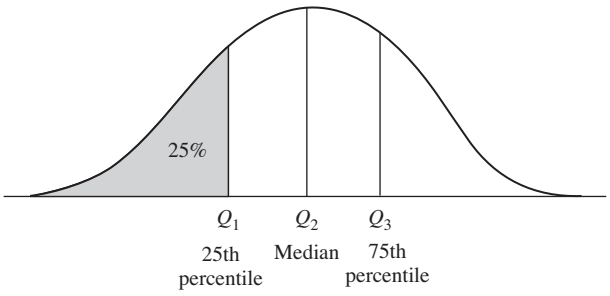


Figure 2.2 A plot of the data distribution for some attribute X. The quantiles plotted are quartiles. The three quartiles divide the distribution into four equal-size consecutive subsets. The second quartile corresponds to the median.

The quartiles give an indication of a distribution's center, spread, and shape. The **first quartile**, denoted by Q_1, is the 25th percentile. It cuts off the lowest 25% of the data. The **third quartile**, denoted by Q_3, is the 75th percentile—it cuts off the lowest 75% (or highest 25%) of the data. The second quartile is the 50th percentile. As the median, it gives the center of the data distribution.

The distance between the first and third quartiles is a simple measure of spread that gives the range covered by the middle half of the data. This distance is called the **interquartile range** (**IQR**) and is defined as

$$IQR = Q_3 - Q_1. \tag{2.5}$$

Example 2.10 **Interquartile range.** The quartiles are the three values that split the sorted data set into four equal parts. The data of Example 2.6 contain 12 observations, already sorted in increasing order. Thus, the quartiles for this data are the third, sixth, and ninth values, respectively, in the sorted list. Therefore, $Q_1 = \$47{,}000$ and Q_3 is \$63,000. Thus, the interquartile range is $IQR = 63 - 47 = \$16{,}000$. (Note that the sixth value is a median, \$52,000, although this data set has two medians since the number of data values is even.) ■

Five-Number Summary, Boxplots, and Outliers

No single numeric measure of spread (e.g., *IQR*) is very useful for describing skewed distributions. Have a look at the symmetric and skewed data distributions of Figure 2.1. In the symmetric distribution, the median (and other measures of central tendency) splits the data into equal-size halves. This does not occur for skewed distributions. Therefore, it is more informative to also provide the two quartiles Q_1 and Q_3, along with the median. A common rule of thumb for identifying suspected **outliers** is to single out values falling at least $1.5 \times IQR$ above the third quartile or below the first quartile.

Because Q_1, the median, and Q_3 together contain no information about the endpoints (e.g., tails) of the data, a fuller summary of the shape of a distribution can be obtained by providing the lowest and highest data values as well. This is known as the *five-number summary*. The **five-number summary** of a distribution consists of the median (Q_2), the quartiles Q_1 and Q_3, and the smallest and largest individual observations, written in the order of *Minimum*, Q_1, *Median*, Q_3, *Maximum*.

Boxplots are a popular way of visualizing a distribution. A boxplot incorporates the five-number summary as follows:

- Typically, the ends of the box are at the quartiles so that the box length is the interquartile range.
- The median is marked by a line within the box.
- Two lines (called *whiskers*) outside the box extend to the smallest (*Minimum*) and largest (*Maximum*) observations.

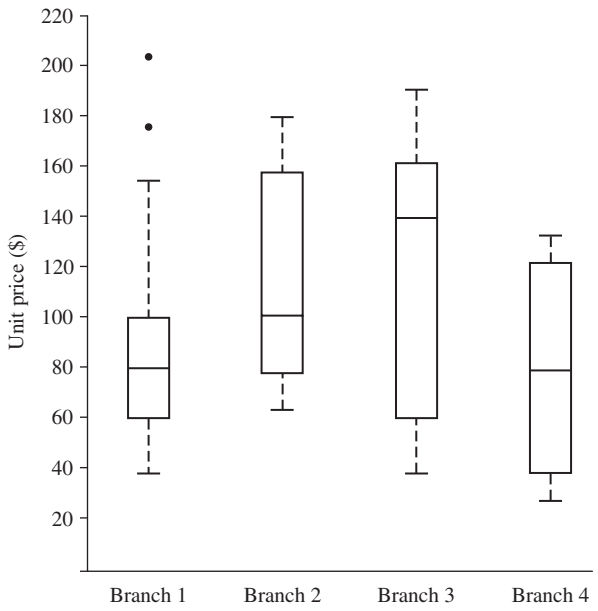


Figure 2.3 Boxplot for the unit price data for items sold at four branches of *AllElectronics* during a given time period.

When dealing with a moderate number of observations, it is worthwhile to plot potential outliers individually. To do this in a boxplot, the whiskers are extended to the extreme low and high observations *only if* these values are less than $1.5 \times IQR$ beyond the quartiles. Otherwise, the whiskers terminate at the most extreme observations occurring within $1.5 \times IQR$ of the quartiles. The remaining cases are plotted individually. Boxplots can be used in the comparisons of several sets of compatible data.

Example 2.11 **Boxplot.** Figure 2.3 shows boxplots for unit price data for items sold at four branches of *AllElectronics* during a given time period. For branch 1, we see that the median price of items sold is \$80, Q_1 is \$60, and Q_3 is \$100. Notice that two outlying observations for this branch were plotted individually, as their values of 175 and 202 are more than 1.5 times the IQR here of 40. ■

Boxplots can be computed in $O(n \log n)$ time. Approximate boxplots can be computed in linear or sublinear time depending on the quality guarantee required.

Variance and Standard Deviation

Variance and standard deviation are measures of data dispersion. They indicate how spread out a data distribution is. A low standard deviation means that the data observations tend to be very close to the mean, while a high standard deviation indicates that the data are spread out over a large range of values.

The **variance** of N observations, $x_1, x_2, \ldots, x_N$, for a numeric attribute X is

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 = \left(\frac{1}{N}\sum_{i=1}^{N}x_i^2\right) - \bar{x}^2, \tag{2.6}$$

where $\bar{x}$ is the mean value of the observations, as defined in Eq. (2.1). The **standard deviation**, σ, of the observations is the square root of the variance, σ^2.

Example 2.12 **Variance and standard deviation.** In Example 2.6, we found $\bar{x} = \$58{,}000$ using Eq. (2.1) for the mean. To determine the variance and standard deviation of the data from that example, we set $N = 12$ and use Eq. (2.6) to obtain

$$\begin{aligned}\sigma^2 &= \frac{1}{12}(30^2 + 36^2 + 47^2 \ldots + 110^2) - 58^2 \\ &\approx 379.17 \\ \sigma &\approx \sqrt{379.17} \approx 19.47.\end{aligned}$$

■

The basic properties of the standard deviation, σ, as a measure of spread are as follows:

- σ measures spread about the mean and should be considered only when the mean is chosen as the measure of center.
- $\sigma = 0$ only when there is no spread, that is, when all observations have the same value. Otherwise, $\sigma > 0$.

Importantly, an observation is unlikely to be more than several standard deviations away from the mean. Mathematically, using Chebyshev's inequality, it can be shown that at least $\left(1 - \frac{1}{k^2}\right) \times 100\%$ of the observations are no more than k standard deviations from the mean. Therefore, the standard deviation is a good indicator of the spread of a data set.

The computation of the variance and standard deviation is scalable in large databases.

2.2.3 Graphic Displays of Basic Statistical Descriptions of Data

In this section, we study graphic displays of basic statistical descriptions. These include *quantile plots*, *quantile–quantile plots*, *histograms*, and *scatter plots*. Such graphs are helpful for the visual inspection of data, which is useful for data preprocessing. The first three of these show univariate distributions (i.e., data for one attribute), while scatter plots show bivariate distributions (i.e., involving two attributes).

Quantile Plot

In this and the following subsections, we cover common graphic displays of data distributions. A **quantile plot** is a simple and effective way to have a first look at a univariate data distribution. First, it displays all of the data for the given attribute (allowing the user

to assess both the overall behavior and unusual occurrences). Second, it plots quantile information (see Section 2.2.2). Let x_i, for $i = 1$ to N, be the data sorted in increasing order so that x_1 is the smallest observation and x_N is the largest for some ordinal or numeric attribute X. Each observation, x_i, is paired with a percentage, f_i, which indicates that approximately $f_i \times 100\%$ of the data are below the value, x_i. We say "approximately" because there may not be a value with exactly a fraction, f_i, of the data below x_i. Note that the 0.25 percentile corresponds to quartile Q_1, the 0.50 percentile is the median, and the 0.75 percentile is Q_3.

Let

$$f_i = \frac{i - 0.5}{N}. \tag{2.7}$$

These numbers increase in equal steps of $1/N$, ranging from $\frac{1}{2N}$ (which is slightly above 0) to $1 - \frac{1}{2N}$ (which is slightly below 1). On a quantile plot, x_i is graphed against f_i. This allows us to compare different distributions based on their quantiles. For example, given the quantile plots of sales data for two different time periods, we can compare their Q_1, median, Q_3, and other f_i values at a glance.

Example 2.13 **Quantile plot.** Figure 2.4 shows a quantile plot for the *unit price* data of Table 2.1. ■

Quantile–Quantile Plot

A **quantile–quantile plot**, or **q-q plot**, graphs the quantiles of one univariate distribution against the corresponding quantiles of another. It is a powerful visualization tool in that it allows the user to view whether there is a shift in going from one distribution to another.

Suppose that we have two sets of observations for the attribute or variable *unit price*, taken from two different branch locations. Let $x_1, \ldots, x_N$ be the data from the first branch, and $y_1, \ldots, y_M$ be the data from the second, where each data set is sorted in increasing order. If $M = N$ (i.e., the number of points in each set is the same), then we simply plot y_i against x_i, where y_i and x_i are both $(i - 0.5)/N$ quantiles of their respective data sets. If $M < N$ (i.e., the second branch has fewer observations than the first), there can be only M points on the q-q plot. Here, y_i is the $(i - 0.5)/M$ quantile of the y

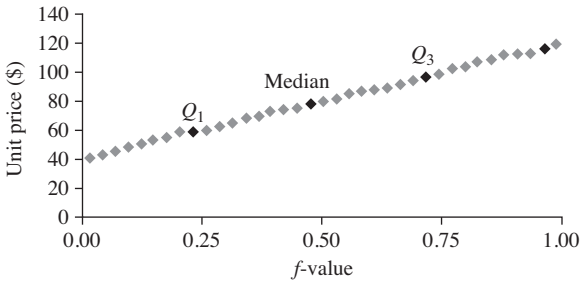


Figure 2.4 A quantile plot for the unit price data of Table 2.1.

Table 2.1 A Set of Unit Price Data for Items Sold at a Branch of *AllElectronics*

Unit price ($)	*Count of items sold*
40	275
43	300
47	250
—	—
74	360
75	515
78	540
—	—
115	320
117	270
120	350

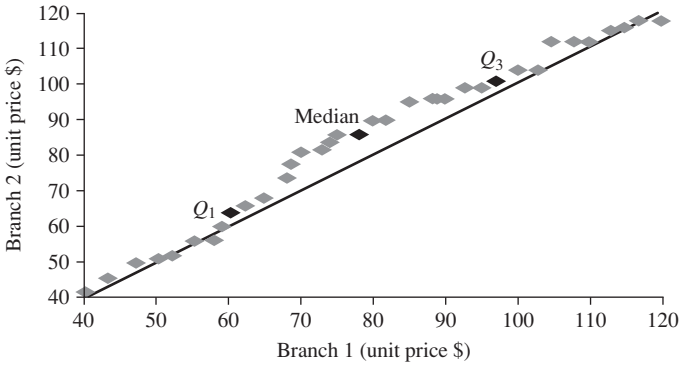


Figure 2.5 A q-q plot for unit price data from two *AllElectronics* branches.

data, which is plotted against the $(i - 0.5)/M$ quantile of the x data. This computation typically involves interpolation.

Example 2.14 **Quantile–quantile plot.** Figure 2.5 shows a quantile–quantile plot for *unit price* data of items sold at two branches of *AllElectronics* during a given time period. Each point corresponds to the same quantile for each data set and shows the unit price of items sold at branch 1 versus branch 2 for that quantile. (To aid in comparison, the straight line represents the case where, for each given quantile, the unit price at each branch is the same. The darker points correspond to the data for Q_1, the median, and Q_3, respectively.)

We see, for example, that at Q_1, the unit price of items sold at branch 1 was slightly less than that at branch 2. In other words, 25% of items sold at branch 1 were less than or

equal to \$60, while 25% of items sold at branch 2 were less than or equal to \$64. At the 50th percentile (marked by the median, which is also Q_2), we see that 50% of items sold at branch 1 were less than \$78, while 50% of items at branch 2 were less than \$85. In general, we note that there is a shift in the distribution of branch 1 with respect to branch 2 in that the unit prices of items sold at branch 1 tend to be lower than those at branch 2. ■

Histograms

Histograms (or **frequency histograms**) are at least a century old and are widely used. "Histos" means pole or mast, and "gram" means chart, so a histogram is a chart of poles. Plotting histograms is a graphical method for summarizing the distribution of a given attribute, X. If X is nominal, such as *automobile_model* or *item_type*, then a pole or vertical bar is drawn for each known value of X. The height of the bar indicates the frequency (i.e., count) of that X value. The resulting graph is more commonly known as a **bar chart**.

If X is numeric, the term *histogram* is preferred. The range of values for X is partitioned into disjoint consecutive subranges. The subranges, referred to as *buckets* or *bins*, are disjoint subsets of the data distribution for X. The range of a bucket is known as the **width**. Typically, the buckets are of equal width. For example, a *price* attribute with a value range of \$1 to \$200 (rounded up to the nearest dollar) can be partitioned into subranges 1 to 20, 21 to 40, 41 to 60, and so on. For each subrange, a bar is drawn with a height that represents the total count of items observed within the subrange. Histograms and partitioning rules are further discussed in Chapter 3 on data reduction.

Example 2.15 **Histogram.** Figure 2.6 shows a histogram for the data set of Table 2.1, where buckets (or bins) are defined by equal-width ranges representing \$20 increments and the frequency is the count of items sold. ■

Although histograms are widely used, they may not be as effective as the quantile plot, q-q plot, and boxplot methods in comparing groups of univariate observations.

Scatter Plots and Data Correlation

A **scatter plot** is one of the most effective graphical methods for determining if there appears to be a relationship, pattern, or trend between two numeric attributes. To construct a scatter plot, each pair of values is treated as a pair of coordinates in an algebraic sense and plotted as points in the plane. Figure 2.7 shows a scatter plot for the set of data in Table 2.1.

The scatter plot is a useful method for providing a first look at bivariate data to see clusters of points and outliers, or to explore the possibility of correlation relationships. Two attributes, X, and Y, are **correlated** if one attribute implies the other. Correlations can be positive, negative, or null (uncorrelated). Figure 2.8 shows examples of positive and negative correlations between two attributes. If the plotted points pattern slopes

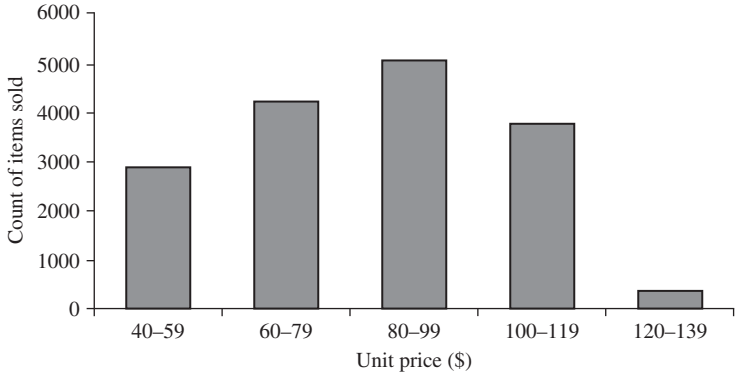


Figure 2.6 A histogram for the Table 2.1 data set.

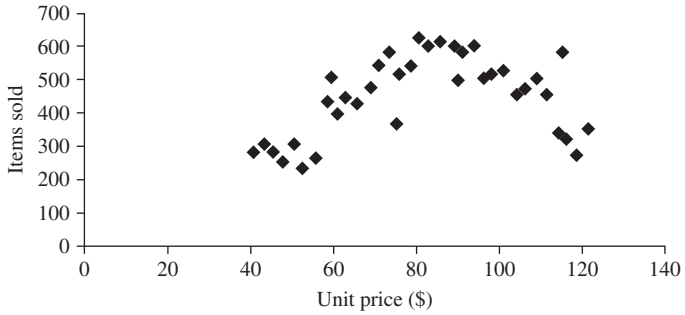


Figure 2.7 A scatter plot for the Table 2.1 data set.

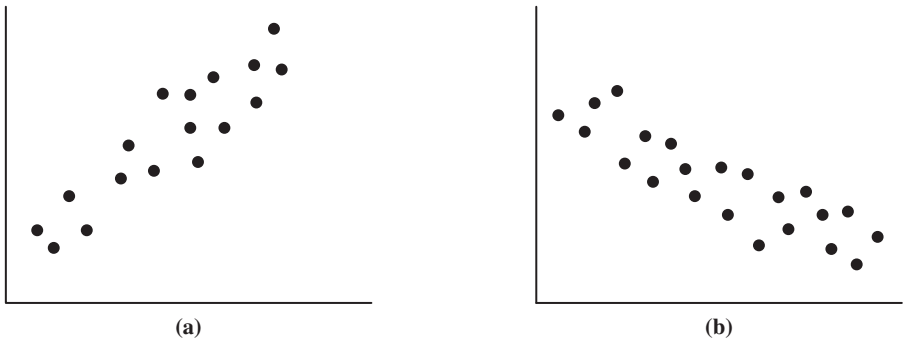


Figure 2.8 Scatter plots can be used to find (a) positive or (b) negative correlations between attributes.

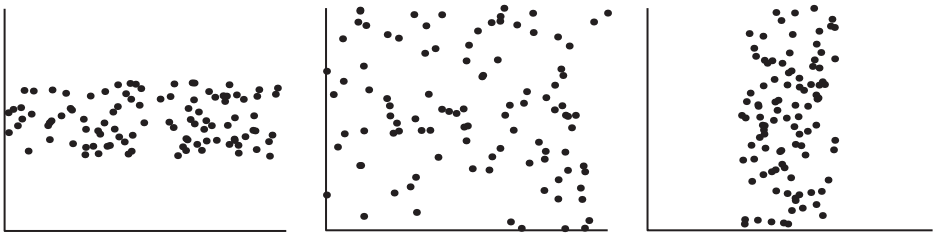


Figure 2.9 Three cases where there is no observed correlation between the two plotted attributes in each of the data sets.

from lower left to upper right, this means that the values of X increase as the values of Y increase, suggesting a *positive correlation* (Figure 2.8a). If the pattern of plotted points slopes from upper left to lower right, the values of X increase as the values of Y decrease, suggesting a *negative correlation* (Figure 2.8b). A line of best fit can be drawn to study the correlation between the variables. Statistical tests for correlation are given in Chapter 3 on data integration (Eq. (3.3)). Figure 2.9 shows three cases for which there is no correlation relationship between the two attributes in each of the given data sets. Section 2.3.2 shows how scatter plots can be extended to n attributes, resulting in a *scatter-plot matrix*.

In conclusion, basic data descriptions (e.g., measures of central tendency and measures of dispersion) and graphic statistical displays (e.g., quantile plots, histograms, and scatter plots) provide valuable insight into the overall behavior of your data. By helping to identify noise and outliers, they are especially useful for data cleaning.

2.3 Data Visualization

How can we convey data to users effectively? **Data visualization** aims to communicate data clearly and effectively through graphical representation. Data visualization has been used extensively in many applications—for example, at work for reporting, managing business operations, and tracking progress of tasks. More popularly, we can take advantage of visualization techniques to discover data relationships that are otherwise not easily observable by looking at the raw data. Nowadays, people also use data visualization to create fun and interesting graphics.

In this section, we briefly introduce the basic concepts of data visualization. We start with multidimensional data such as those stored in relational databases. We discuss several representative approaches, including pixel-oriented techniques, geometric projection techniques, icon-based techniques, and hierarchical and graph-based techniques. We then discuss the visualization of complex data and relations.

2.3.1 Pixel-Oriented Visualization Techniques

A simple way to visualize the value of a dimension is to use a pixel where the color of the pixel reflects the dimension's value. For a data set of m dimensions, **pixel-oriented techniques** create m windows on the screen, one for each dimension. The m dimension values of a record are mapped to m pixels at the corresponding positions in the windows. The colors of the pixels reflect the corresponding values.

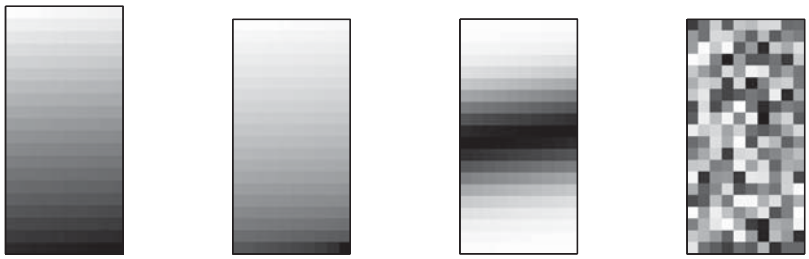
Inside a window, the data values are arranged in some global order shared by all windows. The global order may be obtained by sorting all data records in a way that's meaningful for the task at hand.

Example 2.16 **Pixel-oriented visualization.** *AllElectronics* maintains a customer information table, which consists of four dimensions: *income*, *credit_limit*, *transaction_volume*, and *age*. Can we analyze the correlation between *income* and the other attributes by visualization?

We can sort all customers in income-ascending order, and use this order to lay out the customer data in the four visualization windows, as shown in Figure 2.10. The pixel colors are chosen so that the smaller the value, the lighter the shading. Using pixel-based visualization, we can easily observe the following: *credit_limit* increases as *income* increases; customers whose income is in the middle range are more likely to purchase more from *AllElectronics*; there is no clear correlation between *income* and *age*. ■

In pixel-oriented techniques, data records can also be ordered in a query-dependent way. For example, given a point query, we can sort all records in descending order of similarity to the point query.

Filling a window by laying out the data records in a linear way may not work well for a wide window. The first pixel in a row is far away from the last pixel in the previous row, though they are next to each other in the global order. Moreover, a pixel is next to the one above it in the window, even though the two are not next to each other in the global order. To solve this problem, we can lay out the data records in a space-filling curve



(a) *income* **(b)** *credit_limit* **(c)** *transaction_volume* **(d)** *age*

Figure 2.10 Pixel-oriented visualization of four attributes by sorting all customers in *income* ascending order.

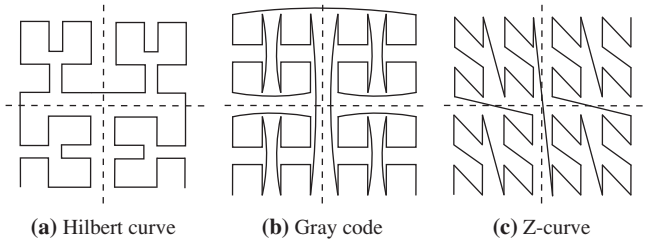


Figure 2.11 Some frequently used 2-D space-filling curves.

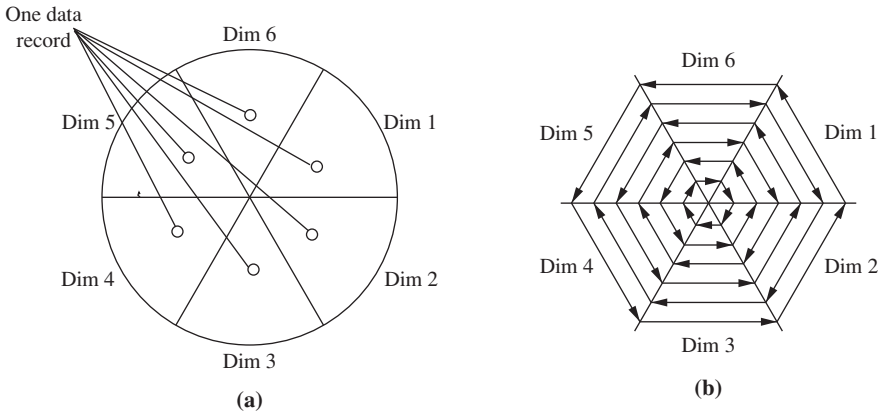


Figure 2.12 The circle segment technique. (a) Representing a data record in circle segments. (b) Laying out pixels in circle segments.

to fill the windows. A *space-filling curve* is a curve with a range that covers the entire n-dimensional unit hypercube. Since the visualization windows are 2-D, we can use any 2-D space-filling curve. Figure 2.11 shows some frequently used 2-D space-filling curves.

Note that the windows do not have to be rectangular. For example, the *circle segment technique* uses windows in the shape of segments of a circle, as illustrated in Figure 2.12. This technique can ease the comparison of dimensions because the dimension windows are located side by side and form a circle.

2.3.2 Geometric Projection Visualization Techniques

A drawback of pixel-oriented visualization techniques is that they cannot help us much in understanding the distribution of data in a multidimensional space. For example, they do not show whether there is a dense area in a multidimensional subspace. **Geometric**

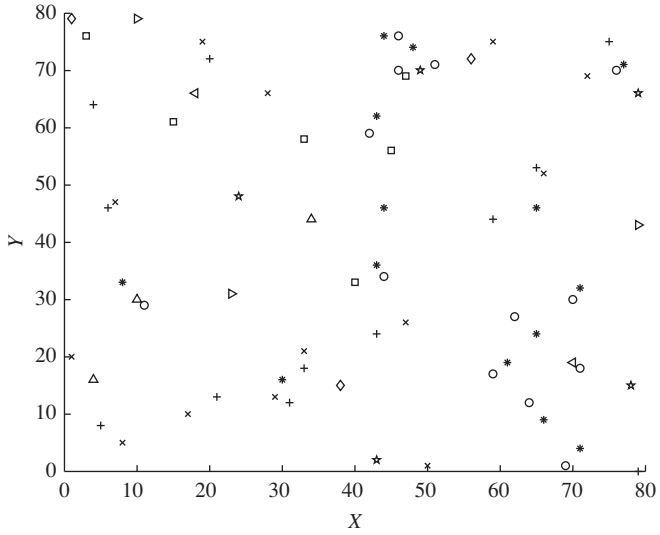


Figure 2.13 Visualization of a 2-D data set using a scatter plot. *Source: www.cs.sfu.ca/jpei/publications/rareevent-geoinformatica06.pdf*.

projection techniques help users find interesting projections of multidimensional data sets. The central challenge the geometric projection techniques try to address is how to visualize a high-dimensional space on a 2-D display.

A **scatter plot** displays 2-D data points using Cartesian coordinates. A third dimension can be added using different colors or shapes to represent different data points. Figure 2.13 shows an example, where X and Y are two spatial attributes and the third dimension is represented by different shapes. Through this visualization, we can see that points of types "+" and "×" tend to be colocated.

A 3-D scatter plot uses three axes in a Cartesian coordinate system. If it also uses color, it can display up to 4-D data points (Figure 2.14).

For data sets with more than four dimensions, scatter plots are usually ineffective. The **scatter-plot matrix** technique is a useful extension to the scatter plot. For an n-dimensional data set, a scatter-plot matrix is an $n \times n$ grid of 2-D scatter plots that provides a visualization of each dimension with every other dimension. Figure 2.15 shows an example, which visualizes the Iris data set. The data set consists of 450 samples from each of three species of Iris flowers. There are five dimensions in the data set: length and width of sepal and petal, and species.

The scatter-plot matrix becomes less effective as the dimensionality increases. Another popular technique, called parallel coordinates, can handle higher dimensionality. To visualize n-dimensional data points, the **parallel coordinates** technique draws n equally spaced axes, one for each dimension, parallel to one of the display axes.

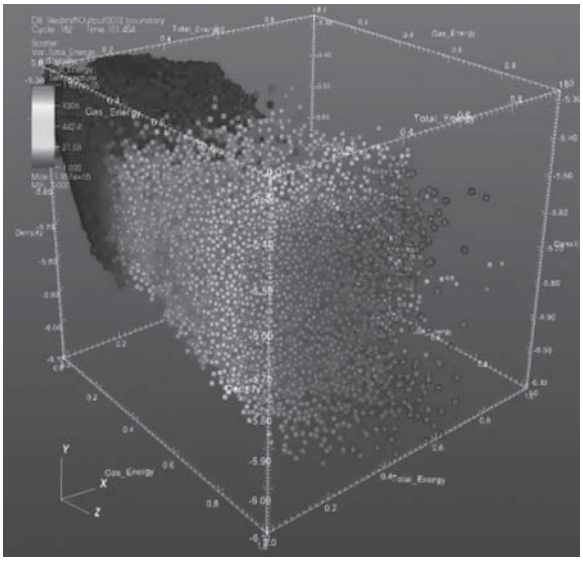


Figure 2.14 Visualization of a 3-D data set using a scatter plot. *Source: http://upload.wikimedia.org/wikipedia/commons/c/c4/Scatter_plot.jpg.*

A data record is represented by a polygonal line that intersects each axis at the point corresponding to the associated dimension value (Figure 2.16).

A major limitation of the parallel coordinates technique is that it cannot effectively show a data set of many records. Even for a data set of several thousand records, visual clutter and overlap often reduce the readability of the visualization and make the patterns hard to find.

2.3.3 Icon-Based Visualization Techniques

Icon-based visualization techniques use small icons to represent multidimensional data values. We look at two popular icon-based techniques: *Chernoff faces* and *stick figures*.

Chernoff faces were introduced in 1973 by statistician Herman Chernoff. They display multidimensional data of up to 18 variables (or dimensions) as a cartoon human face (Figure 2.17). Chernoff faces help reveal trends in the data. Components of the face, such as the eyes, ears, mouth, and nose, represent values of the dimensions by their shape, size, placement, and orientation. For example, dimensions can be mapped to the following facial characteristics: eye size, eye spacing, nose length, nose width, mouth curvature, mouth width, mouth openness, pupil size, eyebrow slant, eye eccentricity, and head eccentricity.

Chernoff faces make use of the ability of the human mind to recognize small differences in facial characteristics and to assimilate many facial characteristics at once.

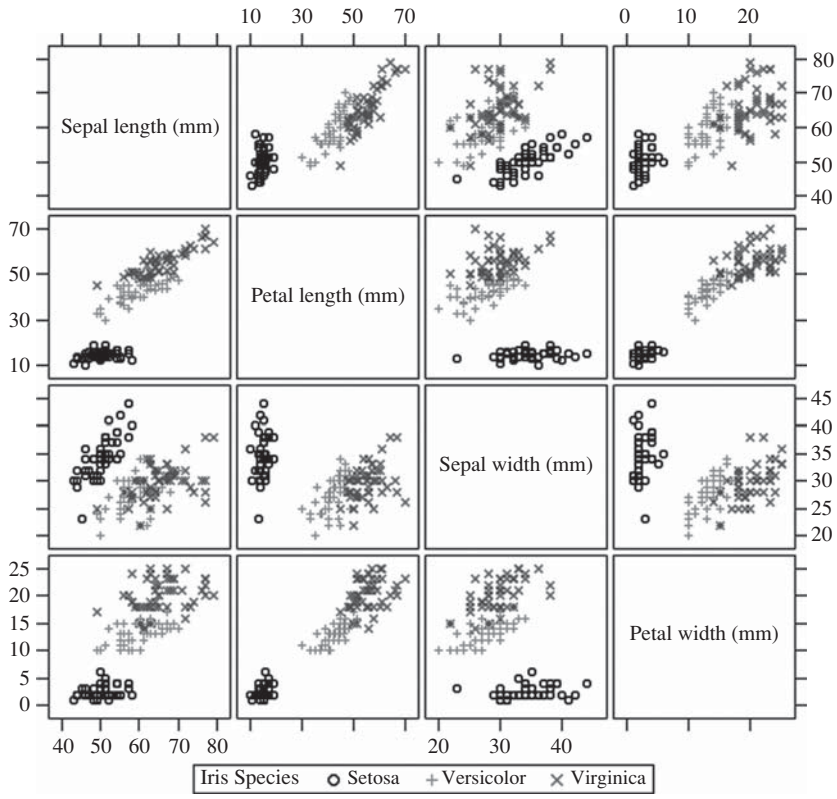


Figure 2.15 Visualization of the Iris data set using a scatter-plot matrix. *Source: http://support.sas.com/documentation/cdl/en/grstatproc/61948/HTML/default/images/gsgscmat.gif*.

Viewing large tables of data can be tedious. By condensing the data, Chernoff faces make the data easier for users to digest. In this way, they facilitate visualization of regularities and irregularities present in the data, although their power in relating multiple relationships is limited. Another limitation is that specific data values are not shown. Furthermore, facial features vary in perceived importance. This means that the similarity of two faces (representing two multidimensional data points) can vary depending on the order in which dimensions are assigned to facial characteristics. Therefore, this mapping should be carefully chosen. Eye size and eyebrow slant have been found to be important.

Asymmetrical Chernoff faces were proposed as an extension to the original technique. Since a face has vertical symmetry (along the y-axis), the left and right side of a face are identical, which wastes space. Asymmetrical Chernoff faces double the number of facial characteristics, thus allowing up to 36 dimensions to be displayed.

The **stick figure** visualization technique maps multidimensional data to five-piece stick figures, where each figure has four limbs and a body. Two dimensions are mapped to the display (x and y) axes and the remaining dimensions are mapped to the angle

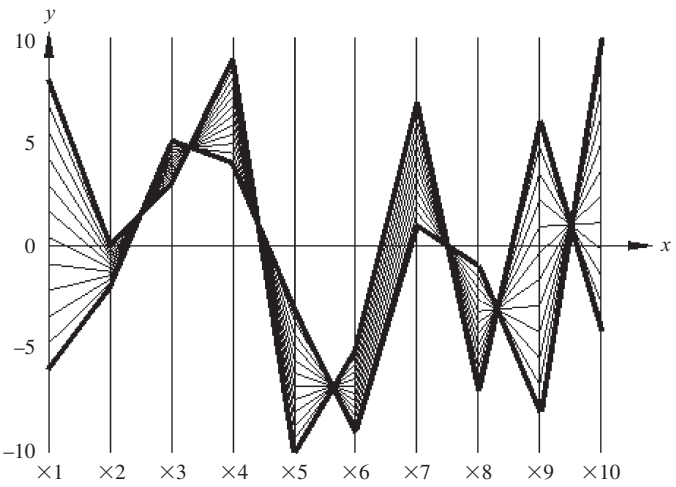


Figure 2.16 Here is a visualization that uses parallel coordinates. *Source: www.stat.columbia.edu/~cook/movabletype/archives/2007/10/parallel_coordi.thml.*

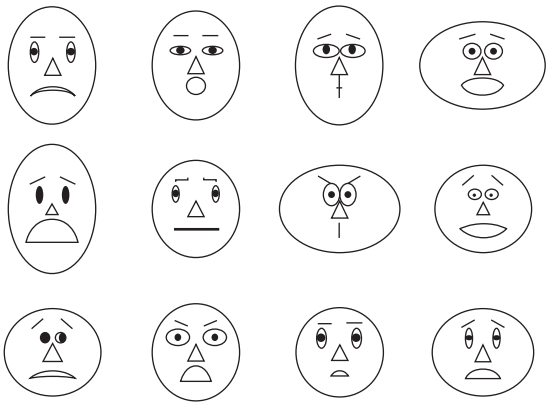


Figure 2.17 Chernoff faces. Each face represents an n-dimensional data point ($n \leq 18$).

and/or length of the limbs. Figure 2.18 shows census data, where *age* and *income* are mapped to the display axes, and the remaining dimensions (*gender, education*, and so on) are mapped to stick figures. If the data items are relatively dense with respect to the two display dimensions, the resulting visualization shows texture patterns, reflecting data trends.

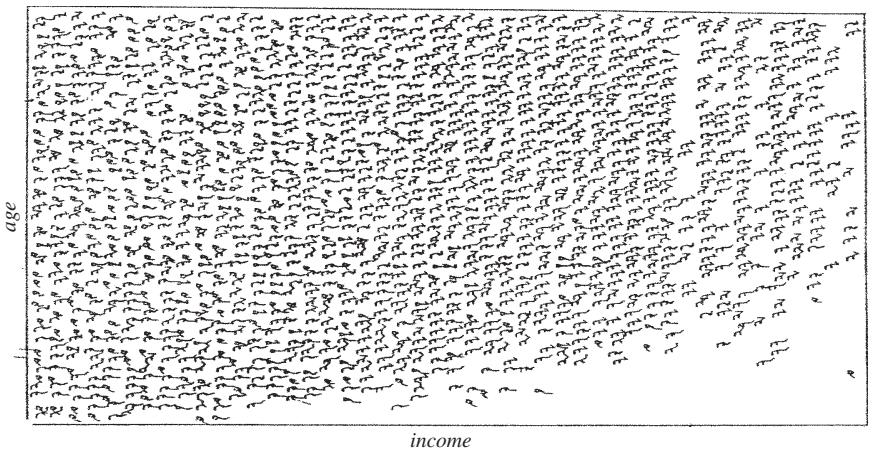


Figure 2.18 Census data represented using stick figures. *Source:* Professor G. Grinstein, Department of Computer Science, University of Massachusetts at Lowell.

2.3.4 Hierarchical Visualization Techniques

The visualization techniques discussed so far focus on visualizing multiple dimensions simultaneously. However, for a large data set of high dimensionality, it would be difficult to visualize all dimensions at the same time. **Hierarchical visualization techniques** partition all dimensions into subsets (i.e., subspaces). The subspaces are visualized in a hierarchical manner.

"Worlds-within-Worlds," also known as *n*-Vision, is a representative hierarchical visualization method. Suppose we want to visualize a 6-D data set, where the dimensions are $F, X_1, \ldots, X_5$. We want to observe how dimension F changes with respect to the other dimensions. We can first fix the values of dimensions X_3, X_4, X_5 to some selected values, say, c_3, c_4, c_5. We can then visualize F, X_1, X_2 using a 3-D plot, called a *world*, as shown in Figure 2.19. The position of the origin of the inner world is located at the point (c_3, c_4, c_5) in the outer world, which is another 3-D plot using dimensions X_3, X_4, X_5. A user can interactively change, in the outer world, the location of the origin of the inner world. The user then views the resulting changes of the inner world. Moreover, a user can vary the dimensions used in the inner world and the outer world. Given more dimensions, more levels of worlds can be used, which is why the method is called "worlds-within-worlds."

As another example of hierarchical visualization methods, **tree-maps** display hierarchical data as a set of nested rectangles. For example, Figure 2.20 shows a tree-map visualizing Google news stories. All news stories are organized into seven categories, each shown in a large rectangle of a unique color. Within each category (i.e., each rectangle at the top level), the news stories are further partitioned into smaller subcategories.

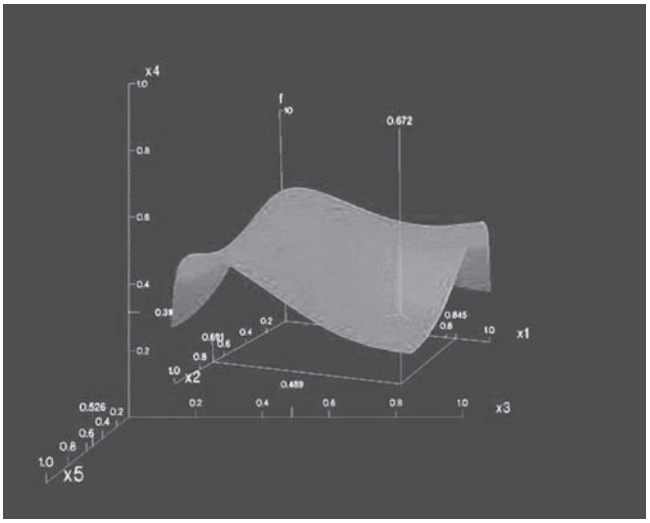


Figure 2.19 "Worlds-within-Worlds" (also known as *n*-Vision). *Source: http://graphics.cs.columbia.edu/projects/AutoVisual/images/1.dipstick.5.gif.*

2.3.5 Visualizing Complex Data and Relations

In early days, visualization techniques were mainly for numeric data. Recently, more and more non-numeric data, such as text and social networks, have become available. Visualizing and analyzing such data attracts a lot of interest.

There are many new visualization techniques dedicated to these kinds of data. For example, many people on the Web tag various objects such as pictures, blog entries, and product reviews. A **tag cloud** is a visualization of statistics of user-generated tags. Often, in a tag cloud, tags are listed alphabetically or in a user-preferred order. The importance of a tag is indicated by font size or color. Figure 2.21 shows a tag cloud for visualizing the popular tags used in a Web site.

Tag clouds are often used in two ways. First, in a tag cloud for a single item, we can use the size of a tag to represent the number of times that the tag is applied to this item by different users. Second, when visualizing the tag statistics on multiple items, we can use the size of a tag to represent the number of items that the tag has been applied to, that is, the popularity of the tag.

In addition to complex data, complex relations among data entries also raise challenges for visualization. For example, Figure 2.22 uses a disease influence graph to visualize the correlations between diseases. The nodes in the graph are diseases, and the size of each node is proportional to the prevalence of the corresponding disease. Two nodes are linked by an edge if the corresponding diseases have a strong correlation. The width of an edge is proportional to the strength of the correlation pattern of the two corresponding diseases.

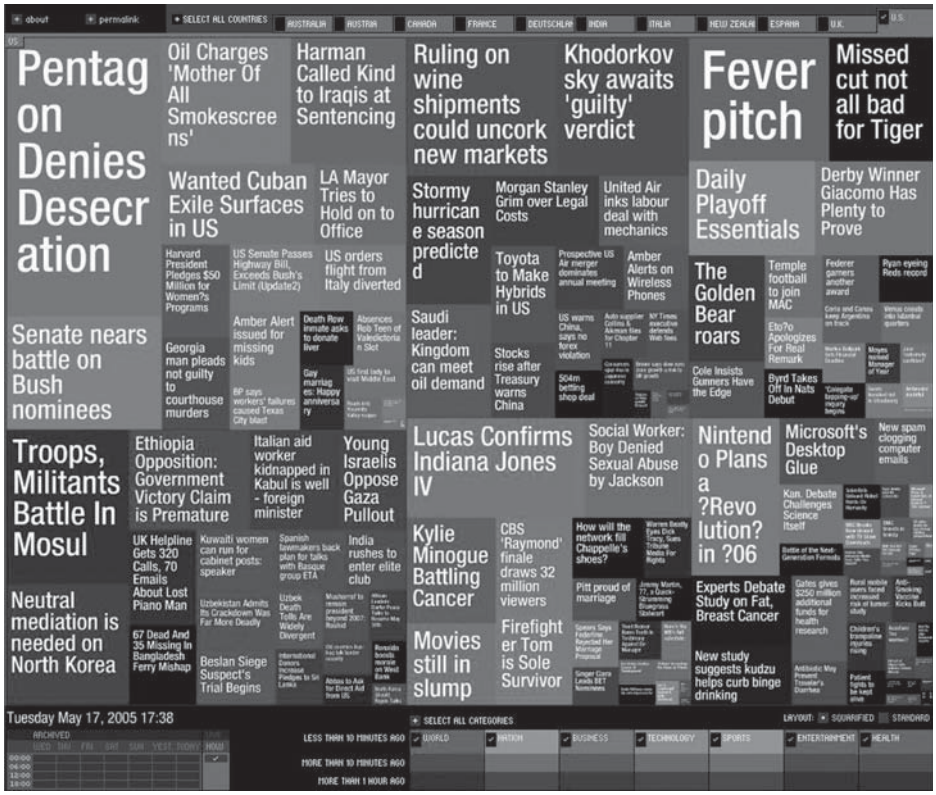


Figure 2.20 Newsmap: Use of tree-maps to visualize Google news headline stories. *Source: www.cs.umd.edu/class/spring2005/cmsc838s/viz4all/ss/newsmap.png.*

In summary, visualization provides effective tools to explore data. We have introduced several popular methods and the essential ideas behind them. There are many existing tools and methods. Moreover, visualization can be used in data mining in various aspects. In addition to visualizing data, visualization can be used to represent the data mining process, the patterns obtained from a mining method, and user interaction with the data. Visual data mining is an important research and development direction.

2.4 Measuring Data Similarity and Dissimilarity

In data mining applications, such as clustering, outlier analysis, and nearest-neighbor classification, we need ways to assess how alike or unalike objects are in comparison to one another. For example, a store may want to search for clusters of *customer* objects, resulting in groups of customers with similar characteristics (e.g., similar income, area of residence, and age). Such information can then be used for marketing. A **cluster** is

Figure 2.21 Using a tag cloud to visualize popular Web site tags. *Source:* A snapshot of *www.flickr.com/photos/tags/*, January 23, 2010.

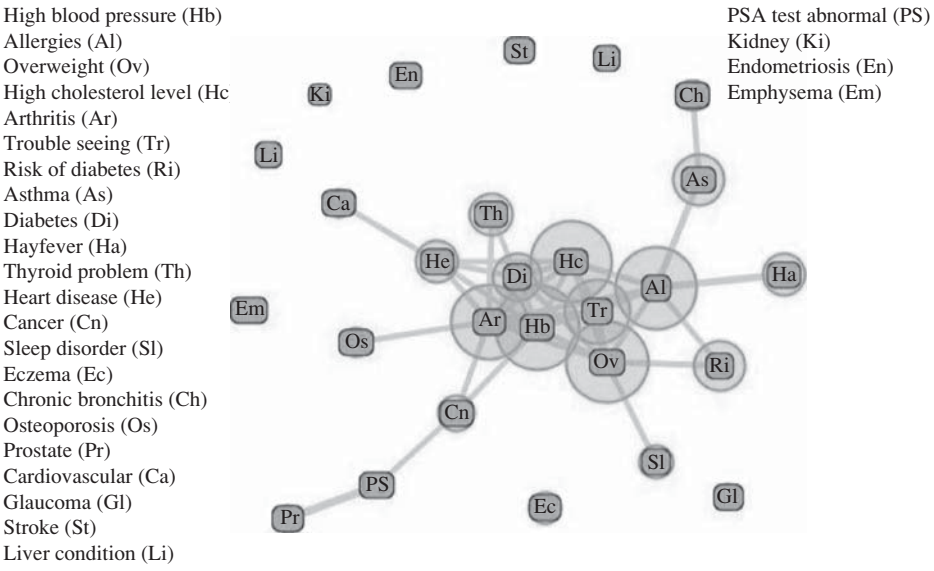


Figure 2.22 Disease influence graph of people at least 20 years old in the NHANES data set.

a collection of data objects such that the objects within a cluster are *similar* to one another and *dissimilar* to the objects in other clusters. Outlier analysis also employs clustering-based techniques to identify potential outliers as objects that are highly dissimilar to others. Knowledge of object similarities can also be used in nearest-neighbor classification schemes where a given object (e.g., a *patient*) is assigned a class label (relating to, say, a *diagnosis*) based on its similarity toward other objects in the model.

This section presents similarity and dissimilarity measures, which are referred to as measures of *proximity*. Similarity and dissimilarity are related. A similarity measure for two objects, i and j, will typically return the value 0 if the objects are unalike. The higher the similarity value, the greater the similarity between objects. (Typically, a value of 1 indicates complete similarity, that is, the objects are identical.) A dissimilarity measure works the opposite way. It returns a value of 0 if the objects are the same (and therefore, far from being dissimilar). The higher the dissimilarity value, the more dissimilar the two objects are.

In Section 2.4.1 we present two data structures that are commonly used in the above types of applications: the *data matrix* (used to store the data objects) and the *dissimilarity matrix* (used to store dissimilarity values for pairs of objects). We also switch to a different notation for data objects than previously used in this chapter since now we are dealing with objects described by more than one attribute. We then discuss how object dissimilarity can be computed for objects described by *nominal* attributes (Section 2.4.2), by *binary* attributes (Section 2.4.3), by *numeric* attributes (Section 2.4.4), by *ordinal* attributes (Section 2.4.5), or by combinations of these attribute types (Section 2.4.6). Section 2.4.7 provides similarity measures for very long and sparse data vectors, such as term-frequency vectors representing documents in information retrieval. Knowing how to compute dissimilarity is useful in studying attributes and will also be referenced in later topics on clustering (Chapters 10 and 11), outlier analysis (Chapter 12), and nearest-neighbor classification (Chapter 9).

2.4.1 Data Matrix versus Dissimilarity Matrix

In Section 2.2, we looked at ways of studying the central tendency, dispersion, and spread of observed values for some attribute X. Our objects there were one-dimensional, that is, described by a single attribute. In this section, we talk about objects described by *multiple* attributes. Therefore, we need a change in notation. Suppose that we have n objects (e.g., persons, items, or courses) described by p attributes (also called *measurements* or *features*, such as age, height, weight, or gender). The objects are $x_1 = (x_{11}, x_{12}, \ldots, x_{1p})$, $x_2 = (x_{21}, x_{22}, \ldots, x_{2p})$, and so on, where x_{ij} is the value for object x_i of the jth attribute. For brevity, we hereafter refer to object x_i as object i. The objects may be tuples in a relational database, and are also referred to as *data samples* or *feature vectors*.

Main memory-based clustering and nearest-neighbor algorithms typically operate on either of the following two data structures:

- **Data matrix** (or *object-by-attribute structure*): This structure stores the n data objects in the form of a relational table, or n-by-p matrix (n objects $\times p$ attributes):

$$\begin{bmatrix} x_{11} & \cdots & x_{1f} & \cdots & x_{1p} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{i1} & \cdots & x_{if} & \cdots & x_{ip} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{n1} & \cdots & x_{nf} & \cdots & x_{np} \end{bmatrix}. \tag{2.8}$$

Each row corresponds to an object. As part of our notation, we may use f to index through the p attributes.

- **Dissimilarity matrix** (or *object-by-object structure*): This structure stores a collection of proximities that are available for all pairs of n objects. It is often represented by an n-by-n table:

$$\begin{bmatrix} 0 & & & & \\ d(2,1) & 0 & & & \\ d(3,1) & d(3,2) & 0 & & \\ \vdots & \vdots & \vdots & & \\ d(n,1) & d(n,2) & \cdots & \cdots & 0 \end{bmatrix}, \tag{2.9}$$

where $d(i, j)$ is the measured **dissimilarity** or "difference" between objects i and j. In general, $d(i, j)$ is a non-negative number that is close to 0 when objects i and j are highly similar or "near" each other, and becomes larger the more they differ. Note that $d(i, i) = 0$; that is, the difference between an object and itself is 0. Furthermore, $d(i, j) = d(j, i)$. (For readability, we do not show the $d(j, i)$ entries; the matrix is symmetric.) Measures of dissimilarity are discussed throughout the remainder of this chapter.

Measures of similarity can often be expressed as a function of measures of dissimilarity. For example, for nominal data,

$$sim(i, j) = 1 - d(i, j), \tag{2.10}$$

where $sim(i, j)$ is the similarity between objects i and j. Throughout the rest of this chapter, we will also comment on measures of similarity.

A data matrix is made up of two entities or "things," namely rows (for objects) and columns (for attributes). Therefore, the data matrix is often called a **two-mode** matrix. The dissimilarity matrix contains one kind of entity (dissimilarities) and so is called a **one-mode** matrix. Many clustering and nearest-neighbor algorithms operate on a dissimilarity matrix. Data in the form of a data matrix can be transformed into a dissimilarity matrix before applying such algorithms.

2.4.2 Proximity Measures for Nominal Attributes

A nominal attribute can take on two or more states (Section 2.1.2). For example, *map_color* is a nominal attribute that may have, say, five states: *red, yellow, green, pink*, and *blue*.

Let the number of states of a nominal attribute be M. The states can be denoted by letters, symbols, or a set of integers, such as $1, 2, \ldots, M$. Notice that such integers are used just for data handling and do not represent any specific ordering.

"How is dissimilarity computed between objects described by nominal attributes?" The dissimilarity between two objects i and j can be computed based on the ratio of mismatches:

$$d(i, j) = \frac{p - m}{p}, \tag{2.11}$$

where m is the number of *matches* (i.e., the number of attributes for which i and j are in the same state), and p is the total number of attributes describing the objects. Weights can be assigned to increase the effect of m or to assign greater weight to the matches in attributes having a larger number of states.

Example 2.17 **Dissimilarity between nominal attributes.** Suppose that we have the sample data of Table 2.2, except that only the *object-identifier* and the attribute *test-1* are available, where *test-1* is nominal. (We will use *test-2* and *test-3* in later examples.) Let's compute the dissimilarity matrix (Eq. 2.9), that is,

$$\begin{bmatrix} 0 & & & \\ d(2,1) & 0 & & \\ d(3,1) & d(3,2) & 0 & \\ d(4,1) & d(4,2) & d(4,3) & 0 \end{bmatrix}.$$

Since here we have one nominal attribute, *test-1*, we set $p = 1$ in Eq. (2.11) so that $d(i, j)$ evaluates to 0 if objects i and j match, and 1 if the objects differ. Thus, we get

$$\begin{bmatrix} 0 & & & \\ 1 & 0 & & \\ 1 & 1 & 0 & \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

From this, we see that all objects are dissimilar except objects 1 and 4 (i.e., $d(4, 1) = 0$). ■

Table 2.2 A Sample Data Table Containing Attributes of Mixed Type

Object Identifier	*test-1 (nominal)*	*test-2 (ordinal)*	*test-3 (numeric)*
1	code A	excellent	45
2	code B	fair	22
3	code C	good	64
4	code A	excellent	28

Alternatively, similarity can be computed as

$$sim(i, j) = 1 - d(i, j) = \frac{m}{p}. \tag{2.12}$$

Proximity between objects described by nominal attributes can be computed using an alternative encoding scheme. Nominal attributes can be encoded using asymmetric binary attributes by creating a new binary attribute for each of the M states. For an object with a given state value, the binary attribute representing that state is set to 1, while the remaining binary attributes are set to 0. For example, to encode the nominal attribute *map_color*, a binary attribute can be created for each of the five colors previously listed. For an object having the color *yellow*, the *yellow* attribute is set to 1, while the remaining four attributes are set to 0. Proximity measures for this form of encoding can be calculated using the methods discussed in the next subsection.

2.4.3 Proximity Measures for Binary Attributes

Let's look at dissimilarity and similarity measures for objects described by either *symmetric* or *asymmetric binary attributes*.

Recall that a binary attribute has only one of two states: 0 and 1, where 0 means that the attribute is absent, and 1 means that it is present (Section 2.1.3). Given the attribute *smoker* describing a patient, for instance, 1 indicates that the patient smokes, while 0 indicates that the patient does not. Treating binary attributes as if they are numeric can be misleading. Therefore, methods specific to binary data are necessary for computing dissimilarity.

"*So, how can we compute the dissimilarity between two binary attributes?*" One approach involves computing a dissimilarity matrix from the given binary data. If all binary attributes are thought of as having the same weight, we have the 2×2 contingency table of Table 2.3, where q is the number of attributes that equal 1 for both objects i and j, r is the number of attributes that equal 1 for object i but equal 0 for object j, s is the number of attributes that equal 0 for object i but equal 1 for object j, and t is the number of attributes that equal 0 for both objects i and j. The total number of attributes is p, where $p = q + r + s + t$.

Recall that for symmetric binary attributes, each state is equally valuable. Dissimilarity that is based on symmetric binary attributes is called **symmetric binary dissimilarity**. If objects i and j are described by symmetric binary attributes, then the

Table 2.3 Contingency Table for Binary Attributes

		Object j		
		1	0	sum
	1	q	r	$q+r$
Object i	0	s	t	$s+t$
	sum	$q+s$	$r+t$	p

dissimilarity between i and j is

$$d(i, j) = \frac{r + s}{q + r + s + t}. \tag{2.13}$$

For asymmetric binary attributes, the two states are not equally important, such as the *positive* (1) and *negative* (0) outcomes of a disease test. Given two asymmetric binary attributes, the agreement of two 1s (a positive match) is then considered more significant than that of two 0s (a negative match). Therefore, such binary attributes are often considered "monary" (having one state). The dissimilarity based on these attributes is called **asymmetric binary dissimilarity**, where the number of negative matches, t, is considered unimportant and is thus ignored in the following computation:

$$d(i, j) = \frac{r + s}{q + r + s}. \tag{2.14}$$

Complementarily, we can measure the difference between two binary attributes based on the notion of similarity instead of dissimilarity. For example, the **asymmetric binary similarity** between the objects i and j can be computed as

$$sim(i, j) = \frac{q}{q + r + s} = 1 - d(i, j). \tag{2.15}$$

The coefficient $sim(i, j)$ of Eq. (2.15) is called the **Jaccard coefficient** and is popularly referenced in the literature.

When both symmetric and asymmetric binary attributes occur in the same data set, the mixed attributes approach described in Section 2.4.6 can be applied.

Example 2.18 **Dissimilarity between binary attributes.** Suppose that a patient record table (Table 2.4) contains the attributes *name, gender, fever, cough, test-1, test-2, test-3*, and *test-4*, where *name* is an object identifier, *gender* is a symmetric attribute, and the remaining attributes are asymmetric binary.

For asymmetric attribute values, let the values *Y* (*yes*) and *P* (*positive*) be set to 1, and the value *N* (*no* or *negative*) be set to 0. Suppose that the distance between objects

Table 2.4 Relational Table Where Patients Are Described by Binary Attributes

name	*gender*	*fever*	*cough*	*test-1*	*test-2*	*test-3*	*test-4*
Jack	M	Y	N	P	N	N	N
Jim	M	Y	Y	N	N	N	N
Mary	F	Y	N	P	N	P	N
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(patients) is computed based only on the asymmetric attributes. According to Eq. (2.14), the distance between each pair of the three patients—Jack, Mary, and Jim—is

$$d(Jack, Jim) = \frac{1+1}{1+1+1} = 0.67,$$

$$d(Jack, Mary) = \frac{0+1}{2+0+1} = 0.33,$$

$$d(Jim, Mary) = \frac{1+2}{1+1+2} = 0.75.$$

These measurements suggest that Jim and Mary are unlikely to have a similar disease because they have the highest dissimilarity value among the three pairs. Of the three patients, Jack and Mary are the most likely to have a similar disease. ■

2.4.4 Dissimilarity of Numeric Data: Minkowski Distance

In this section, we describe distance measures that are commonly used for computing the dissimilarity of objects described by numeric attributes. These measures include the *Euclidean, Manhattan*, and *Minkowski distances*.

In some cases, the data are normalized before applying distance calculations. This involves transforming the data to fall within a smaller or common range, such as $[-1, 1]$ or $[0.0, 1.0]$. Consider a *height* attribute, for example, which could be measured in either meters or inches. In general, expressing an attribute in smaller units will lead to a larger range for that attribute, and thus tend to give such attributes greater effect or "weight." Normalizing the data attempts to give all attributes an equal weight. It may or may not be useful in a particular application. Methods for normalizing data are discussed in detail in Chapter 3 on data preprocessing.

The most popular distance measure is **Euclidean distance** (i.e., straight line or "as the crow flies"). Let $i = (x_{i1}, x_{i2}, \ldots, x_{ip})$ and $j = (x_{j1}, x_{j2}, \ldots, x_{jp})$ be two objects described by p numeric attributes. The Euclidean distance between objects i and j is defined as

$$d(i, j) = \sqrt{(x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + \cdots + (x_{ip} - x_{jp})^2}. \tag{2.16}$$

Another well-known measure is the **Manhattan (or city block) distance**, named so because it is the distance in blocks between any two points in a city (such as 2 blocks down and 3 blocks over for a total of 5 blocks). It is defined as

$$d(i, j) = |x_{i1} - x_{j1}| + |x_{i2} - x_{j2}| + \cdots + |x_{ip} - x_{jp}|. \tag{2.17}$$

Both the Euclidean and the Manhattan distance satisfy the following mathematical properties:

Non-negativity: $d(i, j) \geq 0$: Distance is a non-negative number.

Identity of indiscernibles: $d(i, i) = 0$: The distance of an object to itself is 0.

Symmetry: $d(i, j) = d(j, i)$: Distance is a symmetric function.

Triangle inequality: $d(i, j) \leq d(i, k) + d(k, j)$: Going directly from object i to object j in space is no more than making a detour over any other object k.

A measure that satisfies these conditions is known as **metric**. Please note that the non-negativity property is implied by the other three properties.

Example 2.19 **Euclidean distance and Manhattan distance.** Let $\boldsymbol{x_1} = (1, 2)$ and $\boldsymbol{x_2} = (3, 5)$ represent two objects as shown in Figure 2.23. The Euclidean distance between the two is $\sqrt{2^2 + 3^2} = 3.61$. The Manhattan distance between the two is $2 + 3 = 5$. ■

Minkowski distance is a generalization of the Euclidean and Manhattan distances. It is defined as

$$d(i, j) = \sqrt[h]{|x_{i1} - x_{j1}|^h + |x_{i2} - x_{j2}|^h + \cdots + |x_{ip} - x_{jp}|^h}, \tag{2.18}$$

where h is a real number such that $h \geq 1$. (Such a distance is also called L_p **norm** in some literature, where the symbol p refers to our notation of h. We have kept p as the number of attributes to be consistent with the rest of this chapter.) It represents the Manhattan distance when $h = 1$ (i.e., L_1 norm) and Euclidean distance when $h = 2$ (i.e., L_2 norm).

The **supremum distance** (also referred to as $\boldsymbol{L_{max}}$, $\boldsymbol{L_\infty}$ **norm** and as the **Chebyshev distance**) is a generalization of the Minkowski distance for $h \to \infty$. To compute it, we find the attribute f that gives the maximum difference in values between the two objects. This difference is the supremum distance, defined more formally as:

$$d(i, j) = \lim_{h \to \infty} \left(\sum_{f=1}^{p} |x_{if} - x_{jf}|^h \right)^{\frac{1}{h}} = \max_{f}^{p} |x_{if} - x_{jf}|. \tag{2.19}$$

The L^∞ norm is also known as the *uniform norm*.

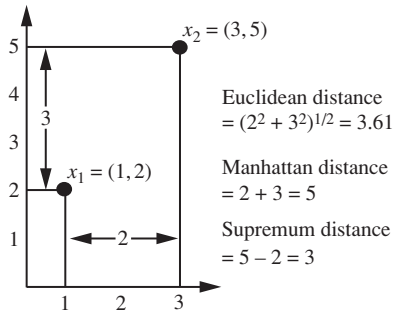


Figure 2.23 Euclidean, Manhattan, and supremum distances between two objects.

Example 2.20 **Supremum distance.** Let's use the same two objects, $\boldsymbol{x_1} = (1, 2)$ and $\boldsymbol{x_2} = (3, 5)$, as in Figure 2.23. The second attribute gives the greatest difference between values for the objects, which is $5 - 2 = 3$. This is the supremum distance between both objects. ■

If each attribute is assigned a weight according to its perceived importance, the **weighted Euclidean distance** can be computed as

$$d(i, j) = \sqrt{w_1|x_{i1} - x_{j1}|^2 + w_2|x_{i2} - x_{j2}|^2 + \cdots + w_m|x_{ip} - x_{jp}|^2}. \tag{2.20}$$

Weighting can also be applied to other distance measures as well.

2.4.5 Proximity Measures for Ordinal Attributes

The values of an ordinal attribute have a meaningful order or ranking about them, yet the magnitude between successive values is unknown (Section 2.1.4). An example includes the sequence *small*, *medium*, *large* for a *size* attribute. Ordinal attributes may also be obtained from the discretization of numeric attributes by splitting the value range into a finite number of categories. These categories are organized into ranks. That is, the range of a numeric attribute can be mapped to an ordinal attribute f having M_f states. For example, the range of the interval-scaled attribute *temperature* (in Celsius) can be organized into the following states: -30 to -10, -10 to 10, 10 to 30, representing the categories *cold temperature*, *moderate temperature*, and *warm temperature*, respectively. Let M represent the number of possible states that an ordinal attribute can have. These ordered states define the ranking $1, \ldots, M_f$.

"*How are ordinal attributes handled?*" The treatment of ordinal attributes is quite similar to that of numeric attributes when computing dissimilarity between objects. Suppose that f is an attribute from a set of ordinal attributes describing n objects. The dissimilarity computation with respect to f involves the following steps:

1. The value of f for the ith object is x_{if}, and f has M_f ordered states, representing the ranking $1, \ldots, M_f$. Replace each x_{if} by its corresponding rank, $r_{if} \in \{1, \ldots, M_f\}$.
2. Since each ordinal attribute can have a different number of states, it is often necessary to map the range of each attribute onto [0.0, 1.0] so that each attribute has equal weight. We perform such data normalization by replacing the rank r_{if} of the ith object in the fth attribute by

$$z_{if} = \frac{r_{if} - 1}{M_f - 1}. \tag{2.21}$$

3. Dissimilarity can then be computed using any of the distance measures described in Section 2.4.4 for numeric attributes, using z_{if} to represent the f value for the ith object.

Example 2.21 **Dissimilarity between ordinal attributes.** Suppose that we have the sample data shown earlier in Table 2.2, except that this time only the *object-identifier* and the continuous ordinal attribute, *test-2*, are available. There are three states for *test-2*: *fair*, *good*, and *excellent*, that is, $M_f = 3$. For step 1, if we replace each value for *test-2* by its rank, the four objects are assigned the ranks 3, 1, 2, and 3, respectively. Step 2 normalizes the ranking by mapping rank 1 to 0.0, rank 2 to 0.5, and rank 3 to 1.0. For step 3, we can use, say, the Euclidean distance (Eq. 2.16), which results in the following dissimilarity matrix:

$$\begin{bmatrix} 0 & & & \\ 1.0 & 0 & & \\ 0.5 & 0.5 & 0 & \\ 0 & 1.0 & 0.5 & 0 \end{bmatrix}.$$

Therefore, objects 1 and 2 are the most dissimilar, as are objects 2 and 4 (i.e., $d(2,1) = 1.0$ and $d(4,2) = 1.0$). This makes intuitive sense since objects 1 and 4 are both *excellent*. Object 2 is *fair*, which is at the opposite end of the range of values for *test-2*. ■

Similarity values for ordinal attributes can be interpreted from dissimilarity as $sim(i,j) = 1 - d(i,j)$.

2.4.6 Dissimilarity for Attributes of Mixed Types

Sections 2.4.2 through 2.4.5 discussed how to compute the dissimilarity between objects described by attributes of the same type, where these types may be either *nominal*, *symmetric binary*, *asymmetric binary*, *numeric*, or *ordinal*. However, in many real databases, objects are described by a *mixture* of attribute types. In general, a database can contain all of these attribute types.

"So, how can we compute the dissimilarity between objects of mixed attribute types?" One approach is to group each type of attribute together, performing separate data mining (e.g., clustering) analysis for each type. This is feasible if these analyses derive compatible results. However, in real applications, it is unlikely that a separate analysis per attribute type will generate compatible results.

A more preferable approach is to process all attribute types together, performing a single analysis. One such technique combines the different attributes into a single dissimilarity matrix, bringing all of the meaningful attributes onto a common scale of the interval [0.0, 1.0].

Suppose that the data set contains p attributes of mixed type. The dissimilarity $d(i, j)$ between objects i and j is defined as

$$d(i,j) = \frac{\sum_{f=1}^{p} \delta_{ij}^{(f)} d_{ij}^{(f)}}{\sum_{f=1}^{p} \delta_{ij}^{(f)}}, \tag{2.22}$$

where the indicator $\delta_{ij}^{(f)} = 0$ if either (1) x_{if} or x_{jf} is missing (i.e., there is no measurement of attribute f for object i or object j), or (2) $x_{if} = x_{jf} = 0$ and attribute f is asymmetric binary; otherwise, $\delta_{ij}^{(f)} = 1$. The contribution of attribute f to the dissimilarity between i and j (i.e., $d_{ij}^{(f)}$) is computed dependent on its type:

- If f is numeric: $d_{ij}^{(f)} = \frac{|x_{if} - x_{jf}|}{max_h x_{hf} - min_h x_{hf}}$, where h runs over all nonmissing objects for attribute f.
- If f is nominal or binary: $d_{ij}^{(f)} = 0$ if $x_{if} = x_{jf}$; otherwise, $d_{ij}^{(f)} = 1$.
- If f is ordinal: compute the ranks r_{if} and $z_{if} = \frac{r_{if} - 1}{M_f - 1}$, and treat z_{if} as numeric.

These steps are identical to what we have already seen for each of the individual attribute types. The only difference is for numeric attributes, where we normalize so that the values map to the interval [0.0, 1.0]. Thus, the dissimilarity between objects can be computed even when the attributes describing the objects are of different types.

Example 2.22 **Dissimilarity between attributes of mixed type.** Let's compute a dissimilarity matrix for the objects in Table 2.2. Now we will consider *all* of the attributes, which are of different types. In Examples 2.17 and 2.21, we worked out the dissimilarity matrices for each of the individual attributes. The procedures we followed for *test-1* (which is nominal) and *test-2* (which is ordinal) are the same as outlined earlier for processing attributes of mixed types. Therefore, we can use the dissimilarity matrices obtained for *test-1* and *test-2* later when we compute Eq. (2.22). First, however, we need to compute the dissimilarity matrix for the third attribute, *test-3* (which is numeric). That is, we must compute $d_{ij}^{(3)}$. Following the case for numeric attributes, we let $max_h x_h = 64$ and $min_h x_h = 22$. The difference between the two is used in Eq. (2.22) to normalize the values of the dissimilarity matrix. The resulting dissimilarity matrix for *test-3* is

$$\begin{bmatrix} 0 & & & \\ 0.55 & 0 & & \\ 0.45 & 1.00 & 0 & \\ 0.40 & 0.14 & 0.86 & 0 \end{bmatrix}.$$

We can now use the dissimilarity matrices for the three attributes in our computation of Eq. (2.22). The indicator $\delta_{ij}^{(f)} = 1$ for each of the three attributes, f. We get, for example, $d(3, 1) = \frac{1(1) + 1(0.50) + 1(0.45)}{3} = 0.65$. The resulting dissimilarity matrix obtained for the

data described by the three attributes of mixed types is:

$$\begin{bmatrix} 0 & & & \\ 0.85 & 0 & & \\ 0.65 & 0.83 & 0 & \\ 0.13 & 0.71 & 0.79 & 0 \end{bmatrix}.$$

From Table 2.2, we can intuitively guess that objects 1 and 4 are the most similar, based on their values for *test*-1 and *test*-2. This is confirmed by the dissimilarity matrix, where $d(4, 1)$ is the lowest value for any pair of different objects. Similarly, the matrix indicates that objects 1 and 2 are the least similar. ■

2.4.7 Cosine Similarity

A document can be represented by thousands of attributes, each recording the frequency of a particular word (such as a keyword) or phrase in the document. Thus, each document is an object represented by what is called a *term-frequency vector*. For example, in Table 2.5, we see that *Document1* contains five instances of the word *team*, while *hockey* occurs three times. The word *coach* is absent from the entire document, as indicated by a count value of 0. Such data can be highly asymmetric.

Term-frequency vectors are typically very long and **sparse** (i.e., they have many 0 values). Applications using such structures include information retrieval, text document clustering, biological taxonomy, and gene feature mapping. The traditional distance measures that we have studied in this chapter do not work well for such sparse numeric data. For example, two term-frequency vectors may have many 0 values in common, meaning that the corresponding documents do not share many words, but this does not make them similar. We need a measure that will focus on the words that the two documents *do* have in common, and the occurrence frequency of such words. In other words, we need a measure for numeric data that ignores zero-matches.

Cosine similarity is a measure of similarity that can be used to compare documents or, say, give a ranking of documents with respect to a given vector of query words. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two vectors for comparison. Using the cosine measure as a

Table 2.5 Document Vector or Term-Frequency Vector

Document	*team*	*coach*	*hockey*	*baseball*	*soccer*	*penalty*	*score*	*win*	*loss*	*season*
Document1	5	0	3	0	2	0	0	2	0	0
Document2	3	0	2	0	1	1	0	1	0	1
Document3	0	7	0	2	1	0	0	3	0	0
Document4	0	1	0	0	1	2	2	0	3	0

similarity function, we have

$$sim(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{||\boldsymbol{x}||||\boldsymbol{y}||}, \tag{2.23}$$

where $||\boldsymbol{x}||$ is the Euclidean norm of vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_p)$, defined as $\sqrt{x_1^2 + x_2^2 + \cdots + x_p^2}$. Conceptually, it is the length of the vector. Similarly, $||\boldsymbol{y}||$ is the Euclidean norm of vector $\boldsymbol{y}$. The measure computes the cosine of the angle between vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. A cosine value of 0 means that the two vectors are at 90 degrees to each other (orthogonal) and have no match. The closer the cosine value to 1, the smaller the angle and the greater the match between vectors. Note that because the cosine similarity measure does not obey all of the properties of Section 2.4.4 defining metric measures, it is referred to as a *nonmetric measure*.

Example 2.23 **Cosine similarity between two term-frequency vectors.** Suppose that $\boldsymbol{x}$ and $\boldsymbol{y}$ are the first two term-frequency vectors in Table 2.5. That is, $\boldsymbol{x} = (5, 0, 3, 0, 2, 0, 0, 2, 0, 0)$ and $\boldsymbol{y} = (3, 0, 2, 0, 1, 1, 0, 1, 0, 1)$. How similar are $\boldsymbol{x}$ and $\boldsymbol{y}$? Using Eq. (2.23) to compute the cosine similarity between the two vectors, we get:

$$\begin{aligned}
\boldsymbol{x}^t \cdot \boldsymbol{y} &= 5 \times 3 + 0 \times 0 + 3 \times 2 + 0 \times 0 + 2 \times 1 + 0 \times 1 + 0 \times 0 + 2 \times 1 \\
&\quad + 0 \times 0 + 0 \times 1 = 25 \\
||\boldsymbol{x}|| &= \sqrt{5^2 + 0^2 + 3^2 + 0^2 + 2^2 + 0^2 + 0^2 + 2^2 + 0^2 + 0^2} = 6.48 \\
||\boldsymbol{y}|| &= \sqrt{3^2 + 0^2 + 2^2 + 0^2 + 1^2 + 1^2 + 0^2 + 1^2 + 0^2 + 1^2} = 4.12 \\
sim(\boldsymbol{x}, \boldsymbol{y}) &= 0.94
\end{aligned}$$

Therefore, if we were using the cosine similarity measure to compare these documents, they would be considered quite similar. ■

When attributes are binary-valued, the cosine similarity function can be interpreted in terms of shared features or attributes. Suppose an object $\boldsymbol{x}$ possesses the ith attribute if $x_i = 1$. Then $\boldsymbol{x}^t \cdot \boldsymbol{y}$ is the number of attributes possessed (i.e., shared) by both $\boldsymbol{x}$ and $\boldsymbol{y}$, and $|\boldsymbol{x}||\boldsymbol{y}|$ is the *geometric mean* of the number of attributes possessed by $\boldsymbol{x}$ and the number possessed by $\boldsymbol{y}$. Thus, $sim(\boldsymbol{x}, \boldsymbol{y})$ is a measure of relative possession of common attributes.

A simple variation of cosine similarity for the preceding scenario is

$$sim(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{\boldsymbol{x} \cdot \boldsymbol{x} + \boldsymbol{y} \cdot \boldsymbol{y} - \boldsymbol{x} \cdot \boldsymbol{y}}, \tag{2.24}$$

which is the ratio of the number of attributes shared by $\boldsymbol{x}$ and $\boldsymbol{y}$ to the number of attributes possessed by $\boldsymbol{x}$ or $\boldsymbol{y}$. This function, known as the **Tanimoto coefficient** or **Tanimoto distance**, is frequently used in information retrieval and biology taxonomy.

2.5 Summary

- Data sets are made up of data objects. A **data object** represents an entity. Data objects are described by attributes. Attributes can be nominal, binary, ordinal, or numeric.
- The values of a **nominal** (or **categorical**) **attribute** are symbols or names of things, where each value represents some kind of category, code, or state.
- **Binary attributes** are nominal attributes with only two possible states (such as 1 and 0 or true and false). If the two states are equally important, the attribute is *symmetric*; otherwise it is *asymmetric*.
- An **ordinal attribute** is an attribute with possible values that have a meaningful order or ranking among them, but the magnitude between successive values is not known.
- A **numeric attribute** is *quantitative* (i.e., it is a measurable quantity) represented in integer or real values. Numeric attribute types can be *interval-scaled* or *ratio-scaled*. The values of an **interval-scaled attribute** are measured in fixed and equal units. **Ratio-scaled attributes** are numeric attributes with an inherent zero-point. Measurements are ratio-scaled in that we can speak of values as being an order of magnitude larger than the unit of measurement.
- **Basic statistical descriptions** provide the analytical foundation for data preprocessing. The basic statistical measures for data summarization include *mean, weighted mean, median*, and *mode* for measuring the central tendency of data; and *range, quantiles, quartiles, interquartile range, variance*, and *standard deviation* for measuring the dispersion of data. Graphical representations (e.g., *boxplots, quantile plots, quantile–quantile plots, histograms*, and *scatter plots*) facilitate visual inspection of the data and are thus useful for data preprocessing and mining.
- **Data visualization** techniques may be *pixel-oriented, geometric-based, icon-based*, or *hierarchical*. These methods apply to multidimensional relational data. Additional techniques have been proposed for the visualization of complex data, such as text and social networks.
- Measures of object **similarity** and **dissimilarity** are used in data mining applications such as clustering, outlier analysis, and nearest-neighbor classification. Such measures of *proximity* can be computed for each attribute type studied in this chapter, or for combinations of such attributes. Examples include the *Jaccard coefficient* for asymmetric binary attributes and *Euclidean*, *Manhattan*, *Minkowski*, and *supremum* distances for numeric attributes. For applications involving sparse numeric data vectors, such as term-frequency vectors, the *cosine measure* and the *Tanimoto coefficient* are often used in the assessment of similarity.

2.6 Exercises

2.1 Give three additional commonly used statistical measures that are not already illustrated in this chapter for the characterization of *data dispersion*. Discuss how they can be computed efficiently in large databases.

2.2 Suppose that the data for analysis includes the attribute *age*. The *age* values for the data tuples are (in increasing order) 13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25, 25, 25, 30, 33, 33, 35, 35, 35, 35, 36, 40, 45, 46, 52, 70.

(a) What is the *mean* of the data? What is the *median*?

(b) What is the *mode* of the data? Comment on the data's modality (i.e., bimodal, trimodal, etc.).

(c) What is the *midrange* of the data?

(d) Can you find (roughly) the first quartile (Q_1) and the third quartile (Q_3) of the data?

(e) Give the *five-number summary* of the data.

(f) Show a *boxplot* of the data.

(g) How is a *quantile–quantile plot* different from a *quantile plot*?

2.3 Suppose that the values for a given set of data are grouped into intervals. The intervals and corresponding frequencies are as follows:

age	*frequency*
1–5	200
6–15	450
16–20	300
21–50	1500
51–80	700
81–110	44

Compute an *approximate median* value for the data.

2.4 Suppose that a hospital tested the age and body fat data for 18 randomly selected adults with the following results:

age	23	23	27	27	39	41	47	49	50
%fat	9.5	26.5	7.8	17.8	31.4	25.9	27.4	27.2	31.2
age	52	54	54	56	57	58	58	60	61
%fat	34.6	42.5	28.8	33.4	30.2	34.1	32.9	41.2	35.7

(a) Calculate the mean, median, and standard deviation of *age* and *%fat*.

(b) Draw the boxplots for *age* and *%fat*.

(c) Draw a *scatter plot* and a *q-q plot* based on these two variables.

2.5 Briefly outline how to compute the dissimilarity between objects described by the following:

(a) Nominal attributes

(b) Asymmetric binary attributes

(c) Numeric attributes
(d) Term-frequency vectors

2.6 Given two objects represented by the tuples (22, 1, 42, 10) and (20, 0, 36, 8):

(a) Compute the *Euclidean distance* between the two objects.
(b) Compute the *Manhattan distance* between the two objects.
(c) Compute the *Minkowski distance* between the two objects, using $q = 3$.
(d) Compute the *supremum distance* between the two objects.

2.7 The *median* is one of the most important holistic measures in data analysis. Propose several methods for median approximation. Analyze their respective complexity under different parameter settings and decide to what extent the real value can be approximated. Moreover, suggest a heuristic strategy to balance between accuracy and complexity and then apply it to all methods you have given.

2.8 It is important to define or select similarity measures in data analysis. However, there is no commonly accepted subjective similarity measure. Results can vary depending on the similarity measures used. Nonetheless, seemingly different similarity measures may be equivalent after some transformation.

Suppose we have the following 2-D data set:

	A_1	A_2
$\boldsymbol{x_1}$	1.5	1.7
$\boldsymbol{x_2}$	2	1.9
$\boldsymbol{x_3}$	1.6	1.8
$\boldsymbol{x_4}$	1.2	1.5
$\boldsymbol{x_5}$	1.5	1.0

(a) Consider the data as 2-D data points. Given a new data point, $\boldsymbol{x} = (1.4, 1.6)$ as a query, rank the database points based on similarity with the query using Euclidean distance, Manhattan distance, supremum distance, and cosine similarity.
(b) Normalize the data set to make the norm of each data point equal to 1. Use Euclidean distance on the transformed data to rank the data points.

2.7 Bibliographic Notes

Methods for descriptive data summarization have been studied in the statistics literature long before the onset of computers. Good summaries of statistical descriptive data mining methods include Freedman, Pisani, and Purves [FPP07] and Devore [Dev95]. For

statistics-based visualization of data using boxplots, quantile plots, quantile–quantile plots, scatter plots, and loess curves, see Cleveland [Cle93].

Pioneering work on data visualization techniques is described in *The Visual Display of Quantitative Information* [Tuf83], *Envisioning Information* [Tuf90], and *Visual Explanations: Images and Quantities, Evidence and Narrative* [Tuf97], all by Tufte, in addition to *Graphics and Graphic Information Processing* by Bertin [Ber81], *Visualizing Data* by Cleveland [Cle93], and *Information Visualization in Data Mining and Knowledge Discovery* edited by Fayyad, Grinstein, and Wierse [FGW01].

Major conferences and symposiums on visualization include *ACM Human Factors in Computing Systems (CHI)*, *Visualization*, and the *International Symposium on Information Visualization*. Research on visualization is also published in *Transactions on Visualization and Computer Graphics*, *Journal of Computational and Graphical Statistics*, and *IEEE Computer Graphics and Applications*.

Many graphical user interfaces and visualization tools have been developed and can be found in various data mining products. Several books on data mining (e.g., *Data Mining Solutions* by Westphal and Blaxton [WB98]) present many good examples and visual snapshots. For a survey of visualization techniques, see "Visual techniques for exploring databases" by Keim [Kei97].

Similarity and distance measures among various variables have been introduced in many textbooks that study cluster analysis, including Hartigan [Har75]; Jain and Dubes [JD88]; Kaufman and Rousseeuw [KR90]; and Arabie, Hubert, and de Soete [AHS96]. Methods for combining attributes of different types into a single dissimilarity matrix were introduced by Kaufman and Rousseeuw [KR90].

3 Data Preprocessing

Today's real-world databases are highly susceptible to noisy, missing, and inconsistent data due to their typically huge size (often several gigabytes or more) and their likely origin from multiple, heterogenous sources. Low-quality data will lead to low-quality mining results. "*How can the data be preprocessed in order to help improve the quality of the data and, consequently, of the mining results? How can the data be preprocessed so as to improve the efficiency and ease of the mining process?*"

There are several data preprocessing techniques. *Data cleaning* can be applied to remove noise and correct inconsistencies in data. *Data integration* merges data from multiple sources into a coherent data store such as a data warehouse. *Data reduction* can reduce data size by, for instance, aggregating, eliminating redundant features, or clustering. *Data transformations* (e.g., normalization) may be applied, where data are scaled to fall within a smaller range like 0.0 to 1.0. This can improve the accuracy and efficiency of mining algorithms involving distance measurements. These techniques are not mutually exclusive; they may work together. For example, data cleaning can involve transformations to correct wrong data, such as by transforming all entries for a *date* field to a common format.

In Chapter 2, we learned about the different attribute types and how to use basic statistical descriptions to study data characteristics. These can help identify erroneous values and outliers, which will be useful in the data cleaning and integration steps. Data processing techniques, when applied before mining, can substantially improve the overall quality of the patterns mined and/or the time required for the actual mining.

In this chapter, we introduce the basic concepts of data preprocessing in Section 3.1. The methods for data preprocessing are organized into the following categories: data cleaning (Section 3.2), data integration (Section 3.3), data reduction (Section 3.4), and data transformation (Section 3.5).

3.1 Data Preprocessing: An Overview

This section presents an overview of data preprocessing. Section 3.1.1 illustrates the many elements defining data quality. This provides the incentive behind data preprocessing. Section 3.1.2 outlines the major tasks in data preprocessing.

3.1.1 Data Quality: Why Preprocess the Data?

Data have quality if they satisfy the requirements of the intended use. There are many factors comprising **data quality**, including *accuracy, completeness, consistency, timeliness, believability*, and *interpretability*.

Imagine that you are a manager at *AllElectronics* and have been charged with analyzing the company's data with respect to your branch's sales. You immediately set out to perform this task. You carefully inspect the company's database and data warehouse, identifying and selecting the attributes or dimensions (e.g., *item*, *price*, and *units_sold*) to be included in your analysis. Alas! You notice that several of the attributes for various tuples have no recorded value. For your analysis, you would like to include information as to whether each item purchased was advertised as on sale, yet you discover that this information has not been recorded. Furthermore, users of your database system have reported errors, unusual values, and inconsistencies in the data recorded for some transactions. In other words, the data you wish to analyze by data mining techniques are *incomplete* (lacking attribute values or certain attributes of interest, or containing only aggregate data); *inaccurate* or *noisy* (containing errors, or values that deviate from the expected); and *inconsistent* (e.g., containing discrepancies in the department codes used to categorize items). Welcome to the real world!

This scenario illustrates three of the elements defining data quality: **accuracy**, **completeness**, and **consistency**. Inaccurate, incomplete, and inconsistent data are commonplace properties of large real-world databases and data warehouses. There are many possible reasons for inaccurate data (i.e., having incorrect attribute values). The data collection instruments used may be faulty. There may have been human or computer errors occurring at data entry. Users may purposely submit incorrect data values for mandatory fields when they do not wish to submit personal information (e.g., by choosing the default value "January 1" displayed for birthday). This is known as *disguised missing data*. Errors in data transmission can also occur. There may be technology limitations such as limited buffer size for coordinating synchronized data transfer and consumption. Incorrect data may also result from inconsistencies in naming conventions or data codes, or inconsistent formats for input fields (e.g., *date*). Duplicate tuples also require data cleaning.

Incomplete data can occur for a number of reasons. Attributes of interest may not always be available, such as customer information for sales transaction data. Other data may not be included simply because they were not considered important at the time of entry. Relevant data may not be recorded due to a misunderstanding or because of equipment malfunctions. Data that were inconsistent with other recorded data may

have been deleted. Furthermore, the recording of the data history or modifications may have been overlooked. Missing data, particularly for tuples with missing values for some attributes, may need to be inferred.

Recall that data quality depends on the intended use of the data. Two different users may have very different assessments of the quality of a given database. For example, a marketing analyst may need to access the database mentioned before for a list of customer addresses. Some of the addresses are outdated or incorrect, yet overall, 80% of the addresses are accurate. The marketing analyst considers this to be a large customer database for target marketing purposes and is pleased with the database's accuracy, although, as sales manager, you found the data inaccurate.

Timeliness also affects data quality. Suppose that you are overseeing the distribution of monthly sales bonuses to the top sales representatives at *AllElectronics*. Several sales representatives, however, fail to submit their sales records on time at the end of the month. There are also a number of corrections and adjustments that flow in after the month's end. For a period of time following each month, the data stored in the database are incomplete. However, once all of the data are received, it is correct. The fact that the month-end data are not updated in a timely fashion has a negative impact on the data quality.

Two other factors affecting data quality are believability and interpretability. **Believability** reflects how much the data are trusted by users, while **interpretability** reflects how easy the data are understood. Suppose that a database, at one point, had several errors, all of which have since been corrected. The past errors, however, had caused many problems for sales department users, and so they no longer trust the data. The data also use many accounting codes, which the sales department does not know how to interpret. Even though the database is now accurate, complete, consistent, and timely, sales department users may regard it as of low quality due to poor believability and interpretability.

3.1.2 Major Tasks in Data Preprocessing

In this section, we look at the major steps involved in data preprocessing, namely, data cleaning, data integration, data reduction, and data transformation.

Data cleaning routines work to "clean" the data by filling in missing values, smoothing noisy data, identifying or removing outliers, and resolving inconsistencies. If users believe the data are dirty, they are unlikely to trust the results of any data mining that has been applied. Furthermore, dirty data can cause confusion for the mining procedure, resulting in unreliable output. Although most mining routines have some procedures for dealing with incomplete or noisy data, they are not always robust. Instead, they may concentrate on avoiding overfitting the data to the function being modeled. Therefore, a useful preprocessing step is to run your data through some data cleaning routines. Section 3.2 discusses methods for data cleaning.

Getting back to your task at *AllElectronics*, suppose that you would like to include data from multiple sources in your analysis. This would involve integrating multiple databases, data cubes, or files (i.e., **data integration**). Yet some attributes representing a

given concept may have different names in different databases, causing inconsistencies and redundancies. For example, the attribute for customer identification may be referred to as *customer_id* in one data store and *cust_id* in another. Naming inconsistencies may also occur for attribute values. For example, the same first name could be registered as "Bill" in one database, "William" in another, and "B." in a third. Furthermore, you suspect that some attributes may be inferred from others (e.g., annual revenue). Having a large amount of redundant data may slow down or confuse the knowledge discovery process. Clearly, in addition to data cleaning, steps must be taken to help avoid redundancies during data integration. Typically, data cleaning and data integration are performed as a preprocessing step when preparing data for a data warehouse. Additional data cleaning can be performed to detect and remove redundancies that may have resulted from data integration.

"*Hmmm,*" you wonder, as you consider your data even further. "*The data set I have selected for analysis is HUGE, which is sure to slow down the mining process. Is there a way I can reduce the size of my data set without jeopardizing the data mining results?*" **Data reduction** obtains a reduced representation of the data set that is much smaller in volume, yet produces the same (or almost the same) analytical results. Data reduction strategies include *dimensionality reduction* and *numerosity reduction*.

In **dimensionality reduction**, data encoding schemes are applied so as to obtain a reduced or "compressed" representation of the original data. Examples include data compression techniques (e.g., *wavelet transforms* and *principal components analysis*), *attribute subset selection* (e.g., removing irrelevant attributes), and *attribute construction* (e.g., where a small set of more useful attributes is derived from the original set).

In **numerosity reduction**, the data are replaced by alternative, smaller representations using parametric models (e.g., *regression* or *log-linear models*) or nonparametric models (e.g., *histograms, clusters*, *sampling*, or *data aggregation*). Data reduction is the topic of Section 3.4.

Getting back to your data, you have decided, say, that you would like to use a distance-based mining algorithm for your analysis, such as neural networks, nearest-neighbor classifiers, or clustering.[1] Such methods provide better results if the data to be analyzed have been *normalized*, that is, scaled to a smaller range such as [0.0, 1.0]. Your customer data, for example, contain the attributes *age* and *annual salary*. The *annual salary* attribute usually takes much larger values than *age*. Therefore, if the attributes are left unnormalized, the distance measurements taken on *annual salary* will generally outweigh distance measurements taken on *age*. *Discretization* and *concept hierarchy generation* can also be useful, where raw data values for attributes are replaced by ranges or higher conceptual levels. For example, raw values for *age* may be replaced by higher-level concepts, such as *youth*, *adult*, or *senior*.

Discretization and concept hierarchy generation are powerful tools for data mining in that they allow data mining at multiple abstraction levels. Normalization, data

[1] Neural networks and nearest-neighbor classifiers are described in Chapter 9, and clustering is discussed in Chapters 10 and 11.

discretization, and concept hierarchy generation are forms of **data transformation**. You soon realize such data transformation operations are additional data preprocessing procedures that would contribute toward the success of the mining process. Data integration and data discretization are discussed in Sections 3.5.

Figure 3.1 summarizes the data preprocessing steps described here. Note that the previous categorization is not mutually exclusive. For example, the removal of redundant data may be seen as a form of data cleaning, as well as data reduction.

In summary, real-world data tend to be dirty, incomplete, and inconsistent. Data preprocessing techniques can improve data quality, thereby helping to improve the accuracy and efficiency of the subsequent mining process. Data preprocessing is an important step in the knowledge discovery process, because quality decisions must be based on quality data. Detecting data anomalies, rectifying them early, and reducing the data to be analyzed can lead to huge payoffs for decision making.

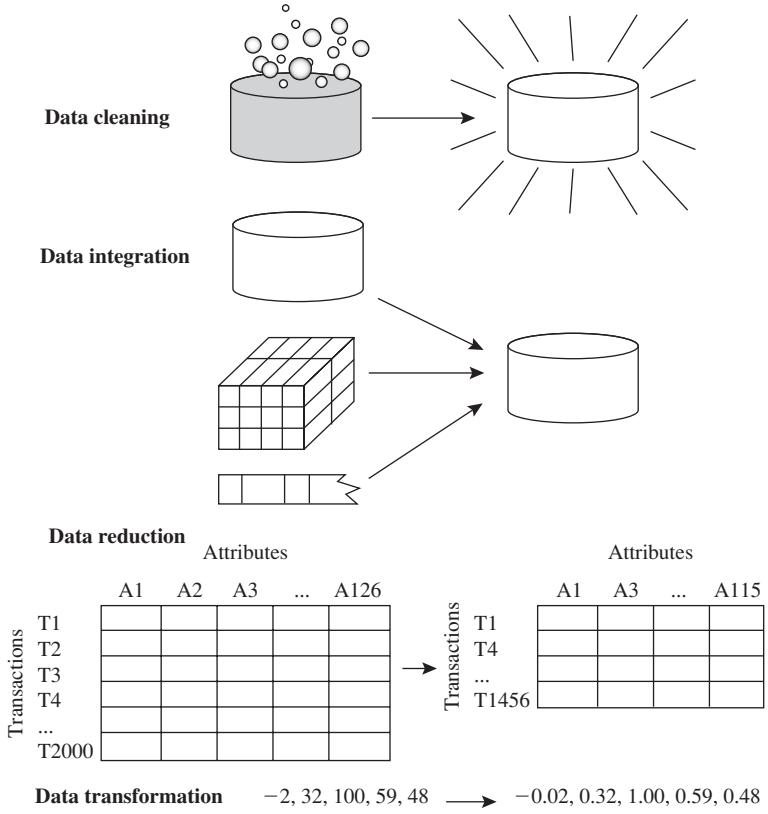


Figure 3.1 Forms of data preprocessing.

3.2 Data Cleaning

Real-world data tend to be incomplete, noisy, and inconsistent. *Data cleaning* (or *data cleansing*) routines attempt to fill in missing values, smooth out noise while identifying outliers, and correct inconsistencies in the data. In this section, you will study basic methods for data cleaning. Section 3.2.1 looks at ways of handling missing values. Section 3.2.2 explains data smoothing techniques. Section 3.2.3 discusses approaches to data cleaning as a process.

3.2.1 Missing Values

Imagine that you need to analyze *AllElectronics* sales and customer data. You note that many tuples have no recorded value for several attributes such as customer *income*. How can you go about filling in the missing values for this attribute? Let's look at the following methods.

1. **Ignore the tuple**: This is usually done when the class label is missing (assuming the mining task involves classification). This method is not very effective, unless the tuple contains several attributes with missing values. It is especially poor when the percentage of missing values per attribute varies considerably. By ignoring the tuple, we do not make use of the remaining attributes' values in the tuple. Such data could have been useful to the task at hand.
2. **Fill in the missing value manually**: In general, this approach is time consuming and may not be feasible given a large data set with many missing values.
3. **Use a global constant to fill in the missing value**: Replace all missing attribute values by the same constant such as a label like "*Unknown*" or $-\infty$. If missing values are replaced by, say, "*Unknown*," then the mining program may mistakenly think that they form an interesting concept, since they all have a value in common—that of "*Unknown*." Hence, although this method is simple, it is not foolproof.
4. **Use a measure of central tendency for the attribute (e.g., the mean or median) to fill in the missing value**: Chapter 2 discussed measures of central tendency, which indicate the "middle" value of a data distribution. For normal (symmetric) data distributions, the mean can be used, while skewed data distribution should employ the median (Section 2.2). For example, suppose that the data distribution regarding the income of *AllElectronics* customers is symmetric and that the mean income is $56,000. Use this value to replace the missing value for *income*.
5. **Use the attribute mean or median for all samples belonging to the same class as the given tuple**: For example, if classifying customers according to *credit_risk*, we may replace the missing value with the mean *income* value for customers in the same credit risk category as that of the given tuple. If the data distribution for a given class is skewed, the median value is a better choice.
6. **Use the most probable value to fill in the missing value**: This may be determined with regression, inference-based tools using a Bayesian formalism, or decision tree

induction. For example, using the other customer attributes in your data set, you may construct a decision tree to predict the missing values for *income*. Decision trees and Bayesian inference are described in detail in Chapters 8 and 9, respectively, while regression is introduced in Section 3.4.5.

Methods 3 through 6 bias the data—the filled-in value may not be correct. Method 6, however, is a popular strategy. In comparison to the other methods, it uses the most information from the present data to predict missing values. By considering the other attributes' values in its estimation of the missing value for *income*, there is a greater chance that the relationships between *income* and the other attributes are preserved.

It is important to note that, in some cases, a missing value may not imply an error in the data! For example, when applying for a credit card, candidates may be asked to supply their driver's license number. Candidates who do not have a driver's license may naturally leave this field blank. Forms should allow respondents to specify values such as "not applicable." Software routines may also be used to uncover other null values (e.g., "don't know," "?" or "none"). Ideally, each attribute should have one or more rules regarding the *null* condition. The rules may specify whether or not nulls are allowed and/or how such values should be handled or transformed. Fields may also be intentionally left blank if they are to be provided in a later step of the business process. Hence, although we can try our best to clean the data after it is seized, good database and data entry procedure design should help minimize the number of missing values or errors in the first place.

3.2.2 Noisy Data

"*What is noise?*" **Noise** is a random error or variance in a measured variable. In Chapter 2, we saw how some basic statistical description techniques (e.g., boxplots and scatter plots), and methods of data visualization can be used to identify outliers, which may represent noise. Given a numeric attribute such as, say, *price*, how can we "smooth" out the data to remove the noise? Let's look at the following data smoothing techniques.

Binning: Binning methods smooth a sorted data value by consulting its "neighborhood," that is, the values around it. The sorted values are distributed into a number of "buckets," or *bins*. Because binning methods consult the neighborhood of values, they perform *local* smoothing. Figure 3.2 illustrates some binning techniques. In this example, the data for *price* are first sorted and then partitioned into *equal-frequency* bins of size 3 (i.e., each bin contains three values). In **smoothing by bin means**, each value in a bin is replaced by the mean value of the bin. For example, the mean of the values 4, 8, and 15 in Bin 1 is 9. Therefore, each original value in this bin is replaced by the value 9.

Similarly, **smoothing by bin medians** can be employed, in which each bin value is replaced by the bin median. In **smoothing by bin boundaries**, the minimum and maximum values in a given bin are identified as the *bin boundaries*. Each bin value is then replaced by the closest boundary value. In general, the larger the width, the

Sorted data for *price* (in dollars): 4, 8, 15, 21, 21, 24, 25, 28, 34

Partition into (equal-frequency) bins:
Bin 1: 4, 8, 15
Bin 2: 21, 21, 24
Bin 3: 25, 28, 34

Smoothing by bin means:
Bin 1: 9, 9, 9
Bin 2: 22, 22, 22
Bin 3: 29, 29, 29

Smoothing by bin boundaries:
Bin 1: 4, 4, 15
Bin 2: 21, 21, 24
Bin 3: 25, 25, 34

Figure 3.2 Binning methods for data smoothing.

greater the effect of the smoothing. Alternatively, bins may be *equal width*, where the interval range of values in each bin is constant. Binning is also used as a discretization technique and is further discussed in Section 3.5.

Regression: Data smoothing can also be done by regression, a technique that conforms data values to a function. *Linear regression* involves finding the "best" line to fit two attributes (or variables) so that one attribute can be used to predict the other. *Multiple linear regression* is an extension of linear regression, where more than two attributes are involved and the data are fit to a multidimensional surface. Regression is further described in Section 3.4.5.

Outlier analysis: Outliers may be detected by clustering, for example, where similar values are organized into groups, or "clusters." Intuitively, values that fall outside of the set of clusters may be considered outliers (Figure 3.3). Chapter 12 is dedicated to the topic of outlier analysis.

Many data smoothing methods are also used for data discretization (a form of data transformation) and data reduction. For example, the binning techniques described before reduce the number of distinct values per attribute. This acts as a form of data reduction for logic-based data mining methods, such as decision tree induction, which repeatedly makes value comparisons on sorted data. Concept hierarchies are a form of data discretization that can also be used for data smoothing. A concept hierarchy for *price*, for example, may map real *price* values into *inexpensive, moderately_priced*, and *expensive*, thereby reducing the number of data values to be handled by the mining

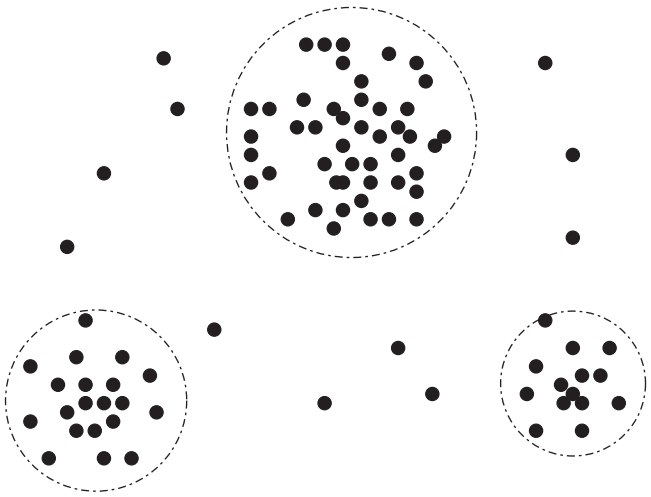


Figure 3.3 A 2-D customer data plot with respect to customer locations in a city, showing three data clusters. Outliers may be detected as values that fall outside of the cluster sets.

process. Data discretization is discussed in Section 3.5. Some methods of classification (e.g., neural networks) have built-in data smoothing mechanisms. Classification is the topic of Chapters 8 and 9.

3.2.3 Data Cleaning as a Process

Missing values, noise, and inconsistencies contribute to inaccurate data. So far, we have looked at techniques for handling missing data and for smoothing data. "*But data cleaning is a big job. What about data cleaning as a process? How exactly does one proceed in tackling this task? Are there any tools out there to help?*"

The first step in data cleaning as a process is *discrepancy detection*. Discrepancies can be caused by several factors, including poorly designed data entry forms that have many optional fields, human error in data entry, deliberate errors (e.g., respondents not wanting to divulge information about themselves), and data decay (e.g., outdated addresses). Discrepancies may also arise from inconsistent data representations and inconsistent use of codes. Other sources of discrepancies include errors in instrumentation devices that record data and system errors. Errors can also occur when the data are (inadequately) used for purposes other than originally intended. There may also be inconsistencies due to data integration (e.g., where a given attribute can have different names in different databases).[2]

[2]Data integration and the removal of redundant data that can result from such integration are further described in Section 3.3.

"*So, how can we proceed with discrepancy detection?*" As a starting point, use any knowledge you may already have regarding properties of the data. Such knowledge or "data about data" is referred to as **metadata**. This is where we can make use of the knowledge we gained about our data in Chapter 2. For example, what are the data type and domain of each attribute? What are the acceptable values for each attribute? The basic statistical data descriptions discussed in Section 2.2 are useful here to grasp data trends and identify anomalies. For example, find the mean, median, and mode values. Are the data symmetric or skewed? What is the range of values? Do all values fall within the expected range? What is the standard deviation of each attribute? Values that are more than two standard deviations away from the mean for a given attribute may be flagged as potential outliers. Are there any known dependencies between attributes? In this step, you may write your own scripts and/or use some of the tools that we discuss further later. From this, you may find noise, outliers, and unusual values that need investigation.

As a data analyst, you should be on the lookout for the inconsistent use of codes and any inconsistent data representations (e.g., "2010/12/25" and "25/12/2010" for *date*). **Field overloading** is another error source that typically results when developers squeeze new attribute definitions into unused (bit) portions of already defined attributes (e.g., an unused bit of an attribute that has a value range that uses only, say, 31 out of 32 bits).

The data should also be examined regarding unique rules, consecutive rules, and null rules. A **unique rule** says that each value of the given attribute must be different from all other values for that attribute. A **consecutive rule** says that there can be no missing values between the lowest and highest values for the attribute, and that all values must also be unique (e.g., as in check numbers). A **null rule** specifies the use of blanks, question marks, special characters, or other strings that may indicate the null condition (e.g., where a value for a given attribute is not available), and how such values should be handled. As mentioned in Section 3.2.1, reasons for missing values may include (1) the person originally asked to provide a value for the attribute refuses and/or finds that the information requested is not applicable (e.g., a *license_number* attribute left blank by nondrivers); (2) the data entry person does not know the correct value; or (3) the value is to be provided by a later step of the process. The null rule should specify how to record the null condition, for example, such as to store zero for numeric attributes, a blank for character attributes, or any other conventions that may be in use (e.g., entries like "don't know" or "?" should be transformed to blank).

There are a number of different commercial tools that can aid in the discrepancy detection step. **Data scrubbing tools** use simple domain knowledge (e.g., knowledge of postal addresses and spell-checking) to detect errors and make corrections in the data. These tools rely on parsing and fuzzy matching techniques when cleaning data from multiple sources. **Data auditing tools** find discrepancies by analyzing the data to discover rules and relationships, and detecting data that violate such conditions. They are variants of data mining tools. For example, they may employ statistical analysis to find correlations, or clustering to identify outliers. They may also use the basic statistical data descriptions presented in Section 2.2.

Some data inconsistencies may be corrected manually using external references. For example, errors made at data entry may be corrected by performing a paper

trace. Most errors, however, will require *data transformations*. That is, once we find discrepancies, we typically need to define and apply (a series of) transformations to correct them.

Commercial tools can assist in the data transformation step. **Data migration tools** allow simple transformations to be specified such as to replace the string *"gender"* by *"sex."* **ETL (extraction/transformation/loading) tools** allow users to specify transforms through a graphical user interface (GUI). These tools typically support only a restricted set of transforms so that, often, we may also choose to write custom scripts for this step of the data cleaning process.

The two-step process of discrepancy detection and data transformation (to correct discrepancies) iterates. This process, however, is error-prone and time consuming. Some transformations may introduce more discrepancies. Some *nested discrepancies* may only be detected after others have been fixed. For example, a typo such as "20010" in a year field may only surface once all date values have been converted to a uniform format. Transformations are often done as a batch process while the user waits without feedback. Only after the transformation is complete can the user go back and check that no new anomalies have been mistakenly created. Typically, numerous iterations are required before the user is satisfied. Any tuples that cannot be automatically handled by a given transformation are typically written to a file without any explanation regarding the reasoning behind their failure. As a result, the entire data cleaning process also suffers from a lack of interactivity.

New approaches to data cleaning emphasize increased interactivity. Potter's Wheel, for example, is a publicly available data cleaning tool that integrates discrepancy detection and transformation. Users gradually build a series of transformations by composing and debugging individual transformations, one step at a time, on a spreadsheet-like interface. The transformations can be specified graphically or by providing examples. Results are shown immediately on the records that are visible on the screen. The user can choose to undo the transformations, so that transformations that introduced additional errors can be "erased." The tool automatically performs discrepancy checking in the background on the latest transformed view of the data. Users can gradually develop and refine transformations as discrepancies are found, leading to more effective and efficient data cleaning.

Another approach to increased interactivity in data cleaning is the development of declarative languages for the specification of data transformation operators. Such work focuses on defining powerful extensions to SQL and algorithms that enable users to express data cleaning specifications efficiently.

As we discover more about the data, it is important to keep updating the metadata to reflect this knowledge. This will help speed up data cleaning on future versions of the same data store.

3.3 Data Integration

Data mining often requires data integration—the merging of data from multiple data stores. Careful integration can help reduce and avoid redundancies and inconsistencies

in the resulting data set. This can help improve the accuracy and speed of the subsequent data mining process.

The semantic heterogeneity and structure of data pose great challenges in data integration. How can we match schema and objects from different sources? This is the essence of the *entity identification problem*, described in Section 3.3.1. Are any attributes correlated? Section 3.3.2 presents correlation tests for numeric and nominal data. Tuple duplication is described in Section 3.3.3. Finally, Section 3.3.4 touches on the detection and resolution of data value conflicts.

3.3.1 Entity Identification Problem

It is likely that your data analysis task will involve *data integration*, which combines data from multiple sources into a coherent data store, as in data warehousing. These sources may include multiple databases, data cubes, or flat files.

There are a number of issues to consider during data integration. *Schema integration* and *object matching* can be tricky. How can equivalent real-world entities from multiple data sources be matched up? This is referred to as the **entity identification problem**. For example, how can the data analyst or the computer be sure that *customer_id* in one database and *cust_number* in another refer to the same attribute? Examples of metadata for each attribute include the name, meaning, data type, and range of values permitted for the attribute, and null rules for handling blank, zero, or null values (Section 3.2). Such metadata can be used to help avoid errors in schema integration. The metadata may also be used to help transform the data (e.g., where data codes for *pay_type* in one database may be *"H"* and *"S"* but *1* and *2* in another). Hence, this step also relates to data cleaning, as described earlier.

When matching attributes from one database to another during integration, special attention must be paid to the *structure* of the data. This is to ensure that any attribute functional dependencies and referential constraints in the source system match those in the target system. For example, in one system, a *discount* may be applied to the order, whereas in another system it is applied to each individual line item within the order. If this is not caught before integration, items in the target system may be improperly discounted.

3.3.2 Redundancy and Correlation Analysis

Redundancy is another important issue in data integration. An attribute (such as *annual revenue*, for instance) may be redundant if it can be "derived" from another attribute or set of attributes. Inconsistencies in attribute or dimension naming can also cause redundancies in the resulting data set.

Some redundancies can be detected by **correlation analysis**. Given two attributes, such analysis can measure how strongly one attribute implies the other, based on the available data. For nominal data, we use the χ^2 (*chi-square*) test. For numeric attributes, we can use the *correlation coefficient* and *covariance*, both of which access how one attribute's values vary from those of another.

χ^2 Correlation Test for Nominal Data

For nominal data, a correlation relationship between two attributes, A and B, can be discovered by a χ^2 (**chi-square**) test. Suppose A has c distinct values, namely $a_1, a_2, \ldots a_c$. B has r distinct values, namely $b_1, b_2, \ldots b_r$. The data tuples described by A and B can be shown as a **contingency table**, with the c values of A making up the columns and the r values of B making up the rows. Let (A_i, B_j) denote the joint event that attribute A takes on value a_i and attribute B takes on value b_j, that is, where $(A = a_i, B = b_j)$. Each and every possible (A_i, B_j) joint event has its own cell (or slot) in the table. The χ^2 value (also known as the *Pearson* χ^2 *statistic*) is computed as

$$\chi^2 = \sum_{i=1}^{c} \sum_{j=1}^{r} \frac{(o_{ij} - e_{ij})^2}{e_{ij}}, \tag{3.1}$$

where o_{ij} is the *observed frequency* (i.e., actual count) of the joint event (A_i, B_j) and e_{ij} is the *expected frequency* of (A_i, B_j), which can be computed as

$$e_{ij} = \frac{count(A = a_i) \times count(B = b_j)}{n}, \tag{3.2}$$

where n is the number of data tuples, $count(A = a_i)$ is the number of tuples having value a_i for A, and $count(B = b_j)$ is the number of tuples having value b_j for B. The sum in Eq. (3.1) is computed over all of the $r \times c$ cells. Note that the cells that contribute the most to the χ^2 value are those for which the actual count is very different from that expected.

The χ^2 statistic tests the hypothesis that A and B are *independent*, that is, there is no correlation between them. The test is based on a significance level, with $(r - 1) \times (c - 1)$ degrees of freedom. We illustrate the use of this statistic in Example 3.1. If the hypothesis can be rejected, then we say that A and B are statistically correlated.

Example 3.1 **Correlation analysis of nominal attributes using χ^2.** Suppose that a group of 1500 people was surveyed. The gender of each person was noted. Each person was polled as to whether his or her preferred type of reading material was fiction or nonfiction. Thus, we have two attributes, *gender* and *preferred_reading*. The observed frequency (or count) of each possible joint event is summarized in the contingency table shown in Table 3.1, where the numbers in parentheses are the expected frequencies. The expected frequencies are calculated based on the data distribution for both attributes using Eq. (3.2).

Using Eq. (3.2), we can verify the expected frequencies for each cell. For example, the expected frequency for the cell (*male, fiction*) is

$$e_{11} = \frac{count(male) \times count(fiction)}{n} = \frac{300 \times 450}{1500} = 90,$$

and so on. Notice that in any row, the sum of the expected frequencies must equal the total observed frequency for that row, and the sum of the expected frequencies in any column must also equal the total observed frequency for that column.

Table 3.1 Example 2.1's 2 × 2 Contingency Table Data

	male	*female*	*Total*
fiction	250 (90)	200 (360)	450
non_fiction	50 (210)	1000 (840)	1050
Total	300	1200	1500

Note: Are *gender* and *preferred_reading* correlated?

Using Eq. (3.1) for χ^2 computation, we get

$$\chi^2 = \frac{(250-90)^2}{90} + \frac{(50-210)^2}{210} + \frac{(200-360)^2}{360} + \frac{(1000-840)^2}{840}$$
$$= 284.44 + 121.90 + 71.11 + 30.48 = 507.93.$$

For this 2 × 2 table, the degrees of freedom are $(2-1)(2-1) = 1$. For 1 degree of freedom, the χ^2 value needed to reject the hypothesis at the 0.001 significance level is 10.828 (taken from the table of upper percentage points of the χ^2 distribution, typically available from any textbook on statistics). Since our computed value is above this, we can reject the hypothesis that *gender* and *preferred_reading* are independent and conclude that the two attributes are (strongly) correlated for the given group of people. ■

Correlation Coefficient for Numeric Data

For numeric attributes, we can evaluate the correlation between two attributes, A and B, by computing the **correlation coefficient** (also known as **Pearson's product moment coefficient**, named after its inventer, Karl Pearson). This is

$$r_{A,B} = \frac{\sum_{i=1}^{n}(a_i - \bar{A})(b_i - \bar{B})}{n\sigma_A\sigma_B} = \frac{\sum_{i=1}^{n}(a_i b_i) - n\bar{A}\bar{B}}{n\sigma_A\sigma_B}, \tag{3.3}$$

where n is the number of tuples, a_i and b_i are the respective values of A and B in tuple i, $\bar{A}$ and $\bar{B}$ are the respective mean values of A and B, σ_A and σ_B are the respective standard deviations of A and B (as defined in Section 2.2.2), and $\Sigma(a_i b_i)$ is the sum of the AB cross-product (i.e., for each tuple, the value for A is multiplied by the value for B in that tuple). Note that $-1 \leq r_{A,B} \leq +1$. If $r_{A,B}$ is greater than 0, then A and B are *positively correlated*, meaning that the values of A increase as the values of B increase. The higher the value, the stronger the correlation (i.e., the more each attribute implies the other). Hence, a higher value may indicate that A (or B) may be removed as a redundancy.

If the resulting value is equal to 0, then A and B are *independent* and there is no correlation between them. If the resulting value is less than 0, then A and B are *negatively correlated*, where the values of one attribute increase as the values of the other attribute decrease. This means that each attribute discourages the other. Scatter plots can also be used to view correlations between attributes (Section 2.2.3). For example, Figure 2.8's

scatter plots respectively show positively correlated data and negatively correlated data, while Figure 2.9 displays uncorrelated data.

Note that correlation does not imply causality. That is, if A and B are correlated, this does not necessarily imply that A causes B or that B causes A. For example, in analyzing a demographic database, we may find that attributes representing the number of hospitals and the number of car thefts in a region are correlated. This does not mean that one causes the other. Both are actually causally linked to a third attribute, namely, *population*.

Covariance of Numeric Data

In probability theory and statistics, correlation and covariance are two similar measures for assessing how much two attributes change together. Consider two numeric attributes A and B, and a set of n observations $\{(a_1, b_1), \ldots, (a_n, b_n)\}$. The mean values of A and B, respectively, are also known as the **expected values** on A and B, that is,

$$E(A) = \bar{A} = \frac{\sum_{i=1}^{n} a_i}{n}$$

and

$$E(B) = \bar{B} = \frac{\sum_{i=1}^{n} b_i}{n}.$$

The **covariance** between A and B is defined as

$$Cov(A, B) = E((A - \bar{A})(B - \bar{B})) = \frac{\sum_{i=1}^{n} (a_i - \bar{A})(b_i - \bar{B})}{n}. \tag{3.4}$$

If we compare Eq. (3.3) for $r_{A,B}$ (correlation coefficient) with Eq. (3.4) for covariance, we see that

$$r_{A,B} = \frac{Cov(A, B)}{\sigma_A \sigma_B}, \tag{3.5}$$

where σ_A and σ_B are the standard deviations of A and B, respectively. It can also be shown that

$$Cov(A, B) = E(A \cdot B) - \bar{A}\bar{B}. \tag{3.6}$$

This equation may simplify calculations.

For two attributes A and B that tend to change together, if A is larger than $\bar{A}$ (the expected value of A), then B is likely to be larger than $\bar{B}$ (the expected value of B). Therefore, the covariance between A and B is *positive*. On the other hand, if one of the attributes tends to be above its expected value when the other attribute is below its expected value, then the covariance of A and B is *negative*.

If A and B are *independent* (i.e., they do not have correlation), then $E(A \cdot B) = E(A) \cdot E(B)$. Therefore, the covariance is $Cov(A, B) = E(A \cdot B) - \bar{A}\bar{B} = E(A) \cdot E(B) - \bar{A}\bar{B} = 0$. However, the converse is not true. Some pairs of random variables (attributes) may have a covariance of 0 but are not independent. Only under some additional assumptions

Table 3.2 Stock Prices for *AllElectronics* and *HighTech*

Time point	*AllElectronics*	*HighTech*
t1	6	20
t2	5	10
t3	4	14
t4	3	5
t5	2	5

(e.g., the data follow multivariate normal distributions) does a covariance of 0 imply independence.

Example 3.2 **Covariance analysis of numeric attributes.** Consider Table 3.2, which presents a simplified example of stock prices observed at five time points for *AllElectronics* and *HighTech*, a high-tech company. If the stocks are affected by the same industry trends, will their prices rise or fall together?

$$E(AllElectronics) = \frac{6+5+4+3+2}{5} = \frac{20}{5} = \$4$$

and

$$E(HighTech) = \frac{20+10+14+5+5}{5} = \frac{54}{5} = \$10.80.$$

Thus, using Eq. (3.4), we compute

$$Cov(AllElectroncis, HighTech) = \frac{6\times 20+5\times 10+4\times 14+3\times 5+2\times 5}{5} - 4\times 10.80$$
$$= 50.2 - 43.2 = 7.$$

Therefore, given the positive covariance we can say that stock prices for both companies rise together. ■

Variance is a special case of covariance, where the two attributes are identical (i.e., the covariance of an attribute with itself). Variance was discussed in Chapter 2.

3.3.3 Tuple Duplication

In addition to detecting redundancies between attributes, duplication should also be detected at the tuple level (e.g., where there are two or more identical tuples for a given unique data entry case). The use of denormalized tables (often done to improve performance by avoiding joins) is another source of data redundancy. Inconsistencies often arise between various duplicates, due to inaccurate data entry or updating some but not all data occurrences. For example, if a purchase order database contains attributes for

the purchaser's name and address instead of a key to this information in a purchaser database, discrepancies can occur, such as the same purchaser's name appearing with different addresses within the purchase order database.

3.3.4 Data Value Conflict Detection and Resolution

Data integration also involves the *detection and resolution of data value conflicts*. For example, for the same real-world entity, attribute values from different sources may differ. This may be due to differences in representation, scaling, or encoding. For instance, a *weight* attribute may be stored in metric units in one system and British imperial units in another. For a hotel chain, the *price* of rooms in different cities may involve not only different currencies but also different services (e.g., free breakfast) and taxes. When exchanging information between schools, for example, each school may have its own curriculum and grading scheme. One university may adopt a quarter system, offer three courses on database systems, and assign grades from A+ to F, whereas another may adopt a semester system, offer two courses on databases, and assign grades from 1 to 10. It is difficult to work out precise course-to-grade transformation rules between the two universities, making information exchange difficult.

Attributes may also differ on the abstraction level, where an attribute in one system is recorded at, say, a lower abstraction level than the "same" attribute in another. For example, the *total_sales* in one database may refer to one branch of *All_Electronics*, while an attribute of the same name in another database may refer to the total sales for *All_Electronics* stores in a given region. The topic of discrepancy detection is further described in Section 3.2.3 on data cleaning as a process.

3.4 Data Reduction

Imagine that you have selected data from the *AllElectronics* data warehouse for analysis. The data set will likely be huge! Complex data analysis and mining on huge amounts of data can take a long time, making such analysis impractical or infeasible.

Data reduction techniques can be applied to obtain a reduced representation of the data set that is much smaller in volume, yet closely maintains the integrity of the original data. That is, mining on the reduced data set should be more efficient yet produce the same (or almost the same) analytical results. In this section, we first present an overview of data reduction strategies, followed by a closer look at individual techniques.

3.4.1 Overview of Data Reduction Strategies

Data reduction strategies include *dimensionality reduction*, *numerosity reduction*, and *data compression*.

Dimensionality reduction is the process of reducing the number of random variables or attributes under consideration. Dimensionality reduction methods include *wavelet*

transforms (Section 3.4.2) and *principal components analysis* (Section 3.4.3), which transform or project the original data onto a smaller space. *Attribute subset selection* is a method of dimensionality reduction in which irrelevant, weakly relevant, or redundant attributes or dimensions are detected and removed (Section 3.4.4).

Numerosity reduction techniques replace the original data volume by alternative, smaller forms of data representation. These techniques may be parametric or nonparametric. For *parametric methods*, a model is used to estimate the data, so that typically only the data parameters need to be stored, instead of the actual data. (Outliers may also be stored.) Regression and log-linear models (Section 3.4.5) are examples. *Nonparametric methods* for storing reduced representations of the data include *histograms* (Section 3.4.6), *clustering* (Section 3.4.7), *sampling* (Section 3.4.8), and *data cube aggregation* (Section 3.4.9).

In **data compression**, transformations are applied so as to obtain a reduced or "compressed" representation of the original data. If the original data can be *reconstructed* from the compressed data without any information loss, the data reduction is called **lossless**. If, instead, we can reconstruct only an approximation of the original data, then the data reduction is called **lossy**. There are several lossless algorithms for string compression; however, they typically allow only limited data manipulation. Dimensionality reduction and numerosity reduction techniques can also be considered forms of data compression.

There are many other ways of organizing methods of data reduction. The computational time spent on data reduction should not outweigh or "erase" the time saved by mining on a reduced data set size.

3.4.2 Wavelet Transforms

The **discrete wavelet transform** (**DWT**) is a linear signal processing technique that, when applied to a data vector $\boldsymbol{X}$, transforms it to a numerically different vector, $\boldsymbol{X}'$, of **wavelet coefficients**. The two vectors are of the same length. When applying this technique to data reduction, we consider each tuple as an n-dimensional data vector, that is, $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$, depicting n measurements made on the tuple from n database attributes.[3]

"How can this technique be useful for data reduction if the wavelet transformed data are of the same length as the original data?" The usefulness lies in the fact that the wavelet transformed data can be truncated. A compressed approximation of the data can be retained by storing only a small fraction of the strongest of the wavelet coefficients. For example, all wavelet coefficients larger than some user-specified threshold can be retained. All other coefficients are set to 0. The resulting data representation is therefore very sparse, so that operations that can take advantage of data sparsity are computationally very fast if performed in wavelet space. The technique also works to remove noise without smoothing out the main features of the data, making it effective for data

[3] In our notation, any variable representing a vector is shown in bold italic font; measurements depicting the vector are shown in italic font.

cleaning as well. Given a set of coefficients, an approximation of the original data can be constructed by applying the *inverse* of the DWT used.

The DWT is closely related to the *discrete Fourier transform (DFT)*, a signal processing technique involving sines and cosines. In general, however, the DWT achieves better lossy compression. That is, if the same number of coefficients is retained for a DWT and a DFT of a given data vector, the DWT version will provide a more accurate approximation of the original data. Hence, for an equivalent approximation, the DWT requires less space than the DFT. Unlike the DFT, wavelets are quite localized in space, contributing to the conservation of local detail.

There is only one DFT, yet there are several families of DWTs. Figure 3.4 shows some wavelet families. Popular wavelet transforms include the Haar-2, Daubechies-4, and Daubechies-6. The general procedure for applying a discrete wavelet transform uses a hierarchical *pyramid algorithm* that halves the data at each iteration, resulting in fast computational speed. The method is as follows:

1. The length, L, of the input data vector must be an integer power of 2. This condition can be met by padding the data vector with zeros as necessary ($L \geq n$).
2. Each transform involves applying two functions. The first applies some data smoothing, such as a sum or weighted average. The second performs a weighted difference, which acts to bring out the detailed features of the data.
3. The two functions are applied to pairs of data points in $\boldsymbol{X}$, that is, to all pairs of measurements (x_{2i}, x_{2i+1}). This results in two data sets of length $L/2$. In general, these represent a smoothed or low-frequency version of the input data and the high-frequency content of it, respectively.
4. The two functions are recursively applied to the data sets obtained in the previous loop, until the resulting data sets obtained are of length 2.
5. Selected values from the data sets obtained in the previous iterations are designated the wavelet coefficients of the transformed data.

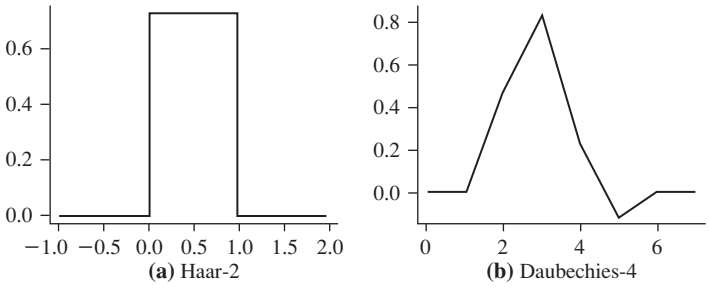


Figure 3.4 Examples of wavelet families. The number next to a wavelet name is the number of *vanishing moments* of the wavelet. This is a set of mathematical relationships that the coefficients must satisfy and is related to the number of coefficients.

Equivalently, a matrix multiplication can be applied to the input data in order to obtain the wavelet coefficients, where the matrix used depends on the given DWT. The matrix must be **orthonormal**, meaning that the columns are unit vectors and are mutually orthogonal, so that the matrix inverse is just its transpose. Although we do not have room to discuss it here, this property allows the reconstruction of the data from the smooth and smooth-difference data sets. By factoring the matrix used into a product of a few sparse matrices, the resulting "fast DWT" algorithm has a complexity of $O(n)$ for an input vector of length n.

Wavelet transforms can be applied to multidimensional data such as a data cube. This is done by first applying the transform to the first dimension, then to the second, and so on. The computational complexity involved is linear with respect to the number of cells in the cube. Wavelet transforms give good results on sparse or skewed data and on data with ordered attributes. Lossy compression by wavelets is reportedly better than JPEG compression, the current commercial standard. Wavelet transforms have many real-world applications, including the compression of fingerprint images, computer vision, analysis of time-series data, and data cleaning.

3.4.3 Principal Components Analysis

In this subsection we provide an intuitive introduction to principal components analysis as a method of dimesionality reduction. A detailed theoretical explanation is beyond the scope of this book. For additional references, please see the bibliographic notes (Section 3.8) at the end of this chapter.

Suppose that the data to be reduced consist of tuples or data vectors described by n attributes or dimensions. **Principal components analysis** (**PCA**; also called the Karhunen-Loeve, or K-L, method) searches for k n-dimensional orthogonal vectors that can best be used to represent the data, where $k \leq n$. The original data are thus projected onto a much smaller space, resulting in dimensionality reduction. Unlike attribute subset selection (Section 3.4.4), which reduces the attribute set size by retaining a subset of the initial set of attributes, PCA "combines" the essence of attributes by creating an alternative, smaller set of variables. The initial data can then be projected onto this smaller set. PCA often reveals relationships that were not previously suspected and thereby allows interpretations that would not ordinarily result.

The basic procedure is as follows:

1. The input data are normalized, so that each attribute falls within the same range. This step helps ensure that attributes with large domains will not dominate attributes with smaller domains.
2. PCA computes k orthonormal vectors that provide a basis for the normalized input data. These are unit vectors that each point in a direction perpendicular to the others. These vectors are referred to as the *principal components*. The input data are a linear combination of the principal components.
3. The principal components are sorted in order of decreasing "significance" or strength. The principal components essentially serve as a new set of axes for the data,

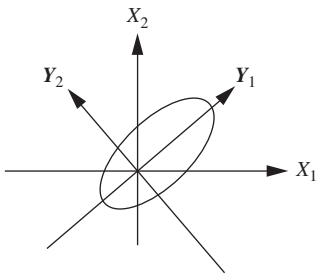


Figure 3.5 Principal components analysis. $\boldsymbol{Y}_1$ and $\boldsymbol{Y}_2$ are the first two principal components for the given data.

providing important information about variance. That is, the sorted axes are such that the first axis shows the most variance among the data, the second axis shows the next highest variance, and so on. For example, Figure 3.5 shows the first two principal components, $\boldsymbol{Y}_1$ and $\boldsymbol{Y}_2$, for the given set of data originally mapped to the axes X_1 and X_2. This information helps identify groups or patterns within the data.

4. Because the components are sorted in decreasing order of "significance," the data size can be reduced by eliminating the weaker components, that is, those with low variance. Using the strongest principal components, it should be possible to reconstruct a good approximation of the original data.

PCA can be applied to ordered and unordered attributes, and can handle sparse data and skewed data. Multidimensional data of more than two dimensions can be handled by reducing the problem to two dimensions. Principal components may be used as inputs to multiple regression and cluster analysis. In comparison with wavelet transforms, PCA tends to be better at handling sparse data, whereas wavelet transforms are more suitable for data of high dimensionality.

3.4.4 Attribute Subset Selection

Data sets for analysis may contain hundreds of attributes, many of which may be irrelevant to the mining task or redundant. For example, if the task is to classify customers based on whether or not they are likely to purchase a popular new CD at *AllElectronics* when notified of a sale, attributes such as the customer's telephone number are likely to be irrelevant, unlike attributes such as *age* or *music_taste*. Although it may be possible for a domain expert to pick out some of the useful attributes, this can be a difficult and time-consuming task, especially when the data's behavior is not well known. (Hence, a reason behind its analysis!) Leaving out relevant attributes or keeping irrelevant attributes may be detrimental, causing confusion for the mining algorithm employed. This can result in discovered patterns of poor quality. In addition, the added volume of irrelevant or redundant attributes can slow down the mining process.

Attribute subset selection[4] reduces the data set size by removing irrelevant or redundant attributes (or dimensions). The goal of attribute subset selection is to find a minimum set of attributes such that the resulting probability distribution of the data classes is as close as possible to the original distribution obtained using all attributes. Mining on a reduced set of attributes has an additional benefit: It reduces the number of attributes appearing in the discovered patterns, helping to make the patterns easier to understand.

"*How can we find a 'good' subset of the original attributes?*" For n attributes, there are 2^n possible subsets. An exhaustive search for the optimal subset of attributes can be prohibitively expensive, especially as n and the number of data classes increase. Therefore, heuristic methods that explore a reduced search space are commonly used for attribute subset selection. These methods are typically **greedy** in that, while searching through attribute space, they always make what looks to be the best choice at the time. Their strategy is to make a locally optimal choice in the hope that this will lead to a globally optimal solution. Such greedy methods are effective in practice and may come close to estimating an optimal solution.

The "best" (and "worst") attributes are typically determined using tests of statistical significance, which assume that the attributes are independent of one another. Many other attribute evaluation measures can be used such as the *information gain* measure used in building decision trees for classification.[5]

Basic heuristic methods of attribute subset selection include the techniques that follow, some of which are illustrated in Figure 3.6.

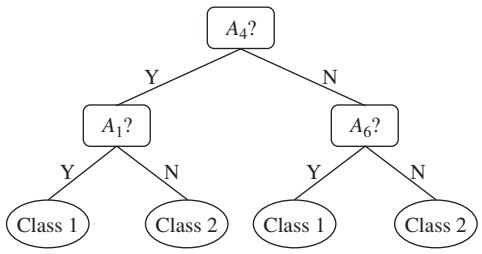
Forward selection	Backward elimination	Decision tree induction
Initial attribute set: $\{A_1, A_2, A_3, A_4, A_5, A_6\}$ Initial reduced set: $\{\}$ $\Rightarrow \{A_1\}$ $\Rightarrow \{A_1, A_4\}$ => Reduced attribute set: $\{A_1, A_4, A_6\}$	Initial attribute set: $\{A_1, A_2, A_3, A_4, A_5, A_6\}$ $\Rightarrow \{A_1, A_3, A_4, A_5, A_6\}$ $\Rightarrow \{A_1, A_4, A_5, A_6\}$ => Reduced attribute set: $\{A_1, A_4, A_6\}$	Initial attribute set: $\{A_1, A_2, A_3, A_4, A_5, A_6\}$  => Reduced attribute set: $\{A_1, A_4, A_6\}$

Figure 3.6 Greedy (heuristic) methods for attribute subset selection.

[4] In machine learning, attribute subset selection is known as *feature subset selection*.

[5] The information gain measure is described in detail in Chapter 8.

1. **Stepwise forward selection**: The procedure starts with an empty set of attributes as the reduced set. The best of the original attributes is determined and added to the reduced set. At each subsequent iteration or step, the best of the remaining original attributes is added to the set.
2. **Stepwise backward elimination**: The procedure starts with the full set of attributes. At each step, it removes the worst attribute remaining in the set.
3. **Combination of forward selection and backward elimination**: The stepwise forward selection and backward elimination methods can be combined so that, at each step, the procedure selects the best attribute and removes the worst from among the remaining attributes.
4. **Decision tree induction**: Decision tree algorithms (e.g., ID3, C4.5, and CART) were originally intended for classification. Decision tree induction constructs a flowchart-like structure where each internal (nonleaf) node denotes a test on an attribute, each branch corresponds to an outcome of the test, and each external (leaf) node denotes a class prediction. At each node, the algorithm chooses the "best" attribute to partition the data into individual classes.

   When decision tree induction is used for attribute subset selection, a tree is constructed from the given data. All attributes that do not appear in the tree are assumed to be irrelevant. The set of attributes appearing in the tree form the reduced subset of attributes.

The stopping criteria for the methods may vary. The procedure may employ a threshold on the measure used to determine when to stop the attribute selection process.

In some cases, we may want to create new attributes based on others. Such **attribute construction**[6] can help improve accuracy and understanding of structure in high-dimensional data. For example, we may wish to add the attribute *area* based on the attributes *height* and *width*. By combining attributes, attribute construction can discover missing information about the relationships between data attributes that can be useful for knowledge discovery.

3.4.5 Regression and Log-Linear Models: Parametric Data Reduction

Regression and log-linear models can be used to approximate the given data. In (simple) **linear regression**, the data are modeled to fit a straight line. For example, a random variable, y (called a *response variable*), can be modeled as a linear function of another random variable, x (called a *predictor variable*), with the equation

$$y = wx + b, \tag{3.7}$$

where the variance of y is assumed to be constant. In the context of data mining, x and y are numeric database attributes. The coefficients, w and b (called *regression coefficients*),

[6] In the machine learning literature, attribute construction is known as *feature construction*.

specify the slope of the line and the y-intercept, respectively. These coefficients can be solved for by the *method of least squares*, which minimizes the error between the actual line separating the data and the estimate of the line. **Multiple linear regression** is an extension of (simple) linear regression, which allows a response variable, y, to be modeled as a linear function of two or more predictor variables.

Log-linear models approximate discrete multidimensional probability distributions. Given a set of tuples in n dimensions (e.g., described by n attributes), we can consider each tuple as a point in an n-dimensional space. Log-linear models can be used to estimate the probability of each point in a multidimensional space for a set of discretized attributes, based on a smaller subset of dimensional combinations. This allows a higher-dimensional data space to be constructed from lower-dimensional spaces. Log-linear models are therefore also useful for dimensionality reduction (since the lower-dimensional points together typically occupy less space than the original data points) and data smoothing (since aggregate estimates in the lower-dimensional space are less subject to sampling variations than the estimates in the higher-dimensional space).

Regression and log-linear models can both be used on sparse data, although their application may be limited. While both methods can handle skewed data, regression does exceptionally well. Regression can be computationally intensive when applied to high-dimensional data, whereas log-linear models show good scalability for up to 10 or so dimensions.

Several software packages exist to solve regression problems. Examples include SAS (*www.sas.com*), SPSS (*www.spss.com*), and S-Plus (*www.insightful.com*). Another useful resource is the book *Numerical Recipes in C*, by Press, Teukolsky, Vetterling, and Flannery [PTVF07], and its associated source code.

3.4.6 Histograms

Histograms use binning to approximate data distributions and are a popular form of data reduction. Histograms were introduced in Section 2.2.3. A **histogram** for an attribute, A, partitions the data distribution of A into disjoint subsets, referred to as *buckets* or *bins*. If each bucket represents only a single attribute–value/frequency pair, the buckets are called *singleton buckets*. Often, buckets instead represent continuous ranges for the given attribute.

Example 3.3 **Histograms.** The following data are a list of *AllElectronics* prices for commonly sold items (rounded to the nearest dollar). The numbers have been sorted: 1, 1, 5, 5, 5, 5, 5, 8, 8, 10, 10, 10, 10, 12, 14, 14, 14, 15, 15, 15, 15, 15, 15, 18, 18, 18, 18, 18, 18, 18, 18, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 25, 25, 25, 25, 25, 28, 28, 30, 30, 30.

Figure 3.7 shows a histogram for the data using singleton buckets. To further reduce the data, it is common to have each bucket denote a continuous value range for the given attribute. In Figure 3.8, each bucket represents a different $10 range for *price*. ■

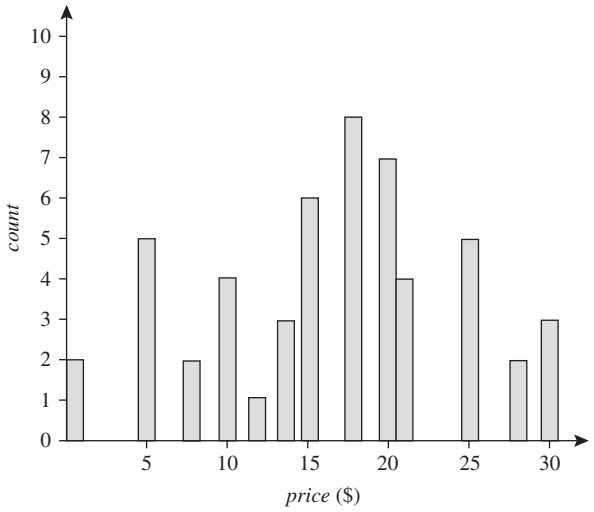


Figure 3.7 A histogram for *price* using singleton buckets—each bucket represents one price–value/frequency pair.

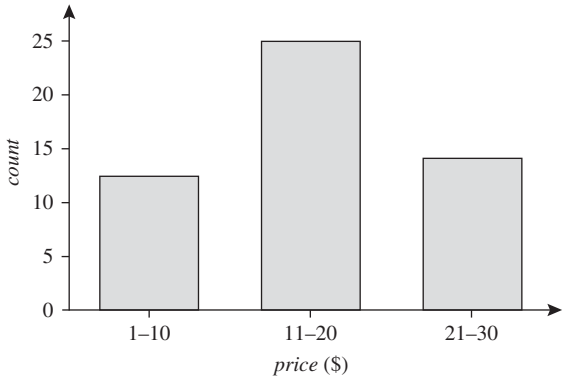


Figure 3.8 An equal-width histogram for *price*, where values are aggregated so that each bucket has a uniform width of $10.

"*How are the buckets determined and the attribute values partitioned?*" There are several partitioning rules, including the following:

- **Equal-width**: In an equal-width histogram, the width of each bucket range is uniform (e.g., the width of $10 for the buckets in Figure 3.8).
- **Equal-frequency** (or equal-depth): In an equal-frequency histogram, the buckets are created so that, roughly, the frequency of each bucket is constant (i.e., each bucket contains roughly the same number of contiguous data samples).

Histograms are highly effective at approximating both sparse and dense data, as well as highly skewed and uniform data. The histograms described before for single attributes can be extended for multiple attributes. *Multidimensional histograms* can capture dependencies between attributes. These histograms have been found effective in approximating data with up to five attributes. More studies are needed regarding the effectiveness of multidimensional histograms for high dimensionalities.

Singleton buckets are useful for storing high-frequency outliers.

3.4.7 Clustering

Clustering techniques consider data tuples as objects. They partition the objects into groups, or *clusters*, so that objects within a cluster are "similar" to one another and "dissimilar" to objects in other clusters. Similarity is commonly defined in terms of how "close" the objects are in space, based on a distance function. The "quality" of a cluster may be represented by its *diameter*, the maximum distance between any two objects in the cluster. **Centroid distance** is an alternative measure of cluster quality and is defined as the average distance of each cluster object from the cluster centroid (denoting the "average object," or average point in space for the cluster). Figure 3.3 showed a 2-D plot of customer data with respect to customer locations in a city. Three data clusters are visible.

In data reduction, the cluster representations of the data are used to replace the actual data. The effectiveness of this technique depends on the data's nature. It is much more effective for data that can be organized into distinct clusters than for smeared data.

There are many measures for defining clusters and cluster quality. Clustering methods are further described in Chapters 10 and 11.

3.4.8 Sampling

Sampling can be used as a data reduction technique because it allows a large data set to be represented by a much smaller random data sample (or subset). Suppose that a large data set, D, contains N tuples. Let's look at the most common ways that we could sample D for data reduction, as illustrated in Figure 3.9.

- **Simple random sample without replacement (SRSWOR) of size** s: This is created by drawing s of the N tuples from D ($s < N$), where the probability of drawing any tuple in D is $1/N$, that is, all tuples are equally likely to be sampled.
- **Simple random sample with replacement (SRSWR) of size** s: This is similar to SRSWOR, except that each time a tuple is drawn from D, it is recorded and then *replaced*. That is, after a tuple is drawn, it is placed back in D so that it may be drawn again.
- **Cluster sample**: If the tuples in D are grouped into M mutually disjoint "clusters," then an SRS of s clusters can be obtained, where $s < M$. For example, tuples in a database are usually retrieved a page at a time, so that each page can be considered

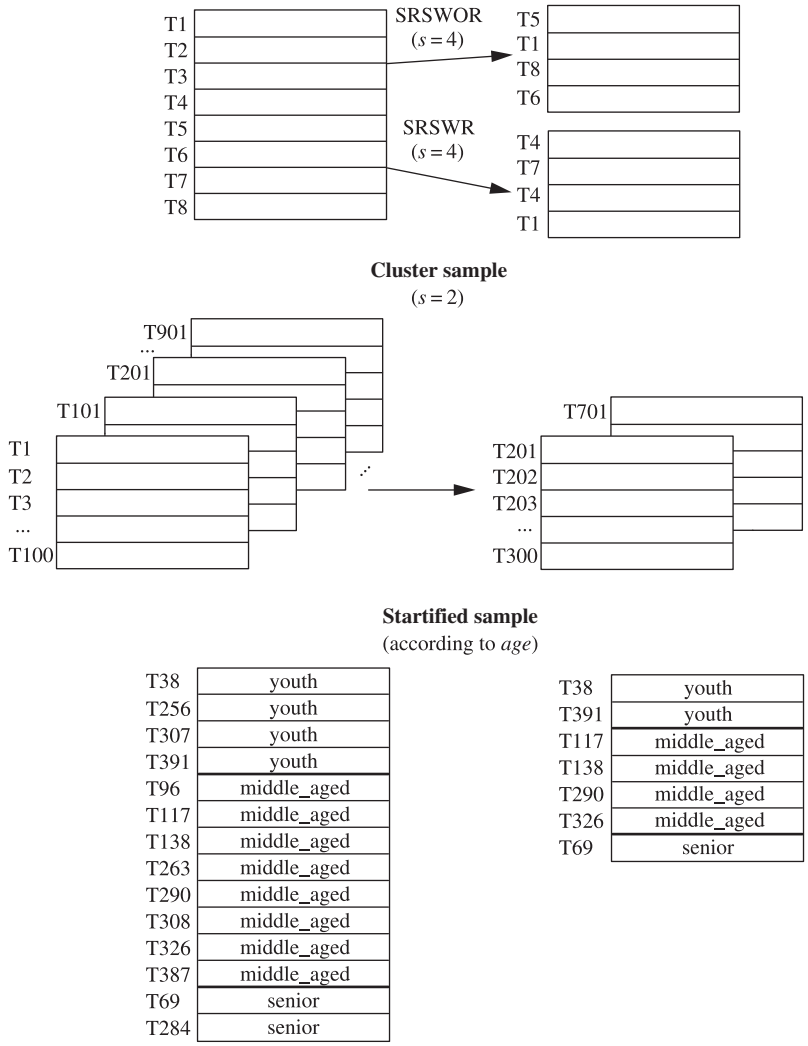


Figure 3.9 Sampling can be used for data reduction.

a cluster. A reduced data representation can be obtained by applying, say, SRSWOR to the pages, resulting in a cluster sample of the tuples. Other clustering criteria conveying rich semantics can also be explored. For example, in a spatial database, we may choose to define clusters geographically based on how closely different areas are located.

- **Stratified sample**: If D is divided into mutually disjoint parts called *strata*, a stratified sample of D is generated by obtaining an SRS at each stratum. This helps ensure a

representative sample, especially when the data are skewed. For example, a stratified sample may be obtained from customer data, where a stratum is created for each customer age group. In this way, the age group having the smallest number of customers will be sure to be represented.

An advantage of sampling for data reduction is that the cost of obtaining a sample *is proportional to the size of the sample*, *s*, as opposed to *N*, the data set size. Hence, sampling complexity is potentially *sublinear* to the size of the data. Other data reduction techniques can require at least one complete pass through *D*. For a fixed sample size, sampling complexity increases only linearly as the number of data dimensions, *n*, increases, whereas techniques using histograms, for example, increase exponentially in *n*.

When applied to data reduction, sampling is most commonly used to estimate the answer to an aggregate query. It is possible (using the central limit theorem) to determine a sufficient sample size for estimating a given function within a specified degree of error. This sample size, *s*, may be extremely small in comparison to *N*. Sampling is a natural choice for the progressive refinement of a reduced data set. Such a set can be further refined by simply increasing the sample size.

3.4.9 Data Cube Aggregation

Imagine that you have collected the data for your analysis. These data consist of the *AllElectronics* sales per quarter, for the years 2008 to 2010. You are, however, interested in the annual sales (total per year), rather than the total per quarter. Thus, the data can be *aggregated* so that the resulting data summarize the total sales per year instead of per quarter. This aggregation is illustrated in Figure 3.10. The resulting data set is smaller in volume, without loss of information necessary for the analysis task.

Data cubes are discussed in detail in Chapter 4 on data warehousing and Chapter 5 on data cube technology. We briefly introduce some concepts here. Data cubes store

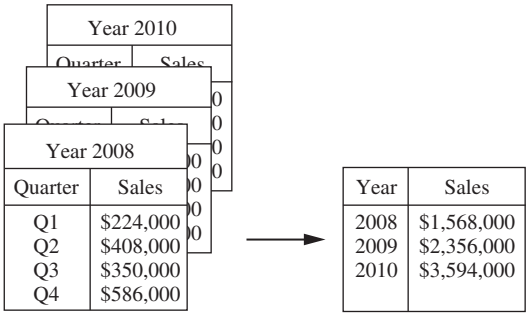


Figure 3.10 Sales data for a given branch of *AllElectronics* for the years 2008 through 2010. On the *left*, the sales are shown per quarter. On the *right*, the data are aggregated to provide the annual sales.

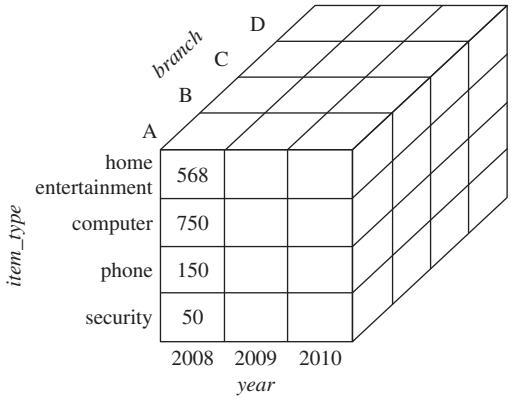


Figure 3.11 A data cube for sales at *AllElectronics*.

multidimensional aggregated information. For example, Figure 3.11 shows a data cube for multidimensional analysis of sales data with respect to annual sales per item type for each *AllElectronics* branch. Each cell holds an aggregate data value, corresponding to the data point in multidimensional space. (For readability, only some cell values are shown.) *Concept hierarchies* may exist for each attribute, allowing the analysis of data at multiple abstraction levels. For example, a hierarchy for *branch* could allow branches to be grouped into regions, based on their address. Data cubes provide fast access to precomputed, summarized data, thereby benefiting online analytical processing as well as data mining.

The cube created at the lowest abstraction level is referred to as the **base cuboid**. The base cuboid should correspond to an individual entity of interest such as *sales* or *customer*. In other words, the lowest level should be usable, or useful for the analysis. A cube at the highest level of abstraction is the **apex cuboid**. For the sales data in Figure 3.11, the apex cuboid would give one total—the total *sales* for all three years, for all item types, and for all branches. Data cubes created for varying levels of abstraction are often referred to as *cuboids*, so that a data cube may instead refer to a *lattice of cuboids*. Each higher abstraction level further reduces the resulting data size. When replying to data mining requests, the *smallest* available cuboid relevant to the given task should be used. This issue is also addressed in Chapter 4.

3.5 Data Transformation and Data Discretization

This section presents methods of data transformation. In this preprocessing step, the data are transformed or consolidated so that the resulting mining process may be more efficient, and the patterns found may be easier to understand. Data discretization, a form of data transformation, is also discussed.

3.5.1 Data Transformation Strategies Overview

In *data transformation*, the data are transformed or consolidated into forms appropriate for mining. Strategies for data transformation include the following:

1. **Smoothing**, which works to remove noise from the data. Techniques include binning, regression, and clustering.
2. **Attribute construction** (or *feature construction*), where new attributes are constructed and added from the given set of attributes to help the mining process.
3. **Aggregation**, where summary or aggregation operations are applied to the data. For example, the daily sales data may be aggregated so as to compute monthly and annual total amounts. This step is typically used in constructing a data cube for data analysis at multiple abstraction levels.
4. **Normalization**, where the attribute data are scaled so as to fall within a smaller range, such as -1.0 to 1.0, or 0.0 to 1.0.
5. **Discretization**, where the raw values of a numeric attribute (e.g., *age*) are replaced by interval labels (e.g., 0–10, 11–20, etc.) or conceptual labels (e.g., *youth, adult, senior*). The labels, in turn, can be recursively organized into higher-level concepts, resulting in a *concept hierarchy* for the numeric attribute. Figure 3.12 shows a concept hierarchy for the attribute *price*. More than one concept hierarchy can be defined for the same attribute to accommodate the needs of various users.
6. **Concept hierarchy generation for nominal data**, where attributes such as *street* can be generalized to higher-level concepts, like *city* or *country*. Many hierarchies for nominal attributes are implicit within the database schema and can be automatically defined at the schema definition level.

Recall that there is much overlap between the major data preprocessing tasks. The first three of these strategies were discussed earlier in this chapter. Smoothing is a form of

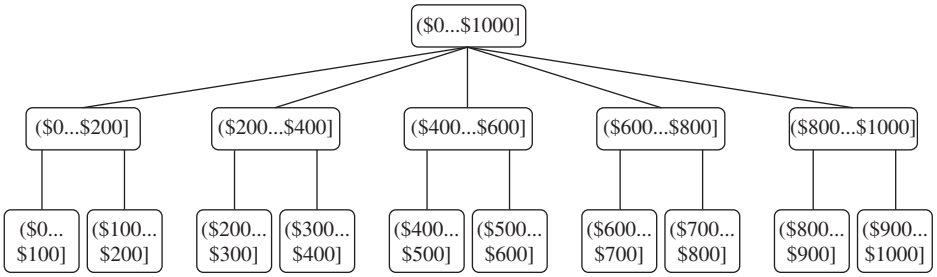


Figure 3.12 A concept hierarchy for the attribute *price*, where an interval ($\$X \ldots \$Y$] denotes the range from $\$X$ (exclusive) to $\$Y$ (inclusive).

data cleaning and was addressed in Section 3.2.2. Section 3.2.3 on the data cleaning process also discussed ETL tools, where users specify transformations to correct data inconsistencies. Attribute construction and aggregation were discussed in Section 3.4 on data reduction. In this section, we therefore concentrate on the latter three strategies.

Discretization techniques can be categorized based on how the discretization is performed, such as whether it uses class information or which direction it proceeds (i.e., top-down vs. bottom-up). If the discretization process uses class information, then we say it is *supervised discretization*. Otherwise, it is *unsupervised*. If the process starts by first finding one or a few points (called *split points* or *cut points*) to split the entire attribute range, and then repeats this recursively on the resulting intervals, it is called *top-down discretization* or *splitting*. This contrasts with *bottom-up discretization* or *merging*, which starts by considering all of the continuous values as potential split-points, removes some by merging neighborhood values to form intervals, and then recursively applies this process to the resulting intervals.

Data discretization and concept hierarchy generation are also forms of data reduction. The raw data are replaced by a smaller number of interval or concept labels. This simplifies the original data and makes the mining more efficient. The resulting patterns mined are typically easier to understand. Concept hierarchies are also useful for mining at multiple abstraction levels.

The rest of this section is organized as follows. First, normalization techniques are presented in Section 3.5.2. We then describe several techniques for data discretization, each of which can be used to generate concept hierarchies for numeric attributes. The techniques include *binning* (Section 3.5.3) and *histogram analysis* (Section 3.5.4), as well as *cluster analysis*, *decision tree analysis*, and *correlation analysis* (Section 3.5.5). Finally, Section 3.5.6 describes the automatic generation of concept hierarchies for nominal data.

3.5.2 Data Transformation by Normalization

The measurement unit used can affect the data analysis. For example, changing measurement units from meters to inches for *height*, or from kilograms to pounds for *weight*, may lead to very different results. In general, expressing an attribute in smaller units will lead to a larger range for that attribute, and thus tend to give such an attribute greater effect or "weight." To help avoid dependence on the choice of measurement units, the data should be *normalized* or *standardized*. This involves transforming the data to fall within a smaller or common range such as $[-1, 1]$ or $[0.0, 1.0]$. (The terms *standardize* and *normalize* are used interchangeably in data preprocessing, although in statistics, the latter term also has other connotations.)

Normalizing the data attempts to give all attributes an equal weight. Normalization is particularly useful for classification algorithms involving neural networks or distance measurements such as nearest-neighbor classification and clustering. If using the neural network backpropagation algorithm for classification mining (Chapter 9), normalizing the input values for each attribute measured in the training tuples will help speed up the learning phase. For distance-based methods, normalization helps prevent

attributes with initially large ranges (e.g., *income*) from outweighing attributes with initially smaller ranges (e.g., binary attributes). It is also useful when given no prior knowledge of the data.

There are many methods for data normalization. We study *min-max normalization, z-score normalization,* and *normalization by decimal scaling.* For our discussion, let A be a numeric attribute with n observed values, $v_1, v_2, \ldots, v_n$.

Min-max normalization performs a linear transformation on the original data. Suppose that min_A and max_A are the minimum and maximum values of an attribute, A. Min-max normalization maps a value, v_i, of A to v'_i in the range $[new\_min_A, new\_max_A]$ by computing

$$v'_i = \frac{v_i - min_A}{max_A - min_A}(new\_max_A - new\_min_A) + new\_min_A. \tag{3.8}$$

Min-max normalization preserves the relationships among the original data values. It will encounter an "out-of-bounds" error if a future input case for normalization falls outside of the original data range for A.

Example 3.4 **Min-max normalization.** Suppose that the minimum and maximum values for the attribute *income* are \$12,000 and \$98,000, respectively. We would like to map *income* to the range $[0.0, 1.0]$. By min-max normalization, a value of \$73,600 for *income* is transformed to $\frac{73{,}600 - 12{,}000}{98{,}000 - 12{,}000}(1.0 - 0) + 0 = 0.716$. ■

In ***z*-score normalization** (or *zero-mean normalization*), the values for an attribute, A, are normalized based on the mean (i.e., average) and standard deviation of A. A value, v_i, of A is normalized to v'_i by computing

$$v'_i = \frac{v_i - \bar{A}}{\sigma_A}, \tag{3.9}$$

where $\bar{A}$ and σ_A are the mean and standard deviation, respectively, of attribute A. The mean and standard deviation were discussed in Section 2.2, where $\bar{A} = \frac{1}{n}(v_1 + v_2 + \cdots + v_n)$ and σ_A is computed as the square root of the variance of A (see Eq. (2.6)). This method of normalization is useful when the actual minimum and maximum of attribute A are unknown, or when there are outliers that dominate the min-max normalization.

Example 3.5 **z-score normalization.** Suppose that the mean and standard deviation of the values for the attribute *income* are \$54,000 and \$16,000, respectively. With z-score normalization, a value of \$73,600 for *income* is transformed to $\frac{73{,}600 - 54{,}000}{16{,}000} = 1.225$. ■

A variation of this z-score normalization replaces the standard deviation of Eq. (3.9) by the *mean absolute deviation* of A. The *mean absolute deviation* of A, denoted s_A, is

$$s_A = \frac{1}{n}(|v_1 - \bar{A}| + |v_2 - \bar{A}| + \cdots + |v_n - \bar{A}|). \tag{3.10}$$

Thus, z-score normalization using the mean absolute deviation is

$$v_i' = \frac{v_i - \bar{A}}{s_A}. \tag{3.11}$$

The mean absolute deviation, s_A, is more robust to outliers than the standard deviation, σ_A. When computing the mean absolute deviation, the deviations from the mean (i.e., $|x_i - \bar{x}|$) are not squared; hence, the effect of outliers is somewhat reduced.

Normalization by decimal scaling normalizes by moving the decimal point of values of attribute A. The number of decimal points moved depends on the maximum absolute value of A. A value, v_i, of A is normalized to v_i' by computing

$$v_i' = \frac{v_i}{10^j}, \tag{3.12}$$

where j is the smallest integer such that $max(|v_i'|) < 1$.

Example 3.6 **Decimal scaling.** Suppose that the recorded values of A range from -986 to 917. The maximum absolute value of A is 986. To normalize by decimal scaling, we therefore divide each value by 1000 (i.e., $j = 3$) so that -986 normalizes to -0.986 and 917 normalizes to 0.917. ■

Note that normalization can change the original data quite a bit, especially when using z-score normalization or decimal scaling. It is also necessary to save the normalization parameters (e.g., the mean and standard deviation if using z-score normalization) so that future data can be normalized in a uniform manner.

3.5.3 Discretization by Binning

Binning is a top-down splitting technique based on a specified number of bins. Section 3.2.2 discussed binning methods for data smoothing. These methods are also used as discretization methods for data reduction and concept hierarchy generation. For example, attribute values can be discretized by applying equal-width or equal-frequency binning, and then replacing each bin value by the bin mean or median, as in *smoothing by bin means* or *smoothing by bin medians*, respectively. These techniques can be applied recursively to the resulting partitions to generate concept hierarchies.

Binning does not use class information and is therefore an unsupervised discretization technique. It is sensitive to the user-specified number of bins, as well as the presence of outliers.

3.5.4 Discretization by Histogram Analysis

Like binning, histogram analysis is an unsupervised discretization technique because it does not use class information. Histograms were introduced in Section 2.2.3. A histogram partitions the values of an attribute, A, into disjoint ranges called *buckets* or *bins*.

Various partitioning rules can be used to define histograms (Section 3.4.6). In an *equal-width* histogram, for example, the values are partitioned into equal-size partitions or ranges (e.g., earlier in Figure 3.8 for *price*, where each bucket has a width of $10). With an *equal-frequency* histogram, the values are partitioned so that, ideally, each partition contains the same number of data tuples. The histogram analysis algorithm can be applied recursively to each partition in order to automatically generate a multilevel concept hierarchy, with the procedure terminating once a prespecified number of concept levels has been reached. A *minimum interval size* can also be used per level to control the recursive procedure. This specifies the minimum width of a partition, or the minimum number of values for each partition at each level. Histograms can also be partitioned based on cluster analysis of the data distribution, as described next.

3.5.5 Discretization by Cluster, Decision Tree, and Correlation Analyses

Clustering, decision tree analysis, and correlation analysis can be used for data discretization. We briefly study each of these approaches.

Cluster analysis is a popular data discretization method. A clustering algorithm can be applied to discretize a numeric attribute, A, by partitioning the values of A into clusters or groups. Clustering takes the distribution of A into consideration, as well as the closeness of data points, and therefore is able to produce high-quality discretization results.

Clustering can be used to generate a concept hierarchy for A by following either a top-down splitting strategy or a bottom-up merging strategy, where each cluster forms a node of the concept hierarchy. In the former, each initial cluster or partition may be further decomposed into several subclusters, forming a lower level of the hierarchy. In the latter, clusters are formed by repeatedly grouping neighboring clusters in order to form higher-level concepts. Clustering methods for data mining are studied in Chapters 10 and 11.

Techniques to generate decision trees for classification (Chapter 8) can be applied to discretization. Such techniques employ a top-down splitting approach. Unlike the other methods mentioned so far, decision tree approaches to discretization are supervised, that is, they make use of class label information. For example, we may have a data set of patient symptoms (the attributes) where each patient has an associated *diagnosis* class label. Class distribution information is used in the calculation and determination of split-points (data values for partitioning an attribute range). Intuitively, the main idea is to select split-points so that a given resulting partition contains as many tuples of the same class as possible. *Entropy* is the most commonly used measure for this purpose. To discretize a numeric attribute, A, the method selects the value of A that has the minimum entropy as a split-point, and recursively partitions the resulting intervals to arrive at a hierarchical discretization. Such discretization forms a concept hierarchy for A.

Because decision tree–based discretization uses class information, it is more likely that the interval boundaries (split-points) are defined to occur in places that may help improve classification accuracy. Decision trees and the entropy measure are described in greater detail in Section 8.2.2.

Measures of correlation can be used for discretization. *ChiMerge* is a χ^2-based discretization method. The discretization methods that we have studied up to this point have all employed a top-down, splitting strategy. This contrasts with ChiMerge, which employs a bottom-up approach by finding the best neighboring intervals and then merging them to form larger intervals, recursively. As with decision tree analysis, ChiMerge is supervised in that it uses class information. The basic notion is that for accurate discretization, the relative class frequencies should be fairly consistent within an interval. Therefore, if two adjacent intervals have a very similar distribution of classes, then the intervals can be merged. Otherwise, they should remain separate.

ChiMerge proceeds as follows. Initially, each distinct value of a numeric attribute A is considered to be one interval. χ^2 tests are performed for every pair of adjacent intervals. Adjacent intervals with the least χ^2 values are merged together, because low χ^2 values for a pair indicate similar class distributions. This merging process proceeds recursively until a predefined stopping criterion is met.

3.5.6 Concept Hierarchy Generation for Nominal Data

We now look at data transformation for nominal data. In particular, we study concept hierarchy generation for nominal attributes. Nominal attributes have a finite (but possibly large) number of distinct values, with no ordering among the values. Examples include *geographic_location*, *job_category*, and *item_type*.

Manual definition of concept hierarchies can be a tedious and time-consuming task for a user or a domain expert. Fortunately, many hierarchies are implicit within the database schema and can be automatically defined at the schema definition level. The concept hierarchies can be used to transform the data into multiple levels of granularity. For example, data mining patterns regarding sales may be found relating to specific regions or countries, in addition to individual branch locations.

We study four methods for the generation of concept hierarchies for nominal data, as follows.

1. **Specification of a partial ordering of attributes explicitly at the schema level by users or experts:** Concept hierarchies for nominal attributes or dimensions typically involve a group of attributes. A user or expert can easily define a concept hierarchy by specifying a partial or total ordering of the attributes at the schema level. For example, suppose that a relational database contains the following group of attributes: *street, city, province_or_state*, and *country*. Similarly, a data warehouse *location* dimension may contain the same attributes. A hierarchy can be defined by specifying the total ordering among these attributes at the schema level such as *street* $<$ *city* $<$ *province_or_state* $<$ *country*.
2. **Specification of a portion of a hierarchy by explicit data grouping:** This is essentially the manual definition of a portion of a concept hierarchy. In a large database, it is unrealistic to define an entire concept hierarchy by explicit value enumeration. On the contrary, we can easily specify explicit groupings for a small portion of intermediate-level data. For example, after specifying that *province* and *country*

form a hierarchy at the schema level, a user could define some intermediate levels manually, such as "{*Alberta, Saskatchewan, Manitoba*} $\subset$ *prairies_Canada*" and "{*British Columbia, prairies_Canada*} $\subset$ *Western_Canada*."

3. **Specification of a *set of attributes*, but not of their partial ordering:** A user may specify a set of attributes forming a concept hierarchy, but omit to explicitly state their partial ordering. The system can then try to automatically generate the attribute ordering so as to construct a meaningful concept hierarchy.

   "*Without knowledge of data semantics, how can a hierarchical ordering for an arbitrary set of nominal attributes be found?*" Consider the observation that since higher-level concepts generally cover several subordinate lower-level concepts, an attribute defining a high concept level (e.g., *country*) will usually contain a smaller number of distinct values than an attribute defining a lower concept level (e.g., *street*). Based on this observation, a concept hierarchy can be automatically generated based on the number of distinct values per attribute in the given attribute set. The attribute with the most distinct values is placed at the lowest hierarchy level. The lower the number of distinct values an attribute has, the higher it is in the generated concept hierarchy. This heuristic rule works well in many cases. Some local-level swapping or adjustments may be applied by users or experts, when necessary, after examination of the generated hierarchy.

Let's examine an example of this third method.

Example 3.7 **Concept hierarchy generation based on the number of distinct values per attribute.** Suppose a user selects a set of location-oriented attributes—*street, country, province_or_state*, and *city*—from the *AllElectronics* database, but does not specify the hierarchical ordering among the attributes.

A concept hierarchy for *location* can be generated automatically, as illustrated in Figure 3.13. First, sort the attributes in ascending order based on the number of distinct values in each attribute. This results in the following (where the number of distinct values per attribute is shown in parentheses): *country* (15), *province_or_state* (365), *city* (3567), and *street* (674,339). Second, generate the hierarchy from the top down according to the sorted order, with the first attribute at the top level and the last attribute at the bottom level. Finally, the user can examine the generated hierarchy, and when necessary, modify it to reflect desired semantic relationships among the attributes. In this example, it is obvious that there is no need to modify the generated hierarchy. ■

Note that this heuristic rule is not foolproof. For example, a time dimension in a database may contain 20 distinct years, 12 distinct months, and 7 distinct days of the week. However, this does not suggest that the time hierarchy should be "*year* < *month* < *days_of_the_week*," with *days_of_the_week* at the top of the hierarchy.

4. **Specification of only a partial set of attributes:** Sometimes a user can be careless when defining a hierarchy, or have only a vague idea about what should be included in a hierarchy. Consequently, the user may have included only a small subset of the

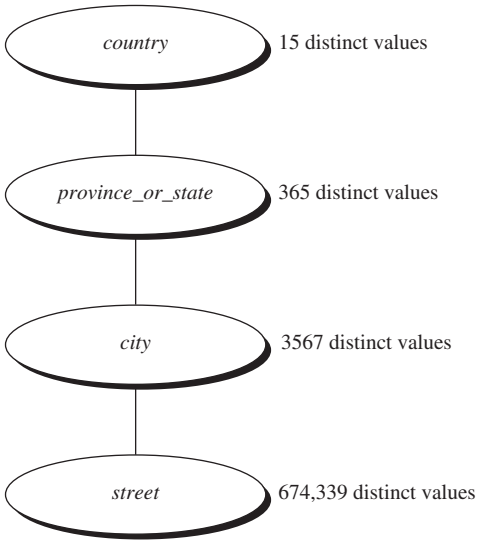


Figure 3.13 Automatic generation of a schema concept hierarchy based on the number of distinct attribute values.

relevant attributes in the hierarchy specification. For example, instead of including all of the hierarchically relevant attributes for *location*, the user may have specified only *street* and *city*. To handle such partially specified hierarchies, it is important to embed data semantics in the database schema so that attributes with tight semantic connections can be pinned together. In this way, the specification of one attribute may trigger a whole group of semantically tightly linked attributes to be "dragged in" to form a complete hierarchy. Users, however, should have the option to override this feature, as necessary.

Example 3.8 **Concept hierarchy generation using prespecified semantic connections.** Suppose that a data mining expert (serving as an administrator) has pinned together the five attributes *number, street, city, province_or_state*, and *country*, because they are closely linked semantically regarding the notion of *location*. If a user were to specify only the attribute *city* for a hierarchy defining *location*, the system can automatically drag in all five semantically related attributes to form a hierarchy. The user may choose to drop any of these attributes (e.g., *number* and *street*) from the hierarchy, keeping *city* as the lowest conceptual level. ■

In summary, information at the schema level and on attribute–value counts can be used to generate concept hierarchies for nominal data. Transforming nominal data with the use of concept hierarchies allows higher-level knowledge patterns to be found. It allows mining at multiple levels of abstraction, which is a common requirement for data mining applications.

3.6 Summary

- **Data quality** is defined in terms of *accuracy, completeness, consistency, timeliness, believability*, and *interpretabilty*. These qualities are assessed based on the intended use of the data.
- **Data cleaning** routines attempt to fill in missing values, smooth out noise while identifying outliers, and correct inconsistencies in the data. Data cleaning is usually performed as an iterative two-step process consisting of discrepancy detection and data transformation.
- **Data integration** combines data from multiple sources to form a coherent data store. The resolution of semantic heterogeneity, metadata, correlation analysis, tuple duplication detection, and data conflict detection contribute to smooth data integration.
- **Data reduction** techniques obtain a reduced representation of the data while minimizing the loss of information content. These include methods of *dimensionality reduction*, *numerosity reduction*, and *data compression*. **Dimensionality reduction** reduces the number of random variables or attributes under consideration. Methods include *wavelet transforms, principal components analysis, attribute subset selection*, and *attribute creation*. **Numerosity reduction** methods use parametric or nonparatmetric models to obtain smaller representations of the original data. Parametric models store only the model parameters instead of the actual data. Examples include regression and log-linear models. Nonparamteric methods include histograms, clustering, sampling, and data cube aggregation. **Data compression** methods apply transformations to obtain a reduced or "compressed" representation of the original data. The data reduction is *lossless* if the original data can be reconstructed from the compressed data without any loss of information; otherwise, it is *lossy*.
- **Data transformation** routines convert the data into appropriate forms for mining. For example, in **normalization**, attribute data are scaled so as to fall within a small range such as 0.0 to 1.0. Other examples are **data discretization** and **concept hierarchy generation**.
- **Data discretization** transforms numeric data by mapping values to interval or concept labels. Such methods can be used to automatically generate *concept hierarchies* for the data, which allows for mining at multiple levels of granularity. Discretization techniques include binning, histogram analysis, cluster analysis, decision tree analysis, and correlation analysis. For nominal data, **concept hierarchies** may be generated based on schema definitions as well as the number of distinct values per attribute.
- Although numerous methods of data preprocessing have been developed, data preprocessing remains an active area of research, due to the huge amount of inconsistent or dirty data and the complexity of the problem.

3.7 Exercises

3.1 *Data quality* can be assessed in terms of several issues, including accuracy, completeness, and consistency. For each of the above three issues, discuss how data quality assessment can depend on the *intended use* of the data, giving examples. Propose two other dimensions of data quality.

3.2 In real-world data, tuples with *missing values* for some attributes are a common occurrence. Describe various methods for handling this problem.

3.3 Exercise 2.2 gave the following data (in increasing order) for the attribute *age*: 13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25, 25, 25, 30, 33, 33, 35, 35, 35, 35, 36, 40, 45, 46, 52, 70.

(a) Use *smoothing by bin means* to smooth these data, using a bin depth of 3. Illustrate your steps. Comment on the effect of this technique for the given data.

(b) How might you determine *outliers* in the data?

(c) What other methods are there for *data smoothing*?

3.4 Discuss issues to consider during *data integration*.

3.5 What are the value ranges of the following *normalization methods*?

(a) min-max normalization

(b) z-score normalization

(c) z-score normalization using the mean absolute deviation instead of standard deviation

(d) normalization by decimal scaling

3.6 Use these methods to *normalize* the following group of data:

$$200, 300, 400, 600, 1000$$

(a) min-max normalization by setting $min = 0$ and $max = 1$

(b) z-score normalization

(c) z-score normalization using the mean absolute deviation instead of standard deviation

(d) normalization by decimal scaling

3.7 Using the data for *age* given in Exercise 3.3, answer the following:

(a) Use min-max normalization to transform the value 35 for *age* onto the range $[0.0, 1.0]$.

(b) Use z-score normalization to transform the value 35 for *age*, where the standard deviation of *age* is 12.94 years.

(c) Use normalization by decimal scaling to transform the value 35 for *age*.

(d) Comment on which method you would prefer to use for the given data, giving reasons as to why.

3.8 Using the data for *age* and *body fat* given in Exercise 2.4, answer the following:

(a) Normalize the two attributes based on *z-score normalization*.

(b) Calculate the *correlation coefficient* (Pearson's product moment coefficient). Are these two attributes positively or negatively correlated? Compute their covariance.

3.9 Suppose a group of 12 *sales price* records has been sorted as follows:

$$5, 10, 11, 13, 15, 35, 50, 55, 72, 92, 204, 215.$$

Partition them into three bins by each of the following methods:

(a) equal-frequency (equal-depth) partitioning

(b) equal-width partitioning

(c) clustering

3.10 Use a flowchart to summarize the following procedures for *attribute subset selection*:

(a) stepwise forward selection

(b) stepwise backward elimination

(c) a combination of forward selection and backward elimination

3.11 Using the data for *age* given in Exercise 3.3,

(a) Plot an equal-width histogram of width 10.

(b) Sketch examples of each of the following sampling techniques: SRSWOR, SRSWR, cluster sampling, and stratified sampling. Use samples of size 5 and the strata "youth," "middle-aged," and "senior."

3.12 ChiMerge [Ker92] is a supervised, bottom-up (i.e., merge-based) *data discretization* method. It relies on χ^2 analysis: Adjacent intervals with the least χ^2 values are merged together until the chosen stopping criterion satisfies.

(a) Briefly describe how ChiMerge works.

(b) Take the IRIS data set, obtained from the University of California–Irvine Machine Learning Data Repository (*www.ics.uci.edu/~mlearn/MLRepository.html*), as a data set to be discretized. Perform data discretization for each of the four numeric attributes using the ChiMerge method. (Let the stopping criteria be: *max-interval* $= 6$). You need to write a small program to do this to avoid clumsy numerical computation. Submit your simple analysis and your test results: split-points, final intervals, and the documented source program.

3.13 Propose an algorithm, in pseudocode or in your favorite programming language, for the following:

(a) The automatic generation of a concept hierarchy for nominal data based on the number of distinct values of attributes in the given schema.

(b) The automatic generation of a concept hierarchy for numeric data based on the *equal-width* partitioning rule.

(c) The automatic generation of a concept hierarchy for numeric data based on the *equal-frequency* partitioning rule.

3.14 Robust data loading poses a challenge in database systems because the input data are often dirty. In many cases, an input record may miss multiple values; some records could be *contaminated*, with some data values out of range or of a different data type than expected. Work out an automated *data cleaning and loading* algorithm so that the erroneous data will be marked and contaminated data will not be mistakenly inserted into the database during data loading.

3.8 Bibliographic Notes

Data preprocessing is discussed in a number of textbooks, including English [Eng99], Pyle [Pyl99], Loshin [Los01], Redman [Red01], and Dasu and Johnson [DJ03]. More specific references to individual preprocessing techniques are given later.

For discussion regarding data quality, see Redman [Red92]; Wang, Storey, and Firth [WSF95]; Wand and Wang [WW96]; Ballou and Tayi [BT99]; and Olson [Ols03]. Potter's Wheel (*control.cx.berkely.edu/abc*), the interactive data cleaning tool described in Section 3.2.3, is presented in Raman and Hellerstein [RH01]. An example of the development of declarative languages for the specification of data transformation operators is given in Galhardas et al. [GFS+01]. The handling of missing attribute values is discussed in Friedman [Fri77]; Breiman, Friedman, Olshen, and Stone [BFOS84]; and Quinlan [Qui89]. Hua and Pei [HP07] presented a heuristic approach to cleaning *disguised missing data*, where such data are captured when users falsely select default values on forms (e.g., "January 1" for *birthdate*) when they do not want to disclose personal information.

A method for the detection of outlier or "garbage" patterns in a handwritten character database is given in Guyon, Matic, and Vapnik [GMV96]. Binning and data normalization are treated in many texts, including Kennedy et al. [KLV+98], Weiss and Indurkhya [WI98], and Pyle [Pyl99]. Systems that include attribute (or feature) construction include BACON by Langley, Simon, Bradshaw, and Zytkow [LSBZ87]; Stagger by Schlimmer [Sch86]; FRINGE by Pagallo [Pag89]; and AQ17-DCI by Bloedorn and Michalski [BM98]. Attribute construction is also described in Liu and Motoda [LM98a, LM98b]. Dasu et al. built a BELLMAN system and proposed a set of interesting methods for building a data quality browser by mining database structures [DJMS02].

A good survey of data reduction techniques can be found in Barbará et al. [BDF+97]. For algorithms on data cubes and their precomputation, see Sarawagi and Stonebraker [SS94]; Agarwal et al. [AAD+96]; Harinarayan, Rajaraman, and Ullman [HRU96]; Ross and Srivastava [RS97]; and Zhao, Deshpande, and Naughton [ZDN97]. Attribute subset selection (or *feature subset selection*) is described in many texts such as Neter, Kutner, Nachtsheim, and Wasserman [NKNW96]; Dash and Liu [DL97]; and Liu and Motoda [LM98a, LM98b]. A combination forward selection and backward elimination method

was proposed in Siedlecki and Sklansky [SS88]. A wrapper approach to attribute selection is described in Kohavi and John [KJ97]. Unsupervised attribute subset selection is described in Dash, Liu, and Yao [DLY97].

For a description of wavelets for dimensionality reduction, see Press, Teukolosky, Vetterling, and Flannery [PTVF07]. A general account of wavelets can be found in Hubbard [Hub96]. For a list of wavelet software packages, see Bruce, Donoho, and Gao [BDG96]. Daubechies transforms are described in Daubechies [Dau92]. The book by Press et al. [PTVF07] includes an introduction to singular value decomposition for principal components analysis. Routines for PCA are included in most statistical software packages such as SAS (*www.sas.com/SASHome.html*).

An introduction to regression and log-linear models can be found in several textbooks such as James [Jam85]; Dobson [Dob90]; Johnson and Wichern [JW92]; Devore [Dev95]; and Neter, Kutner, Nachtsheim, and Wasserman [NKNW96]. For log-linear models (known as *multiplicative models* in the computer science literature), see Pearl [Pea88]. For a general introduction to histograms, see Barbará et al. [BDF+97] and Devore and Peck [DP97]. For extensions of single-attribute histograms to multiple attributes, see Muralikrishna and DeWitt [MD88] and Poosala and Ioannidis [PI97]. Several references to clustering algorithms are given in Chapters 10 and 11 of this book, which are devoted to the topic.

A survey of multidimensional indexing structures is given in Gaede and Günther [GG98]. The use of multidimensional index trees for data aggregation is discussed in Aoki [Aok98]. Index trees include R-trees (Guttman [Gut84]), quad-trees (Finkel and Bentley [FB74]), and their variations. For discussion on sampling and data mining, see Kivinen and Mannila [KM94] and John and Langley [JL96].

There are many methods for assessing attribute relevance. Each has its own bias. The information gain measure is biased toward attributes with many values. Many alternatives have been proposed, such as gain ratio (Quinlan [Qui93]), which considers the probability of each attribute value. Other relevance measures include the Gini index (Breiman, Friedman, Olshen, and Stone [BFOS84]), the χ^2 contingency table statistic, and the uncertainty coefficient (Johnson and Wichern [JW92]). For a comparison of attribute selection measures for decision tree induction, see Buntine and Niblett [BN92]. For additional methods, see Liu and Motoda [LM98a], Dash and Liu [DL97], and Almuallim and Dietterich [AD91].

Liu et al. [LHTD02] performed a comprehensive survey of data discretization methods. Entropy-based discretization with the C4.5 algorithm is described in Quinlan [Qui93]. In Catlett [Cat91], the D-2 system binarizes a numeric feature recursively. ChiMerge by Kerber [Ker92] and Chi2 by Liu and Setiono [LS95] are methods for the automatic discretization of numeric attributes that both employ the χ^2 statistic. Fayyad and Irani [FI93] apply the minimum description length principle to determine the number of intervals for numeric discretization. Concept hierarchies and their automatic generation from categorical data are described in Han and Fu [HF94].

4 Data Warehousing and Online Analytical Processing

Data warehouses generalize and consolidate data in multidimensional space. The construction of data warehouses involves data cleaning, data integration, and data transformation, and can be viewed as an important preprocessing step for data mining. Moreover, data warehouses provide *online analytical processing (OLAP)* tools for the interactive analysis of multidimensional data of varied granularities, which facilitates effective data generalization and data mining. Many other data mining functions, such as association, classification, prediction, and clustering, can be integrated with OLAP operations to enhance interactive mining of knowledge at multiple levels of abstraction. Hence, the data warehouse has become an increasingly important platform for data analysis and OLAP and will provide an effective platform for data mining. Therefore, data warehousing and OLAP form an essential step in the knowledge discovery process. This chapter presents an overview of data warehouse and OLAP technology. This overview is essential for understanding the overall data mining and knowledge discovery process.

In this chapter, we study a well-accepted definition of the data warehouse and see why more and more organizations are building data warehouses for the analysis of their data (Section 4.1). In particular, we study the *data cube*, a multidimensional data model for data warehouses and OLAP, as well as OLAP operations such as roll-up, drill-down, slicing, and dicing (Section 4.2). We also look at data warehouse design and usage (Section 4.3). In addition, we discuss *multidimensional data mining*, a powerful paradigm that integrates data warehouse and OLAP technology with that of data mining. An overview of data warehouse implementation examines general strategies for efficient data cube computation, OLAP data indexing, and OLAP query processing (Section 4.4). Finally, we study data generalization by attribute-oriented induction (Section 4.5). This method uses concept hierarchies to generalize data to multiple levels of abstraction.

4.1 Data Warehouse: Basic Concepts

This section gives an introduction to data warehouses. We begin with a definition of the data warehouse (Section 4.1.1). We outline the differences between operational database

systems and data warehouses (Section 4.1.2), then explain the need for using data warehouses for data analysis, rather than performing the analysis directly on traditional databases (Section 4.1.3). This is followed by a presentation of data warehouse architecture (Section 4.1.4). Next, we study three data warehouse models—an enterprise model, a data mart, and a virtual warehouse (Section 4.1.5). Section 4.1.6 describes back-end utilities for data warehousing, such as extraction, transformation, and loading. Finally, Section 4.1.7 presents the metadata repository, which stores data about data.

4.1.1 What Is a Data Warehouse?

Data warehousing provides architectures and tools for business executives to systematically organize, understand, and use their data to make strategic decisions. Data warehouse systems are valuable tools in today's competitive, fast-evolving world. In the last several years, many firms have spent millions of dollars in building enterprise-wide data warehouses. Many people feel that with competition mounting in every industry, data warehousing is the latest must-have marketing weapon—a way to retain customers by learning more about their needs.

"*Then, what exactly is a data warehouse?*" Data warehouses have been defined in many ways, making it difficult to formulate a rigorous definition. Loosely speaking, a data warehouse refers to a data repository that is maintained separately from an organization's operational databases. Data warehouse systems allow for integration of a variety of application systems. They support information processing by providing a solid platform of consolidated historic data for analysis.

According to William H. Inmon, a leading architect in the construction of data warehouse systems, "A data warehouse is a subject-oriented, integrated, time-variant, and nonvolatile collection of data in support of management's decision making process" [Inm96]. This short but comprehensive definition presents the major features of a data warehouse. The four keywords—*subject-oriented, integrated, time-variant*, and *nonvolatile*—distinguish data warehouses from other data repository systems, such as relational database systems, transaction processing systems, and file systems.

Let's take a closer look at each of these key features.

- **Subject-oriented**: A data warehouse is organized around major subjects such as customer, supplier, product, and sales. Rather than concentrating on the day-to-day operations and transaction processing of an organization, a data warehouse focuses on the modeling and analysis of data for decision makers. Hence, data warehouses typically provide a simple and concise view of particular subject issues by excluding data that are not useful in the decision support process.
- **Integrated**: A data warehouse is usually constructed by integrating multiple heterogeneous sources, such as relational databases, flat files, and online transaction records. Data cleaning and data integration techniques are applied to ensure consistency in naming conventions, encoding structures, attribute measures, and so on.

- **Time-variant**: Data are stored to provide information from an historic perspective (e.g., the past 5–10 years). Every key structure in the data warehouse contains, either implicitly or explicitly, a time element.
- **Nonvolatile**: A data warehouse is always a physically separate store of data transformed from the application data found in the operational environment. Due to this separation, a data warehouse does not require transaction processing, recovery, and concurrency control mechanisms. It usually requires only two operations in data accessing: *initial loading of data* and *access of data*.

In sum, a data warehouse is a semantically consistent data store that serves as a physical implementation of a decision support data model. It stores the information an enterprise needs to make strategic decisions. A data warehouse is also often viewed as an architecture, constructed by integrating data from multiple heterogeneous sources to support structured and/or ad hoc queries, analytical reporting, and decision making.

Based on this information, we view **data warehousing** as the process of constructing and using data warehouses. The construction of a data warehouse requires data cleaning, data integration, and data consolidation. The utilization of a data warehouse often necessitates a collection of *decision support* technologies. This allows "knowledge workers" (e.g., managers, analysts, and executives) to use the warehouse to quickly and conveniently obtain an overview of the data, and to make sound decisions based on information in the warehouse. Some authors use the term *data warehousing* to refer only to the process of data warehouse *construction*, while the term *warehouse DBMS* is used to refer to the *management and utilization* of data warehouses. We will not make this distinction here.

"*How are organizations using the information from data warehouses?*" Many organizations use this information to support business decision-making activities, including (1) increasing customer focus, which includes the analysis of customer buying patterns (such as buying preference, buying time, budget cycles, and appetites for spending); (2) repositioning products and managing product portfolios by comparing the performance of sales by quarter, by year, and by geographic regions in order to fine-tune production strategies; (3) analyzing operations and looking for sources of profit; and (4) managing customer relationships, making environmental corrections, and managing the cost of corporate assets.

Data warehousing is also very useful from the point of view of *heterogeneous database integration*. Organizations typically collect diverse kinds of data and maintain large databases from multiple, heterogeneous, autonomous, and distributed information sources. It is highly desirable, yet challenging, to integrate such data and provide easy and efficient access to it. Much effort has been spent in the database industry and research community toward achieving this goal.

The traditional database approach to heterogeneous database integration is to build **wrappers** and **integrators** (or **mediators**) on top of multiple, heterogeneous databases. When a query is posed to a client site, a metadata dictionary is used to translate the query into queries appropriate for the individual heterogeneous sites involved. These

queries are then mapped and sent to local query processors. The results returned from the different sites are integrated into a global answer set. This **query-driven approach** requires complex information filtering and integration processes, and competes with local sites for processing resources. It is inefficient and potentially expensive for frequent queries, especially queries requiring aggregations.

Data warehousing provides an interesting alternative to this traditional approach. Rather than using a query-driven approach, data warehousing employs an **update-driven** approach in which information from multiple, heterogeneous sources is integrated in advance and stored in a warehouse for direct querying and analysis. Unlike online transaction processing databases, data warehouses do not contain the most current information. However, a data warehouse brings high performance to the integrated heterogeneous database system because data are copied, preprocessed, integrated, annotated, summarized, and restructured into one semantic data store. Furthermore, query processing in data warehouses does not interfere with the processing at local sources. Moreover, data warehouses can store and integrate historic information and support complex multidimensional queries. As a result, data warehousing has become popular in industry.

4.1.2 Differences between Operational Database Systems and Data Warehouses

Because most people are familiar with commercial relational database systems, it is easy to understand what a data warehouse is by comparing these two kinds of systems.

The major task of online operational database systems is to perform online transaction and query processing. These systems are called **online transaction processing (OLTP)** systems. They cover most of the day-to-day operations of an organization such as purchasing, inventory, manufacturing, banking, payroll, registration, and accounting. Data warehouse systems, on the other hand, serve users or knowledge workers in the role of data analysis and decision making. Such systems can organize and present data in various formats in order to accommodate the diverse needs of different users. These systems are known as **online analytical processing (OLAP)** systems.

The major distinguishing features of OLTP and OLAP are summarized as follows:

- **Users and system orientation**: An OLTP system is *customer-oriented* and is used for transaction and query processing by clerks, clients, and information technology professionals. An OLAP system is *market-oriented* and is used for data analysis by knowledge workers, including managers, executives, and analysts.
- **Data contents**: An OLTP system manages current data that, typically, are too detailed to be easily used for decision making. An OLAP system manages large amounts of historic data, provides facilities for summarization and aggregation, and stores and manages information at different levels of granularity. These features make the data easier to use for informed decision making.

- **Database design**: An OLTP system usually adopts an entity-relationship (ER) data model and an application-oriented database design. An OLAP system typically adopts either a *star* or a *snowflake* model (see Section 4.2.2) and a subject-oriented database design.
- **View**: An OLTP system focuses mainly on the current data within an enterprise or department, without referring to historic data or data in different organizations. In contrast, an OLAP system often spans multiple versions of a database schema, due to the evolutionary process of an organization. OLAP systems also deal with information that originates from different organizations, integrating information from many data stores. Because of their huge volume, OLAP data are stored on multiple storage media.
- **Access patterns**: The access patterns of an OLTP system consist mainly of short, atomic transactions. Such a system requires concurrency control and recovery mechanisms. However, accesses to OLAP systems are mostly read-only operations (because most data warehouses store historic rather than up-to-date information), although many could be complex queries.

Other features that distinguish between OLTP and OLAP systems include database size, frequency of operations, and performance metrics. These are summarized in Table 4.1.

4.1.3 But, Why Have a Separate Data Warehouse?

Because operational databases store huge amounts of data, you may wonder, "*Why not perform online analytical processing directly on such databases instead of spending additional time and resources to construct a separate data warehouse?*" A major reason for such a separation is to help promote the *high performance of both systems*. An operational database is designed and tuned from known tasks and workloads like indexing and hashing using primary keys, searching for particular records, and optimizing "canned" queries. On the other hand, data warehouse queries are often complex. They involve the computation of large data groups at summarized levels, and may require the use of special data organization, access, and implementation methods based on multidimensional views. Processing OLAP queries in operational databases would substantially degrade the performance of operational tasks.

Moreover, an operational database supports the concurrent processing of multiple transactions. Concurrency control and recovery mechanisms (e.g., locking and logging) are required to ensure the consistency and robustness of transactions. An OLAP query often needs read-only access of data records for summarization and aggregation. Concurrency control and recovery mechanisms, if applied for such OLAP operations, may jeopardize the execution of concurrent transactions and thus substantially reduce the throughput of an OLTP system.

Finally, the separation of operational databases from data warehouses is based on the different structures, contents, and uses of the data in these two systems. Decision

Table 4.1 Comparison of OLTP and OLAP Systems

Feature	*OLTP*	*OLAP*
Characteristic	operational processing	informational processing
Orientation	transaction	analysis
User	clerk, DBA, database professional	knowledge worker (e.g., manager, executive, analyst)
Function	day-to-day operations	long-term informational requirements decision support
DB design	ER-based, application-oriented	star/snowflake, subject-oriented
Data	current, guaranteed up-to-date	historic, accuracy maintained over time
Summarization	primitive, highly detailed	summarized, consolidated
View	detailed, flat relational	summarized, multidimensional
Unit of work	short, simple transaction	complex query
Access	read/write	mostly read
Focus	data in	information out
Operations	index/hash on primary key	lots of scans
Number of records accessed	tens	millions
Number of users	thousands	hundreds
DB size	GB to high-order GB	$\geq$ TB
Priority	high performance, high availability	high flexibility, end-user autonomy
Metric	transaction throughput	query throughput, response time

Note: Table is partially based on Chaudhuri and Dayal [CD97].

support requires historic data, whereas operational databases do not typically maintain historic data. In this context, the data in operational databases, though abundant, are usually far from complete for decision making. Decision support requires consolidation (e.g., aggregation and summarization) of data from heterogeneous sources, resulting in high-quality, clean, integrated data. In contrast, operational databases contain only detailed raw data, such as transactions, which need to be consolidated before analysis. Because the two systems provide quite different functionalities and require different kinds of data, it is presently necessary to maintain separate databases. However, many vendors of operational relational database management systems are beginning to optimize such systems to support OLAP queries. As this trend continues, the separation between OLTP and OLAP systems is expected to decrease.

4.1.4 Data Warehousing: A Multitiered Architecture

Data warehouses often adopt a three-tier architecture, as presented in Figure 4.1.

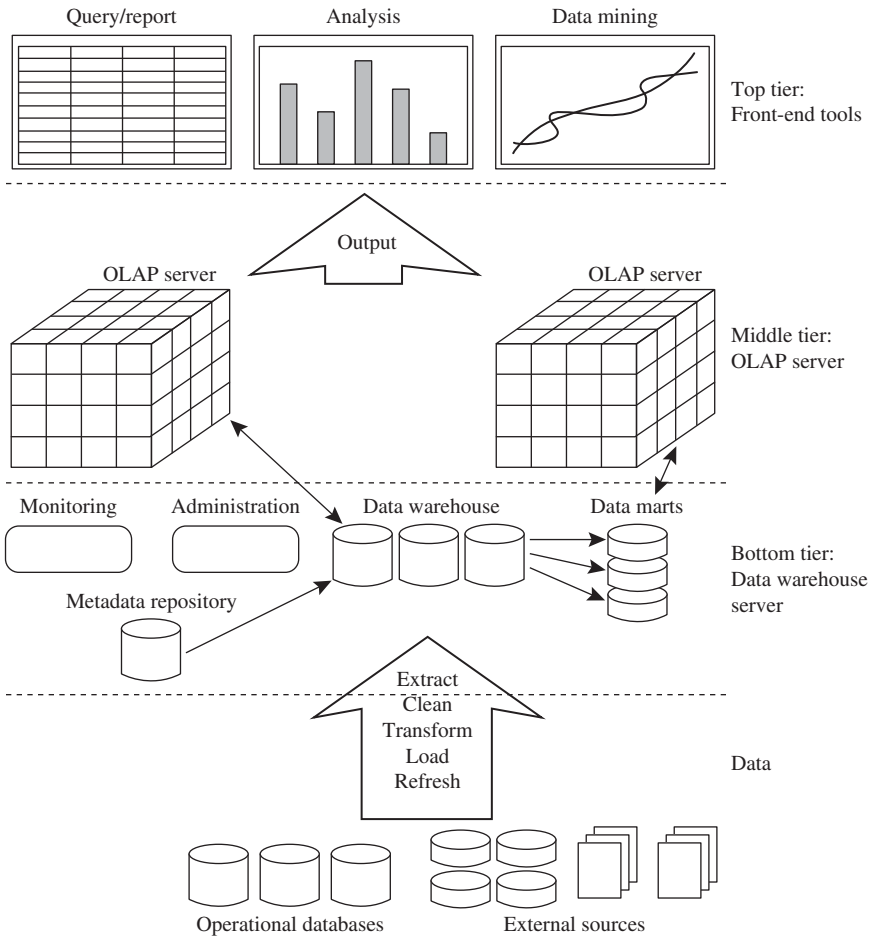


Figure 4.1 A three-tier data warehousing architecture.

1. The bottom tier is a **warehouse database server** that is almost always a relational database system. Back-end tools and utilities are used to feed data into the bottom tier from operational databases or other external sources (e.g., customer profile information provided by external consultants). These tools and utilities perform data extraction, cleaning, and transformation (e.g., to merge similar data from different sources into a unified format), as well as load and refresh functions to update the data warehouse (see Section 4.1.6). The data are extracted using application program interfaces known as **gateways**. A gateway is supported by the underlying DBMS and allows client programs to generate SQL code to be executed at a server. Examples of gateways include ODBC (Open Database Connection) and OLEDB (Object

Linking and Embedding Database) by Microsoft and JDBC (Java Database Connection). This tier also contains a metadata repository, which stores information about the data warehouse and its contents. The metadata repository is further described in Section 4.1.7.

2. The middle tier is an **OLAP server** that is typically implemented using either (1) a **relational OLAP (ROLAP)** model (i.e., an extended relational DBMS that maps operations on multidimensional data to standard relational operations); or (2) a **multidimensional OLAP (MOLAP)** model (i.e., a special-purpose server that directly implements multidimensional data and operations). OLAP servers are discussed in Section 4.4.4.
3. The top tier is a **front-end client layer**, which contains query and reporting tools, analysis tools, and/or data mining tools (e.g., trend analysis, prediction, and so on).

4.1.5 Data Warehouse Models: Enterprise Warehouse, Data Mart, and Virtual Warehouse

From the architecture point of view, there are three data warehouse models: the *enterprise warehouse*, the *data mart*, and the *virtual warehouse*.

Enterprise warehouse: An enterprise warehouse collects all of the information about subjects spanning the entire organization. It provides corporate-wide data integration, usually from one or more operational systems or external information providers, and is cross-functional in scope. It typically contains detailed data as well as summarized data, and can range in size from a few gigabytes to hundreds of gigabytes, terabytes, or beyond. An enterprise data warehouse may be implemented on traditional mainframes, computer superservers, or parallel architecture platforms. It requires extensive business modeling and may take years to design and build.

Data mart: A data mart contains a subset of corporate-wide data that is of value to a specific group of users. The scope is confined to specific selected subjects. For example, a marketing data mart may confine its subjects to customer, item, and sales. The data contained in data marts tend to be summarized.

Data marts are usually implemented on low-cost departmental servers that are Unix/Linux or Windows based. The implementation cycle of a data mart is more likely to be measured in weeks rather than months or years. However, it may involve complex integration in the long run if its design and planning were not enterprise-wide.

Depending on the source of data, data marts can be categorized as independent or dependent. *Independent* data marts are sourced from data captured from one or more operational systems or external information providers, or from data generated locally within a particular department or geographic area. *Dependent* data marts are sourced directly from enterprise data warehouses.

Virtual warehouse: A virtual warehouse is a set of views over operational databases. For efficient query processing, only some of the possible summary views may be materialized. A virtual warehouse is easy to build but requires excess capacity on operational database servers.

"*What are the pros and cons of the top-down and bottom-up approaches to data warehouse development?*" The top-down development of an enterprise warehouse serves as a systematic solution and minimizes integration problems. However, it is expensive, takes a long time to develop, and lacks flexibility due to the difficulty in achieving consistency and consensus for a common data model for the entire organization. The bottom-up approach to the design, development, and deployment of independent data marts provides flexibility, low cost, and rapid return of investment. It, however, can lead to problems when integrating various disparate data marts into a consistent enterprise data warehouse.

A recommended method for the development of data warehouse systems is to implement the warehouse in an incremental and evolutionary manner, as shown in Figure 4.2. First, a high-level corporate data model is defined within a reasonably short period (such as one or two months) that provides a corporate-wide, consistent, integrated view of data among different subjects and potential usages. This high-level model, although it will need to be refined in the further development of enterprise data warehouses and departmental data marts, will greatly reduce future integration problems. Second, independent data marts can be implemented in parallel with the enterprise

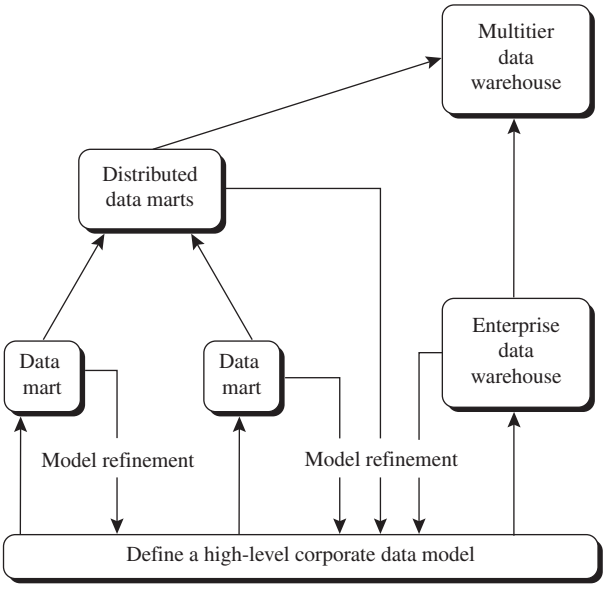


Figure 4.2 A recommended approach for data warehouse development.

warehouse based on the same corporate data model set noted before. Third, distributed data marts can be constructed to integrate different data marts via hub servers. Finally, a **multitier data warehouse** is constructed where the enterprise warehouse is the sole custodian of all warehouse data, which is then distributed to the various dependent data marts.

4.1.6 Extraction, Transformation, and Loading

Data warehouse systems use back-end tools and utilities to populate and refresh their data (Figure 4.1). These tools and utilities include the following functions:

- **Data extraction**, which typically gathers data from multiple, heterogeneous, and external sources.
- **Data cleaning**, which detects errors in the data and rectifies them when possible.
- **Data transformation**, which converts data from legacy or host format to warehouse format.
- **Load**, which sorts, summarizes, consolidates, computes views, checks integrity, and builds indices and partitions.
- **Refresh**, which propagates the updates from the data sources to the warehouse.

Besides cleaning, loading, refreshing, and metadata definition tools, data warehouse systems usually provide a good set of data warehouse management tools.

Data cleaning and data transformation are important steps in improving the data quality and, subsequently, the data mining results (see Chapter 3). Because we are mostly interested in the aspects of data warehousing technology related to data mining, we will not get into the details of the remaining tools, and recommend interested readers to consult books dedicated to data warehousing technology.

4.1.7 Metadata Repository

Metadata are data about data. When used in a data warehouse, metadata are the data that define warehouse objects. Figure 4.1 showed a metadata repository within the bottom tier of the data warehousing architecture. Metadata are created for the data names and definitions of the given warehouse. Additional metadata are created and captured for timestamping any extracted data, the source of the extracted data, and missing fields that have been added by data cleaning or integration processes.

A metadata repository should contain the following:

- A description of the *data warehouse structure*, which includes the warehouse schema, view, dimensions, hierarchies, and derived data definitions, as well as data mart locations and contents.

- *Operational metadata*, which include data lineage (history of migrated data and the sequence of transformations applied to it), currency of data (active, archived, or purged), and monitoring information (warehouse usage statistics, error reports, and audit trails).
- The *algorithms used for summarization*, which include measure and dimension definition algorithms, data on granularity, partitions, subject areas, aggregation, summarization, and predefined queries and reports.
- *Mapping from the operational environment to the data warehouse*, which includes source databases and their contents, gateway descriptions, data partitions, data extraction, cleaning, transformation rules and defaults, data refresh and purging rules, and security (user authorization and access control).
- *Data related to system performance*, which include indices and profiles that improve data access and retrieval performance, in addition to rules for the timing and scheduling of refresh, update, and replication cycles.
- *Business metadata*, which include business terms and definitions, data ownership information, and charging policies.

A data warehouse contains different levels of summarization, of which metadata is one. Other types include current detailed data (which are almost always on disk), older detailed data (which are usually on tertiary storage), lightly summarized data, and highly summarized data (which may or may not be physically housed).

Metadata play a very different role than other data warehouse data and are important for many reasons. For example, metadata are used as a directory to help the decision support system analyst locate the contents of the data warehouse, and as a guide to the data mapping when data are transformed from the operational environment to the data warehouse environment. Metadata also serve as a guide to the algorithms used for summarization between the current detailed data and the lightly summarized data, and between the lightly summarized data and the highly summarized data. Metadata should be stored and managed persistently (i.e., on disk).

4.2 Data Warehouse Modeling: Data Cube and OLAP

Data warehouses and OLAP tools are based on a **multidimensional data model**. This model views data in the form of a *data cube*. In this section, you will learn how data cubes model *n*-dimensional data (Section 4.2.1). In Section 4.2.2, various multidimensional models are shown: star schema, snowflake schema, and fact constellation. You will also learn about concept hierarchies (Section 4.2.3) and measures (Section 4.2.4) and how they can be used in basic OLAP operations to allow interactive mining at multiple levels of abstraction. Typical OLAP operations such as drill-down and roll-up are illustrated

(Section 4.2.5). Finally, the starnet model for querying multidimensional databases is presented (Section 4.2.6).

4.2.1 Data Cube: A Multidimensional Data Model

"*What is a data cube?*" A **data cube** allows data to be modeled and viewed in multiple dimensions. It is defined by dimensions and facts.

In general terms, **dimensions** are the perspectives or entities with respect to which an organization wants to keep records. For example, *AllElectronics* may create a *sales* data warehouse in order to keep records of the store's sales with respect to the dimensions *time*, *item*, *branch*, and *location*. These dimensions allow the store to keep track of things like monthly sales of items and the branches and locations at which the items were sold. Each dimension may have a table associated with it, called a **dimension table**, which further describes the dimension. For example, a dimension table for *item* may contain the attributes *item_name, brand*, and *type*. Dimension tables can be specified by users or experts, or automatically generated and adjusted based on data distributions.

A multidimensional data model is typically organized around a central theme, such as *sales*. This theme is represented by a fact table. **Facts** are numeric measures. Think of them as the quantities by which we want to analyze relationships between dimensions. Examples of facts for a sales data warehouse include *dollars_sold* (sales amount in dollars), *units_sold* (number of units sold), and *amount_budgeted*. The **fact table** contains the names of the *facts*, or measures, as well as keys to each of the related dimension tables. You will soon get a clearer picture of how this works when we look at multidimensional schemas.

Although we usually think of cubes as 3-D geometric structures, in data warehousing the data cube is *n*-dimensional. To gain a better understanding of data cubes and the multidimensional data model, let's start by looking at a simple 2-D data cube that is, in fact, a table or spreadsheet for sales data from *AllElectronics*. In particular, we will look at the *AllElectronics* sales data for items sold per quarter in the city of Vancouver. These data are shown in Table 4.2. In this 2-D representation, the sales for Vancouver are shown with respect to the *time* dimension (organized in quarters) and the *item* dimension (organized according to the types of items sold). The fact or measure displayed is *dollars_sold* (in thousands).

Now, suppose that we would like to view the sales data with a third dimension. For instance, suppose we would like to view the data according to *time* and *item*, as well as *location*, for the cities Chicago, New York, Toronto, and Vancouver. These 3-D data are shown in Table 4.3. The 3-D data in the table are represented as a series of 2-D tables. Conceptually, we may also represent the same data in the form of a 3-D data cube, as in Figure 4.3.

Suppose that we would now like to view our sales data with an additional fourth dimension such as *supplier*. Viewing things in 4-D becomes tricky. However, we can think of a 4-D cube as being a series of 3-D cubes, as shown in Figure 4.4. If we continue

Table 4.2 2-D View of Sales Data for *AllElectronics* According to *time* and *item*

	location = *"Vancouver"*			
	item *(type)*			
time *(quarter)*	*home entertainment*	*computer*	*phone*	*security*
Q1	605	825	14	400
Q2	680	952	31	512
Q3	812	1023	30	501
Q4	927	1038	38	580

Note: The sales are from branches located in the city of Vancouver. The measure displayed is *dollars_sold* (in thousands).

Table 4.3 3-D View of Sales Data for *AllElectronics* According to *time*, *item*, and *location*

	location = "Chicago"				***location*** = "New York"				***location*** = "Toronto"				***location*** = "Vancouver"			
	item				***item***				***item***				***item***			
	home				*home*				*home*				*home*			
time	*ent.*	*comp.*	*phone*	*sec.*	*ent.*	*comp.*	*phone*	*sec.*	*ent.*	*comp.*	*phone*	*sec.*	*ent.*	*comp.*	*phone*	*sec.*
Q1	854	882	89	623	1087	968	38	872	818	746	43	591	605	825	14	400
Q2	943	890	64	698	1130	1024	41	925	894	769	52	682	680	952	31	512
Q3	1032	924	59	789	1034	1048	45	1002	940	795	58	728	812	1023	30	501
Q4	1129	992	63	870	1142	1091	54	984	978	864	59	784	927	1038	38	580

Note: The measure displayed is *dollars_sold* (in thousands).

in this way, we may display any n-dimensional data as a series of $(n-1)$-dimensional "cubes." The data cube is a metaphor for multidimensional data storage. The actual physical storage of such data may differ from its logical representation. The important thing to remember is that data cubes are n-dimensional and do not confine data to 3-D.

Tables 4.2 and 4.3 show the data at different degrees of summarization. In the data warehousing research literature, a data cube like those shown in Figures 4.3 and 4.4 is often referred to as a **cuboid**. Given a set of dimensions, we can generate a cuboid for each of the possible subsets of the given dimensions. The result would form a *lattice* of cuboids, each showing the data at a different level of summarization, or group-by. The lattice of cuboids is then referred to as a data cube. Figure 4.5 shows a lattice of cuboids forming a data cube for the dimensions *time*, *item*, *location*, and *supplier*.

The cuboid that holds the lowest level of summarization is called the **base cuboid**. For example, the 4-D cuboid in Figure 4.4 is the base cuboid for the given *time*, *item*, *location*, and *supplier* dimensions. Figure 4.3 is a 3-D (nonbase) cuboid for *time*, *item*,

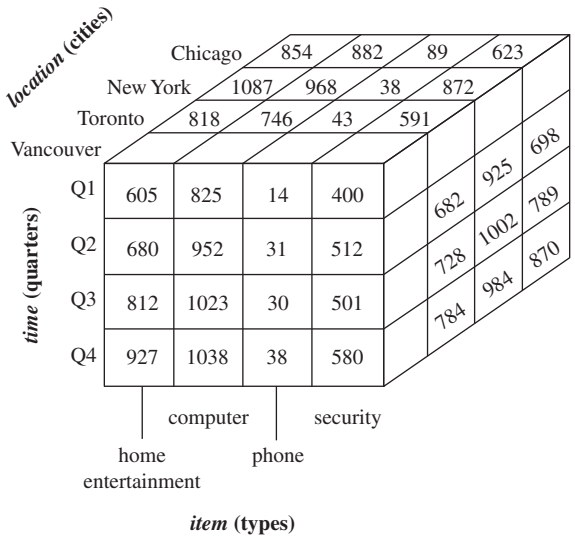


Figure 4.3 A 3-D data cube representation of the data in Table 4.3, according to *time*, *item*, and *location*. The measure displayed is *dollars_sold* (in thousands).

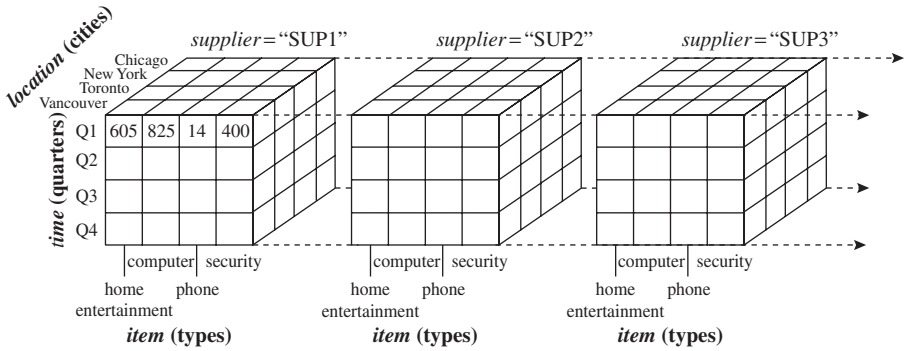


Figure 4.4 A 4-D data cube representation of sales data, according to *time*, *item*, *location*, and *supplier*. The measure displayed is *dollars_sold* (in thousands). For improved readability, only some of the cube values are shown.

and *location*, summarized for all suppliers. The 0-D cuboid, which holds the highest level of summarization, is called the **apex cuboid**. In our example, this is the total sales, or *dollars_sold*, summarized over all four dimensions. The apex cuboid is typically denoted by all.

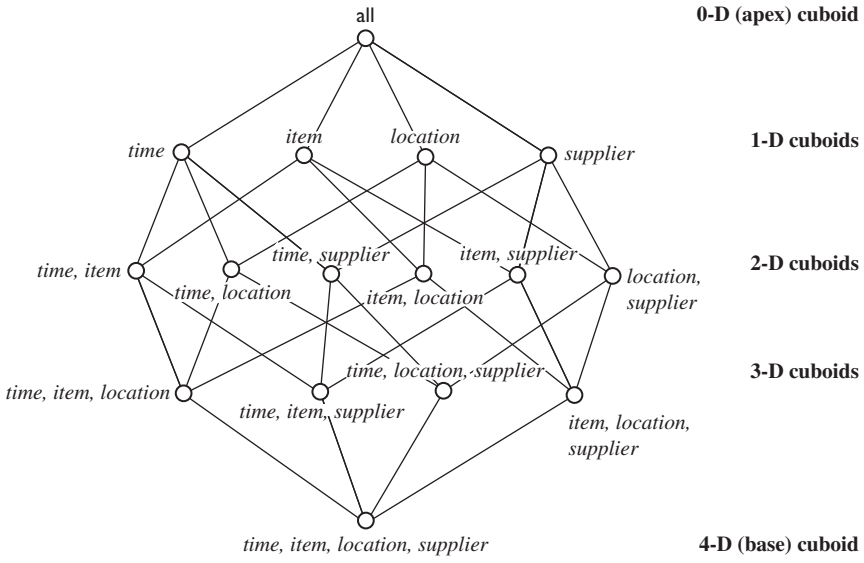


Figure 4.5 Lattice of cuboids, making up a 4-D data cube for *time*, *item*, *location*, and *supplier*. Each cuboid represents a different degree of summarization.

4.2.2 Stars, Snowflakes, and Fact Constellations: Schemas for Multidimensional Data Models

The entity-relationship data model is commonly used in the design of relational databases, where a database schema consists of a set of entities and the relationships between them. Such a data model is appropriate for online transaction processing. A data warehouse, however, requires a concise, subject-oriented schema that facilitates online data analysis.

The most popular data model for a data warehouse is a **multidimensional model**, which can exist in the form of a **star schema**, a **snowflake schema**, or a **fact constellation schema**. Let's look at each of these.

Star schema: The most common modeling paradigm is the star schema, in which the data warehouse contains (1) a large central table (**fact table**) containing the bulk of the data, with no redundancy, and (2) a set of smaller attendant tables (**dimension tables**), one for each dimension. The schema graph resembles a starburst, with the dimension tables displayed in a radial pattern around the central fact table.

Example 4.1 **Star schema.** A star schema for *AllElectronics* sales is shown in Figure 4.6. Sales are considered along four dimensions: *time, item, branch*, and *location*. The schema contains a central fact table for *sales* that contains keys to each of the four dimensions, along

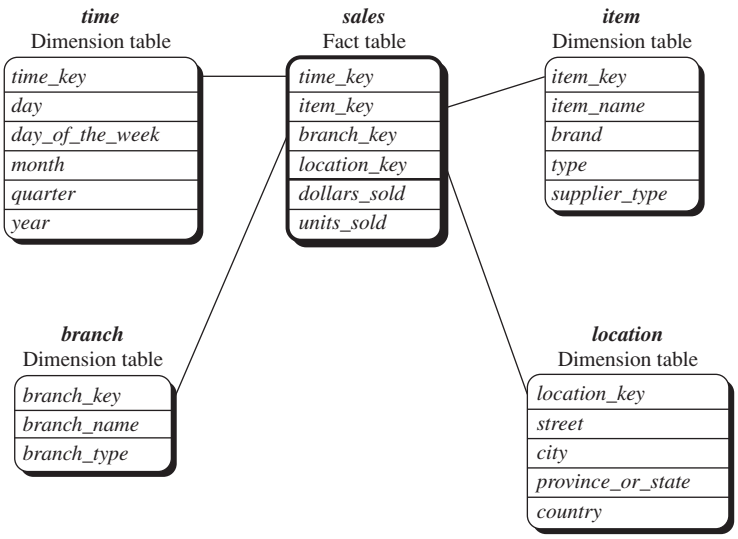


Figure 4.6 Star schema of *sales* data warehouse.

with two measures: *dollars_sold* and *units_sold*. To minimize the size of the fact table, dimension identifiers (e.g., *time_key* and *item_key*) are system-generated identifiers. ■

Notice that in the star schema, each dimension is represented by only one table, and each table contains a set of attributes. For example, the *location* dimension table contains the attribute set {*location_key, street, city, province_or_state, country*}. This constraint may introduce some redundancy. For example, "Urbana" and "Chicago" are both cities in the state of Illinois, USA. Entries for such cities in the *location* dimension table will create redundancy among the attributes *province_or_state* and *country*; that is, (..., Urbana, IL, USA) and (..., Chicago, IL, USA). Moreover, the attributes within a dimension table may form either a hierarchy (total order) or a lattice (partial order).

Snowflake schema: The snowflake schema is a variant of the star schema model, where some dimension tables are *normalized*, thereby further splitting the data into additional tables. The resulting schema graph forms a shape similar to a snowflake.

The major difference between the snowflake and star schema models is that the dimension tables of the snowflake model may be kept in normalized form to reduce redundancies. Such a table is easy to maintain and saves storage space. However, this space savings is negligible in comparison to the typical magnitude of the fact table. Furthermore, the snowflake structure can reduce the effectiveness of browsing, since more joins will be needed to execute a query. Consequently, the system performance may be adversely impacted. Hence, although the snowflake schema reduces redundancy, it is not as popular as the star schema in data warehouse design.

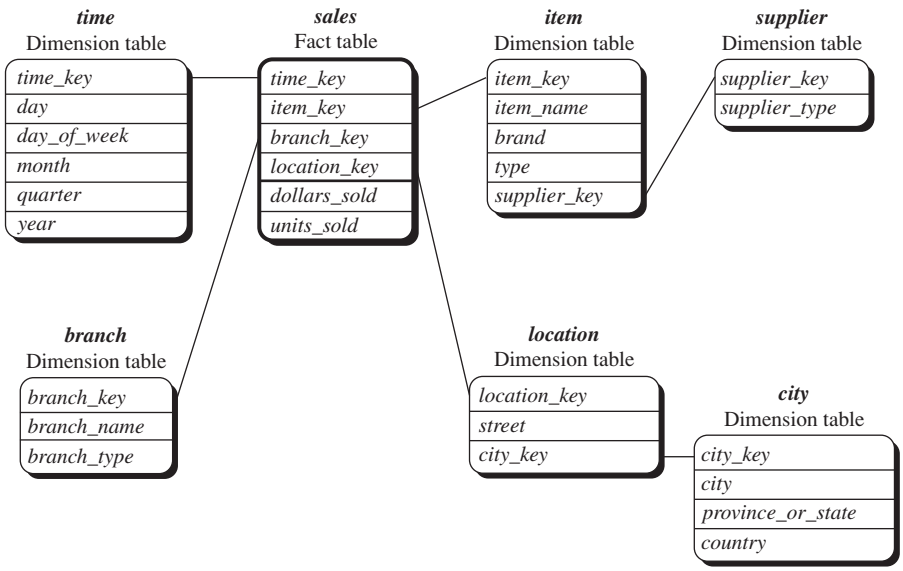


Figure 4.7 Snowflake schema of a *sales* data warehouse.

Example 4.2 **Snowflake schema.** A snowflake schema for *AllElectronics* sales is given in Figure 4.7. Here, the *sales* fact table is identical to that of the star schema in Figure 4.6. The main difference between the two schemas is in the definition of dimension tables. The single dimension table for *item* in the star schema is normalized in the snowflake schema, resulting in new *item* and *supplier* tables. For example, the *item* dimension table now contains the attributes *item_key, item_name, brand, type*, and *supplier_key*, where *supplier_key* is linked to the *supplier* dimension table, containing *supplier_key* and *supplier_type* information. Similarly, the single dimension table for *location* in the star schema can be normalized into two new tables: *location* and *city*. The *city_key* in the new *location* table links to the *city* dimension. Notice that, when desirable, further normalization can be performed on *province_or_state* and *country* in the snowflake schema shown in Figure 4.7. ■

Fact constellation: Sophisticated applications may require multiple fact tables to *share* dimension tables. This kind of schema can be viewed as a collection of stars, and hence is called a **galaxy schema** or a **fact constellation**.

Example 4.3 **Fact constellation.** A fact constellation schema is shown in Figure 4.8. This schema specifies two fact tables, *sales* and *shipping*. The *sales* table definition is identical to that of the star schema (Figure 4.6). The *shipping* table has five dimensions, or keys—*item_key, time_key, shipper_key, from_location*, and *to_location*—and two measures—*dollars_cost*

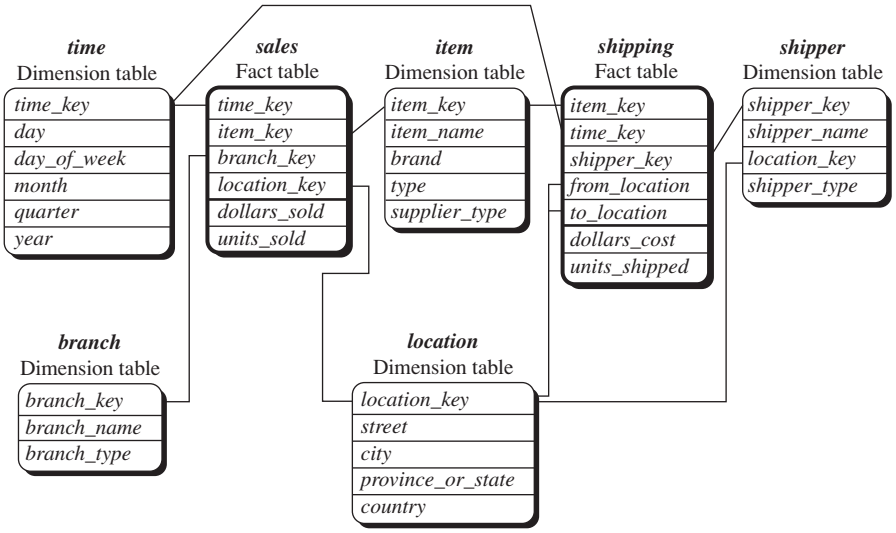


Figure 4.8 Fact constellation schema of a sales and shipping data warehouse.

and *units_shipped*. A fact constellation schema allows dimension tables to be shared between fact tables. For example, the dimensions tables for *time, item*, and *location* are shared between the *sales* and *shipping* fact tables. ■

In data warehousing, there is a distinction between a data warehouse and a data mart. A data warehouse collects information about subjects that span the *entire organization*, such as *customers, items, sales, assets*, and *personnel*, and thus its scope is *enterprise-wide*. For data warehouses, the fact constellation schema is commonly used, since it can model multiple, interrelated subjects. A **data mart**, on the other hand, is a department subset of the data warehouse that focuses on selected subjects, and thus its scope is *department-wide*. For data marts, the *star* or *snowflake* schema is commonly used, since both are geared toward modeling single subjects, although the star schema is more popular and efficient.

4.2.3 Dimensions: The Role of Concept Hierarchies

A **concept hierarchy** defines a sequence of mappings from a set of low-level concepts to higher-level, more general concepts. Consider a concept hierarchy for the dimension *location*. City values for *location* include Vancouver, Toronto, New York, and Chicago. Each city, however, can be mapped to the province or state to which it belongs. For example, Vancouver can be mapped to British Columbia, and Chicago to Illinois. The provinces and states can in turn be mapped to the country (e.g., Canada or the United States) to which they belong. These mappings form a concept hierarchy for the

dimension *location*, mapping a set of low-level concepts (i.e., cities) to higher-level, more general concepts (i.e., countries). This concept hierarchy is illustrated in Figure 4.9.

Many concept hierarchies are implicit within the database schema. For example, suppose that the dimension *location* is described by the attributes *number, street, city, province_or_state, zip_code*, and *country*. These attributes are related by a total order, forming a concept hierarchy such as "*street* < *city* < *province_or_state* < *country*." This hierarchy is shown in Figure 4.10(a). Alternatively, the attributes of a dimension may

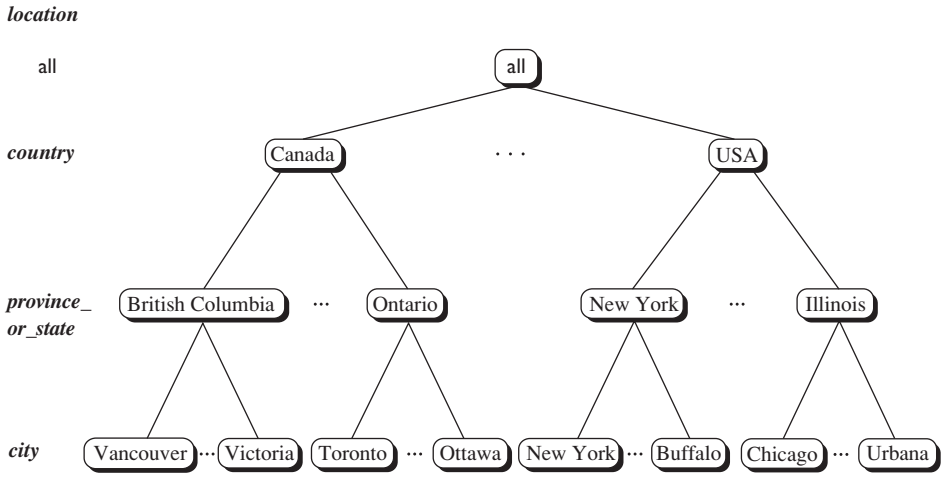


Figure 4.9 A concept hierarchy for *location*. Due to space limitations, not all of the hierarchy nodes are shown, indicated by ellipses between nodes.

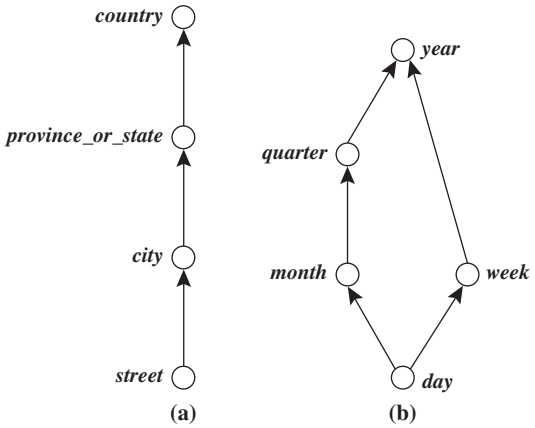


Figure 4.10 Hierarchical and lattice structures of attributes in warehouse dimensions: (a) a hierarchy for *location* and (b) a lattice for *time*.

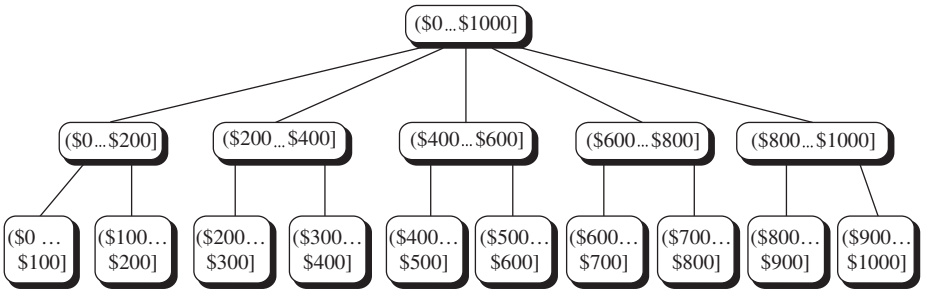


Figure 4.11 A concept hierarchy for *price*.

be organized in a partial order, forming a lattice. An example of a partial order for the *time* dimension based on the attributes *day, week, month, quarter*, and *year* is "*day* < {*month* < *quarter; week*} < *year*."[1] This lattice structure is shown in Figure 4.10(b). A concept hierarchy that is a total or partial order among attributes in a database schema is called a **schema hierarchy**. Concept hierarchies that are common to many applications (e.g., *for time*) may be predefined in the data mining system. Data mining systems should provide users with the flexibility to tailor predefined hierarchies according to their particular needs. For example, users may want to define a fiscal year starting on April 1 or an academic year starting on September 1.

Concept hierarchies may also be defined by discretizing or grouping values for a given dimension or attribute, resulting in a **set-grouping hierarchy**. A total or partial order can be defined among groups of values. An example of a set-grouping hierarchy is shown in Figure 4.11 for the dimension *price*, where an interval ($\$X \ldots \$Y$] denotes the range from $\$X$ (exclusive) to $\$Y$ (inclusive).

There may be more than one concept hierarchy for a given attribute or dimension, based on different user viewpoints. For instance, a user may prefer to organize *price* by defining ranges for *inexpensive, moderately_priced*, and *expensive*.

Concept hierarchies may be provided manually by system users, domain experts, or knowledge engineers, or may be automatically generated based on statistical analysis of the data distribution. The automatic generation of concept hierarchies is discussed in Chapter 3 as a preprocessing step in preparation for data mining.

Concept hierarchies allow data to be handled at varying levels of abstraction, as we will see in Section 4.2.4.

4.2.4 Measures: Their Categorization and Computation

"How are measures computed?" To answer this question, we first study how measures can be categorized. Note that a *multidimensional point* in the data cube space can be defined

[1] Since a *week* often crosses the boundary of two consecutive months, it is usually not treated as a lower abstraction of *month*. Instead, it is often treated as a lower abstraction of *year*, since a year contains approximately 52 weeks.

by a set of dimension–value pairs; for example, ⟨*time* = "Q1", *location* = "Vancouver", *item* = "computer"⟩. A data cube **measure** is a numeric function that can be evaluated at each point in the data cube space. A measure value is computed for a given point by aggregating the data corresponding to the respective dimension–value pairs defining the given point. We will look at concrete examples of this shortly.

Measures can be organized into three categories—distributive, algebraic, and holistic—based on the kind of aggregate functions used.

Distributive: An aggregate function is *distributive* if it can be computed in a distributed manner as follows. Suppose the data are partitioned into n sets. We apply the function to each partition, resulting in n aggregate values. If the result derived by applying the function to the n aggregate values is the same as that derived by applying the function to the entire data set (without partitioning), the function can be computed in a distributed manner. For example, sum() can be computed for a data cube by first partitioning the cube into a set of subcubes, computing sum() for each subcube, and then summing up the counts obtained for each subcube. Hence, sum() is a distributive aggregate function.

For the same reason, count(), min(), and max() are distributive aggregate functions. By treating the count value of each nonempty base cell as 1 by default, count() of any cell in a cube can be viewed as the sum of the count values of all of its corresponding child cells in its subcube. Thus, count() is distributive. A measure is *distributive* if it is obtained by applying a distributive aggregate function. Distributive measures can be computed efficiently because of the way the computation can be partitioned.

Algebraic: An aggregate function is *algebraic* if it can be computed by an algebraic function with M arguments (where M is a bounded positive integer), each of which is obtained by applying a distributive aggregate function. For example, avg() (average) can be computed by sum()/count(), where both sum() and count() are distributive aggregate functions. Similarly, it can be shown that min_N() and max_N() (which find the N minimum and N maximum values, respectively, in a given set) and standard_deviation() are algebraic aggregate functions. A measure is *algebraic* if it is obtained by applying an algebraic aggregate function.

Holistic: An aggregate function is *holistic* if there is no constant bound on the storage size needed to describe a subaggregate. That is, there does not exist an algebraic function with M arguments (where M is a constant) that characterizes the computation. Common examples of holistic functions include median(), mode(), and rank(). A measure is *holistic* if it is obtained by applying a holistic aggregate function.

Most large data cube applications require efficient computation of distributive and algebraic measures. Many efficient techniques for this exist. In contrast, it is difficult to compute holistic measures efficiently. Efficient techniques to *approximate* the computation of some holistic measures, however, do exist. For example, rather than computing the exact median(), Equation (2.3) of Chapter 2 can be used to estimate the approximate median value for a large data set. In many cases, such techniques are sufficient to overcome the difficulties of efficient computation of holistic measures.

Various methods for computing different measures in data cube construction are discussed in depth in Chapter 5. Notice that most of the current data cube technology confines the measures of multidimensional databases to *numeric data*. However, measures can also be applied to other kinds of data, such as spatial, multimedia, or text data.

4.2.5 Typical OLAP Operations

"*How are concept hierarchies useful in OLAP?*" In the multidimensional model, data are organized into multiple dimensions, and each dimension contains multiple levels of abstraction defined by concept hierarchies. This organization provides users with the flexibility to view data from different perspectives. A number of OLAP data cube operations exist to materialize these different views, allowing interactive querying and analysis of the data at hand. Hence, OLAP provides a user-friendly environment for interactive data analysis.

Example 4.4 **OLAP operations.** Let's look at some typical OLAP operations for multidimensional data. Each of the following operations described is illustrated in Figure 4.12. At the center of the figure is a data cube for *AllElectronics* sales. The cube contains the dimensions *location, time*, and *item*, where *location* is aggregated with respect to city values, *time* is aggregated with respect to quarters, and *item* is aggregated with respect to item types. To aid in our explanation, we refer to this cube as the central cube. The measure displayed is *dollars_sold* (in thousands). (For improved readability, only some of the cubes' cell values are shown.) The data examined are for the cities Chicago, New York, Toronto, and Vancouver.

Roll-up: The roll-up operation (also called the *drill-up* operation by some vendors) performs aggregation on a data cube, either by *climbing up a concept hierarchy* for a dimension or by *dimension reduction*. Figure 4.12 shows the result of a roll-up operation performed on the central cube by climbing up the concept hierarchy for *location* given in Figure 4.9. This hierarchy was defined as the total order "*street* < *city* < *province_or_state* < *country*." The roll-up operation shown aggregates the data by ascending the *location* hierarchy from the level of *city* to the level of *country*. In other words, rather than grouping the data by city, the resulting cube groups the data by country.

When roll-up is performed by dimension reduction, one or more dimensions are removed from the given cube. For example, consider a sales data cube containing only the *location* and *time* dimensions. Roll-up may be performed by removing, say, the *time* dimension, resulting in an aggregation of the total sales by location, rather than by location and by time.

Drill-down: Drill-down is the reverse of roll-up. It navigates from less detailed data to more detailed data. Drill-down can be realized by either *stepping down a concept hierarchy* for a dimension or *introducing additional dimensions*. Figure 4.12 shows the result of a drill-down operation performed on the central cube by stepping down a

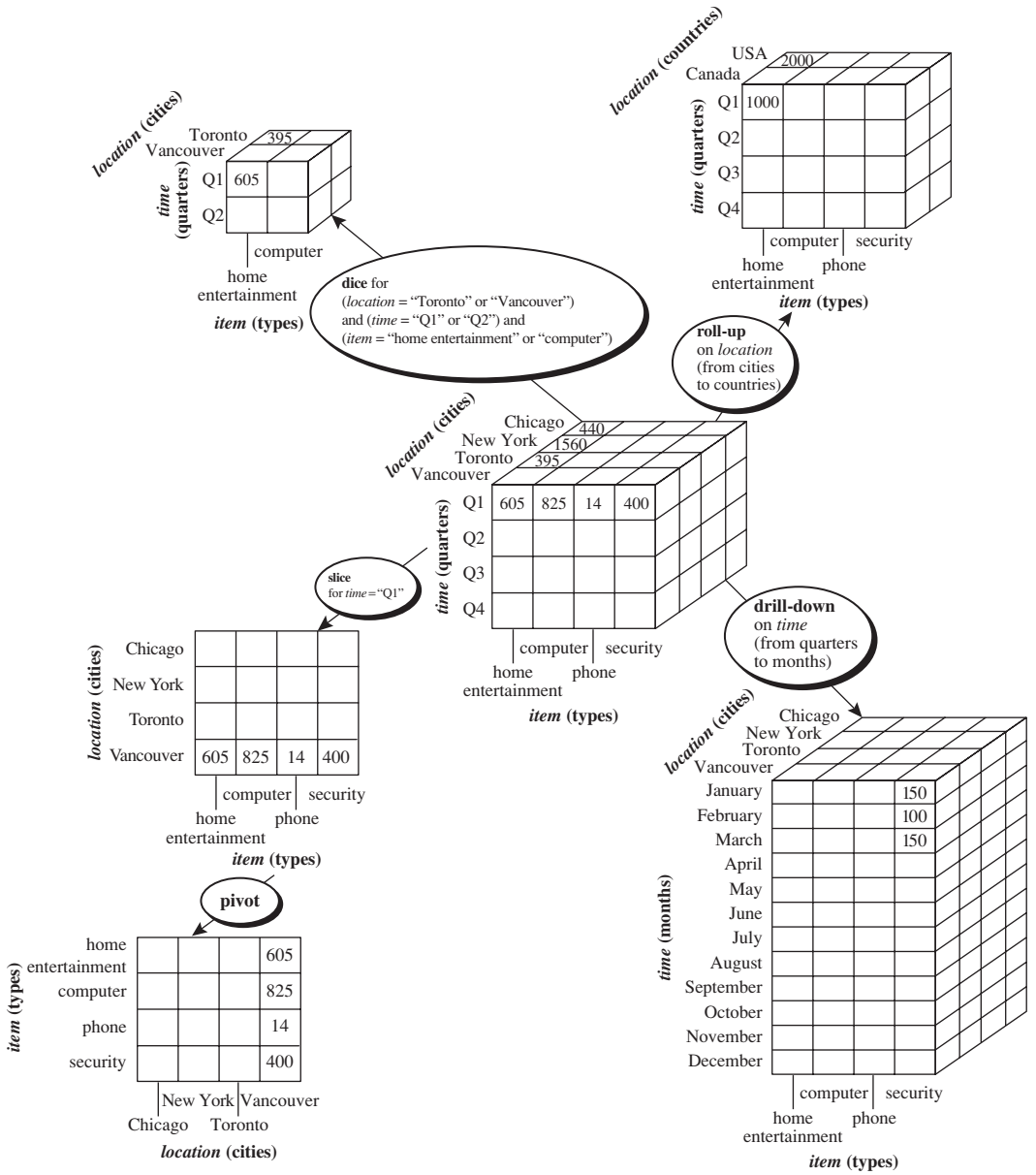


Figure 4.12 Examples of typical OLAP operations on multidimensional data.

concept hierarchy for *time* defined as "*day* < *month* < *quarter* < *year*." Drill-down occurs by descending the *time* hierarchy from the level of *quarter* to the more detailed level of *month*. The resulting data cube details the total sales per month rather than summarizing them by quarter.

Because a drill-down adds more detail to the given data, it can also be performed by adding new dimensions to a cube. For example, a drill-down on the central cube of Figure 4.12 can occur by introducing an additional dimension, such as *customer_group*.

Slice and dice: The *slice* operation performs a selection on one dimension of the given cube, resulting in a subcube. Figure 4.12 shows a slice operation where the sales data are selected from the central cube for the dimension *time* using the criterion *time* = "Q1." The *dice* operation defines a subcube by performing a selection on two or more dimensions. Figure 4.12 shows a dice operation on the central cube based on the following selection criteria that involve three dimensions: (*location* = "Toronto" or "Vancouver") and (*time* = "Q1" or "Q2") and (item = "home entertainment" or "computer").

Pivot (rotate): *Pivot* (also called *rotate*) is a visualization operation that rotates the data axes in view to provide an alternative data presentation. Figure 4.12 shows a pivot operation where the *item* and *location* axes in a 2-D slice are rotated. Other examples include rotating the axes in a 3-D cube, or transforming a 3-D cube into a series of 2-D planes.

Other OLAP operations: Some OLAP systems offer additional drilling operations. For example, **drill-across** executes queries involving (i.e., across) more than one fact table. The **drill-through** operation uses relational SQL facilities to drill through the bottom level of a data cube down to its back-end relational tables.

Other OLAP operations may include ranking the top N or bottom N items in lists, as well as computing moving averages, growth rates, interests, internal return rates, depreciation, currency conversions, and statistical functions. ■

OLAP offers analytical modeling capabilities, including a calculation engine for deriving ratios, variance, and so on, and for computing measures across multiple dimensions. It can generate summarizations, aggregations, and hierarchies at each granularity level and at every dimension intersection. OLAP also supports functional models for forecasting, trend analysis, and statistical analysis. In this context, an OLAP engine is a powerful data analysis tool.

OLAP Systems versus Statistical Databases

Many OLAP systems' characteristics (e.g., the use of a multidimensional data model and concept hierarchies, the association of measures with dimensions, and the notions of roll-up and drill-down) also exist in earlier work on statistical databases (SDBs). A **statistical database** is a database system that is designed to support statistical applications. Similarities between the two types of systems are rarely discussed, mainly due to differences in terminology and application domains.

OLAP and SDB systems, however, have distinguishing differences. While SDBs tend to focus on socioeconomic applications, OLAP has been targeted for business applications. Privacy issues regarding concept hierarchies are a major concern for SDBs. For example, given summarized socioeconomic data, it is controversial to allow users to view the corresponding low-level data. Finally, unlike SDBs, OLAP systems are designed for efficiently handling huge amounts of data.

4.2.6 A Starnet Query Model for Querying Multidimensional Databases

The querying of multidimensional databases can be based on a **starnet model**, which consists of radial lines emanating from a central point, where each line represents a concept hierarchy for a dimension. Each abstraction level in the hierarchy is called a **footprint**. These represent the granularities available for use by OLAP operations such as drill-down and roll-up.

Example 4.5 **Starnet.** A starnet query model for the *AllElectronics* data warehouse is shown in Figure 4.13. This starnet consists of four radial lines, representing concept hierarchies for the dimensions *location, customer, item*, and *time*, respectively. Each line consists of footprints representing abstraction levels of the dimension. For example, the *time* line has four footprints: "day," "month," "quarter," and "year." A concept hierarchy may involve a single attribute (e.g., *date* for the *time* hierarchy) or several attributes (e.g., the concept hierarchy for *location* involves the attributes *street, city, province_or_state*, and *country*). In order to examine the item sales at *AllElectronics*, users can roll up along the

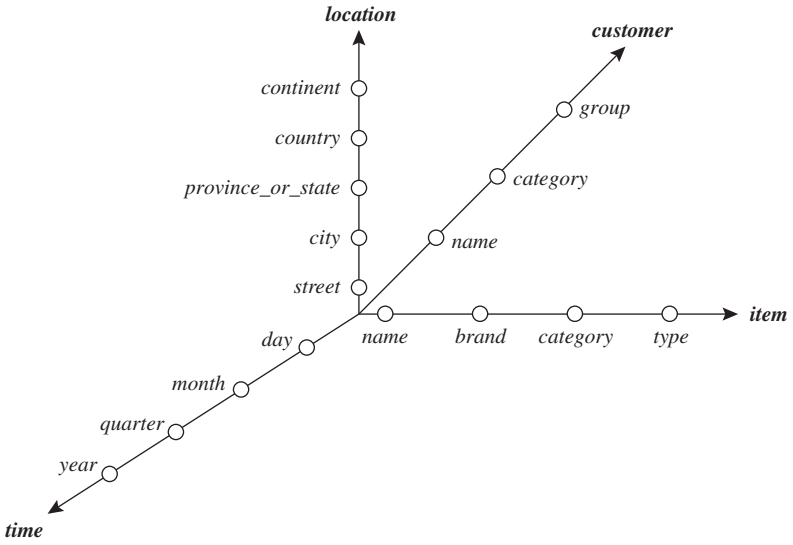


Figure 4.13 A starnet model of business queries.

time dimension from *month* to *quarter*, or, say, drill down along the *location* dimension from *country* to *city*.

Concept hierarchies can be used to **generalize** data by replacing low-level values (such as "day" for the *time* dimension) by higher-level abstractions (such as "year"), or to **specialize** data by replacing higher-level abstractions with lower-level values. ■

4.3 Data Warehouse Design and Usage

"*What goes into a data warehouse design? How are data warehouses used? How do data warehousing and OLAP relate to data mining?*" This section tackles these questions. We study the design and usage of data warehousing for information processing, analytical processing, and data mining. We begin by presenting a business analysis framework for data warehouse design (Section 4.3.1). Section 4.3.2 looks at the design process, while Section 4.3.3 studies data warehouse usage. Finally, Section 4.3.4 describes *multidimensional data mining*, a powerful paradigm that integrates OLAP with data mining technology.

4.3.1 A Business Analysis Framework for Data Warehouse Design

"*What can business analysts gain from having a data warehouse?*" First, having a data warehouse may provide a *competitive advantage* by presenting relevant information from which to measure performance and make critical adjustments to help win over competitors. Second, a data warehouse can enhance business *productivity* because it is able to quickly and efficiently gather information that accurately describes the organization. Third, a data warehouse facilitates *customer relationship management* because it provides a consistent view of customers and items across all lines of business, all departments, and all markets. Finally, a data warehouse may bring about *cost reduction* by tracking trends, patterns, and exceptions over long periods in a consistent and reliable manner.

To design an effective data warehouse we need to understand and analyze business needs and construct a *business analysis framework*. The construction of a large and complex information system can be viewed as the construction of a large and complex building, for which the owner, architect, and builder have different views. These views are combined to form a complex framework that represents the top-down, business-driven, or owner's perspective, as well as the bottom-up, builder-driven, or implementor's view of the information system.

Four different views regarding a data warehouse design must be considered: the *top-down view*, the *data source view*, the *data warehouse view*, and the *business query view*.

- The **top-down view** allows the selection of the relevant information necessary for the data warehouse. This information matches current and future business needs.
- The **data source view** exposes the information being captured, stored, and managed by operational systems. This information may be documented at various levels of detail and accuracy, from individual data source tables to integrated data source tables. Data sources are often modeled by traditional data modeling techniques, such as the entity-relationship model or CASE (computer-aided software engineering) tools.
- The **data warehouse view** includes fact tables and dimension tables. It represents the information that is stored inside the data warehouse, including precalculated totals and counts, as well as information regarding the source, date, and time of origin, added to provide historical context.
- Finally, the **business query view** is the data perspective in the data warehouse from the end-user's viewpoint.

Building and using a data warehouse is a complex task because it requires *business skills, technology skills*, and *program management skills*. Regarding *business skills*, building a data warehouse involves understanding how systems store and manage their data, how to build **extractors** that transfer data from the operational system to the data warehouse, and how to build **warehouse refresh software** that keeps the data warehouse reasonably up-to-date with the operational system's data. Using a data warehouse involves understanding the significance of the data it contains, as well as understanding and translating the business requirements into queries that can be satisfied by the data warehouse.

Regarding *technology skills*, data analysts are required to understand how to make assessments from quantitative information and derive facts based on conclusions from historic information in the data warehouse. These skills include the ability to discover patterns and trends, to extrapolate trends based on history and look for anomalies or paradigm shifts, and to present coherent managerial recommendations based on such analysis. Finally, *program management skills* involve the need to interface with many technologies, vendors, and end-users in order to deliver results in a timely and cost-effective manner.

4.3.2 Data Warehouse Design Process

Let's look at various approaches to the data warehouse design process and the steps involved.

A data warehouse can be built using a *top-down approach*, a *bottom-up approach*, or a *combination of both*. The **top-down approach** starts with overall design and planning. It is useful in cases where the technology is mature and well known, and where the business problems that must be solved are clear and well understood. The **bottom-up approach** starts with experiments and prototypes. This is useful in the early stage of business modeling and technology development. It allows an organization to move

forward at considerably less expense and to evaluate the technological benefits before making significant commitments. In the **combined approach**, an organization can exploit the planned and strategic nature of the top-down approach while retaining the rapid implementation and opportunistic application of the bottom-up approach.

From the software engineering point of view, the design and construction of a data warehouse may consist of the following steps: *planning, requirements study, problem analysis, warehouse design, data integration and testing*, and finally *deployment of the data warehouse*. Large software systems can be developed using one of two methodologies: the *waterfall method* or the *spiral method*. The **waterfall method** performs a structured and systematic analysis at each step before proceeding to the next, which is like a waterfall, falling from one step to the next. The **spiral method** involves the rapid generation of increasingly functional systems, with short intervals between successive releases. This is considered a good choice for data warehouse development, especially for data marts, because the turnaround time is short, modifications can be done quickly, and new designs and technologies can be adapted in a timely manner.

In general, the warehouse design process consists of the following steps:

1. Choose a *business process* to model (e.g., orders, invoices, shipments, inventory, account administration, sales, or the general ledger). If the business process is organizational and involves multiple complex object collections, a data warehouse model should be followed. However, if the process is departmental and focuses on the analysis of one kind of business process, a data mart model should be chosen.
2. Choose the business process *grain*, which is the fundamental, atomic level of data to be represented in the fact table for this process (e.g., individual transactions, individual daily snapshots, and so on).
3. Choose the *dimensions* that will apply to each fact table record. Typical dimensions are time, item, customer, supplier, warehouse, transaction type, and status.
4. Choose the *measures* that will populate each fact table record. Typical measures are numeric additive quantities like *dollars_sold* and *units_sold*.

Because data warehouse construction is a difficult and long-term task, its implementation scope should be clearly defined. The goals of an initial data warehouse implementation should be *specific, achievable*, and *measurable*. This involves determining the time and budget allocations, the subset of the organization that is to be modeled, the number of data sources selected, and the number and types of departments to be served.

Once a data warehouse is designed and constructed, the initial deployment of the warehouse includes initial installation, roll-out planning, training, and orientation. Platform upgrades and maintenance must also be considered. Data warehouse administration includes data refreshment, data source synchronization, planning for disaster recovery, managing access control and security, managing data growth, managing database performance, and data warehouse enhancement and extension. Scope

management includes controlling the number and range of queries, dimensions, and reports; limiting the data warehouse's size; or limiting the schedule, budget, or resources.

Various kinds of data warehouse design tools are available. **Data warehouse development tools** provide functions to define and edit metadata repository contents (e.g., schemas, scripts, or rules), answer queries, output reports, and ship metadata to and from relational database system catalogs. **Planning and analysis tools** study the impact of schema changes and of refresh performance when changing refresh rates or time windows.

4.3.3 Data Warehouse Usage for Information Processing

Data warehouses and data marts are used in a wide range of applications. Business executives use the data in data warehouses and data marts to perform data analysis and make strategic decisions. In many firms, data warehouses are used as an integral part of a *plan-execute-assess* "closed-loop" feedback system for enterprise management. Data warehouses are used extensively in banking and financial services, consumer goods and retail distribution sectors, and controlled manufacturing such as demand-based production.

Typically, the longer a data warehouse has been in use, the more it will have evolved. This evolution takes place throughout a number of phases. Initially, the data warehouse is mainly used for generating reports and answering predefined queries. Progressively, it is used to analyze summarized and detailed data, where the results are presented in the form of reports and charts. Later, the data warehouse is used for strategic purposes, performing multidimensional analysis and sophisticated slice-and-dice operations. Finally, the data warehouse may be employed for knowledge discovery and strategic decision making using data mining tools. In this context, the tools for data warehousing can be categorized into *access and retrieval tools*, *database reporting tools*, *data analysis tools*, and *data mining tools*.

Business users need to have the means to know what exists in the data warehouse (through metadata), how to access the contents of the data warehouse, how to examine the contents using analysis tools, and how to present the results of such analysis.

There are three kinds of data warehouse applications: *information processing, analytical processing*, and *data mining*.

- **Information processing** supports querying, basic statistical analysis, and reporting using crosstabs, tables, charts, or graphs. A current trend in data warehouse information processing is to construct low-cost web-based accessing tools that are then integrated with web browsers.
- **Analytical processing** supports basic OLAP operations, including slice-and-dice, drill-down, roll-up, and pivoting. It generally operates on historic data in both summarized and detailed forms. The major strength of online analytical processing over information processing is the multidimensional data analysis of data warehouse data.

- **Data mining** supports knowledge discovery by finding hidden patterns and associations, constructing analytical models, performing classification and prediction, and presenting the mining results using visualization tools.

"*How does data mining relate to information processing and online analytical processing?*" Information processing, based on queries, can find useful information. However, answers to such queries reflect the information directly stored in databases or computable by aggregate functions. They do not reflect sophisticated patterns or regularities buried in the database. Therefore, information processing is not data mining.

Online analytical processing comes a step closer to data mining because it can derive information summarized at multiple granularities from user-specified subsets of a data warehouse. Such descriptions are equivalent to the class/concept descriptions discussed in Chapter 1. Because data mining systems can also mine generalized class/concept descriptions, this raises some interesting questions: *"Do OLAP systems perform data mining? Are OLAP systems actually data mining systems?"*

The functionalities of OLAP and data mining can be viewed as disjoint: OLAP is a data summarization/aggregation *tool* that helps simplify data analysis, while data mining allows the *automated discovery* of implicit patterns and interesting knowledge hidden in large amounts of data. OLAP tools are targeted toward simplifying and supporting interactive data analysis, whereas the goal of data mining tools is to automate as much of the process as possible, while still allowing users to guide the process. In this sense, data mining goes one step beyond traditional online analytical processing.

An alternative and broader view of data mining may be adopted in which data mining covers both data description and data modeling. Because OLAP systems can present general descriptions of data from data warehouses, OLAP functions are essentially for user-directed data summarization and comparison (by drilling, pivoting, slicing, dicing, and other operations). These are, though limited, data mining functionalities. Yet according to this view, data mining covers a much broader spectrum than simple OLAP operations, because it performs not only data summarization and comparison but also association, classification, prediction, clustering, time-series analysis, and other data analysis tasks.

Data mining is not confined to the analysis of data stored in data warehouses. It may analyze data existing at more detailed granularities than the summarized data provided in a data warehouse. It may also analyze transactional, spatial, textual, and multimedia data that are difficult to model with current multidimensional database technology. In this context, data mining covers a broader spectrum than OLAP with respect to data mining functionality and the complexity of the data handled.

Because data mining involves more automated and deeper analysis than OLAP, it is expected to have broader applications. Data mining can help business managers find and reach more suitable customers, as well as gain critical business insights that may help drive market share and raise profits. In addition, data mining can help managers understand customer group characteristics and develop optimal pricing strategies accordingly. It can correct item bundling based not on intuition but on actual item groups derived from customer purchase patterns, reduce promotional spending, and at the same time increase the overall net effectiveness of promotions.

4.3.4 From Online Analytical Processing to Multidimensional Data Mining

The data mining field has conducted substantial research regarding mining on various data types, including relational data, data from data warehouses, transaction data, time-series data, spatial data, text data, and flat files. **Multidimensional data mining** (also known as *exploratory multidimensional data mining*, **online analytical mining**, or **OLAM**) integrates OLAP with data mining to uncover knowledge in multidimensional databases. Among the many different paradigms and architectures of data mining systems, multidimensional data mining is particularly important for the following reasons:

- **High quality of data in data warehouses:** Most data mining tools need to work on integrated, consistent, and cleaned data, which requires costly data cleaning, data integration, and data transformation as preprocessing steps. A data warehouse constructed by such preprocessing serves as a valuable source of high-quality data for OLAP as well as for data mining. Notice that data mining may serve as a valuable tool for data cleaning and data integration as well.
- **Available information processing infrastructure surrounding data warehouses:** Comprehensive information processing and data analysis infrastructures have been or will be systematically constructed surrounding data warehouses, which include accessing, integration, consolidation, and transformation of multiple heterogeneous databases, ODBC/OLEDB connections, Web accessing and service facilities, and reporting and OLAP analysis tools. It is prudent to make the best use of the available infrastructures rather than constructing everything from scratch.
- **OLAP-based exploration of multidimensional data:** Effective data mining needs exploratory data analysis. A user will often want to traverse through a database, select portions of relevant data, analyze them at different granularities, and present knowledge/results in different forms. Multidimensional data mining provides facilities for mining on different subsets of data and at varying levels of abstraction—by drilling, pivoting, filtering, dicing, and slicing on a data cube and/or intermediate data mining results. This, together with data/knowledge visualization tools, greatly enhances the power and flexibility of data mining.
- **Online selection of data mining functions:** Users may not always know the specific kinds of knowledge they want to mine. By integrating OLAP with various data mining functions, multidimensional data mining provides users with the flexibility to select desired data mining functions and swap data mining tasks dynamically.

Chapter 5 describes data warehouses on a finer level by exploring implementation issues such as data cube computation, OLAP query answering strategies, and multidimensional data mining. The chapters following it are devoted to the study of data mining techniques. As we have seen, the introduction to data warehousing and OLAP technology presented in this chapter is essential to our study of data mining. This is because data warehousing provides users with large amounts of clean, organized,

and summarized data, which greatly facilitates data mining. For example, rather than storing the details of each sales transaction, a data warehouse may store a summary of the transactions per item type for each branch or, summarized to a higher level, for each country. The capability of OLAP to provide multiple and dynamic views of summarized data in a data warehouse sets a solid foundation for successful data mining.

Moreover, we also believe that data mining should be a human-centered process. Rather than asking a data mining system to generate patterns and knowledge automatically, a user will often need to interact with the system to perform exploratory data analysis. OLAP sets a good example for interactive data analysis and provides the necessary preparations for exploratory data mining. Consider the discovery of association patterns, for example. Instead of mining associations at a primitive (i.e., low) data level among transactions, users should be allowed to specify roll-up operations along any dimension.

For example, a user may want to roll up on the *item* dimension to go from viewing the data for particular TV sets that were purchased to viewing the brands of these TVs (e.g., SONY or Toshiba). Users may also navigate from the transaction level to the customer or customer-type level in the search for interesting associations. Such an OLAP data mining style is characteristic of multidimensional data mining. In our study of the principles of data mining in this book, we place particular emphasis on multidimensional data mining, that is, on the *integration of data mining and OLAP technology*.

4.4 Data Warehouse Implementation

Data warehouses contain huge volumes of data. OLAP servers demand that decision support queries be answered in the order of seconds. Therefore, it is crucial for data warehouse systems to support highly efficient cube computation techniques, access methods, and query processing techniques. In this section, we present an overview of methods for the efficient implementation of data warehouse systems. Section 4.4.1 explores how to compute data cubes efficiently. Section 4.4.2 shows how OLAP data can be indexed, using either bitmap or join indices. Next, we study how OLAP queries are processed (Section 4.4.3). Finally, Section 4.4.4 presents various types of warehouse servers for OLAP processing.

4.4.1 Efficient Data Cube Computation: An Overview

At the core of multidimensional data analysis is the efficient computation of aggregations across many sets of dimensions. In SQL terms, these aggregations are referred to as group-by's. Each group-by can be represented by a *cuboid*, where the set of group-by's forms a lattice of cuboids defining a data cube. In this subsection, we explore issues relating to the efficient computation of data cubes.

The compute cube Operator and the Curse of Dimensionality

One approach to cube computation extends SQL so as to include a compute cube operator. The compute cube operator computes aggregates over all subsets of the dimensions specified in the operation. This can require excessive storage space, especially for large numbers of dimensions. We start with an intuitive look at what is involved in the efficient computation of data cubes.

Example 4.6 **A data cube is a lattice of cuboids.** Suppose that you want to create a data cube for *AllElectronics* sales that contains the following: *city, item, year*, and *sales_in_dollars*. You want to be able to analyze the data, with queries such as the following:

- "*Compute the sum of sales, grouping by city and item.*"
- "*Compute the sum of sales, grouping by city.*"
- "*Compute the sum of sales, grouping by item.*"

What is the total number of cuboids, or group-by's, that can be computed for this data cube? Taking the three attributes, *city, item*, and *year*, as the dimensions for the data cube, and *sales_in_dollars* as the measure, the total number of cuboids, or group-by's, that can be computed for this data cube is $2^3 = 8$. The possible group-by's are the following: {(*city, item, year*), (*city, item*), (*city, year*), (*item, year*), (*city*), (*item*), (*year*), ()}, where () means that the group-by is empty (i.e., the dimensions are not grouped). These group-by's form a lattice of cuboids for the data cube, as shown in Figure 4.14.

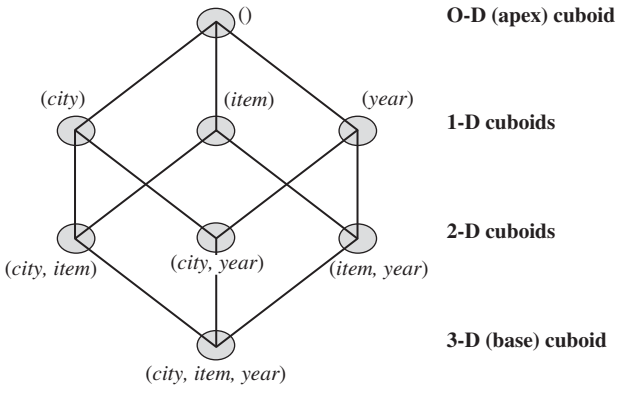


Figure 4.14 Lattice of cuboids, making up a 3-D data cube. Each cuboid represents a different group-by. The base cuboid contains *city, item*, and *year* dimensions.

The **base cuboid** contains all three dimensions, *city, item*, and *year*. It can return the total sales for any combination of the three dimensions. The **apex cuboid**, or 0-D cuboid, refers to the case where the group-by is empty. It contains the total sum of all sales. The base cuboid is the least generalized (most specific) of the cuboids. The apex cuboid is the most generalized (least specific) of the cuboids, and is often denoted as all. If we start at the apex cuboid and explore downward in the lattice, this is equivalent to drilling down within the data cube. If we start at the base cuboid and explore upward, this is akin to rolling up. ■

An SQL query containing no group-by (e.g., "*compute the sum of total sales*") is a *zero-dimensional operation*. An SQL query containing one group-by (e.g., "*compute the sum of sales, group-by city*") is a *one-dimensional operation*. A cube operator on n dimensions is equivalent to a collection of group-by statements, one for each subset of the n dimensions. Therefore, the cube operator is the n-dimensional generalization of the group-by operator.

Similar to the SQL syntax, the data cube in Example 4.1 could be defined as

```
define cube sales_cube [city, item, year]: sum(sales_in_dollars)
```

For a cube with n dimensions, there are a total of 2^n cuboids, including the base cuboid. A statement such as

```
compute cube sales_cube
```

would explicitly instruct the system to compute the sales aggregate cuboids for all eight subsets of the set {*city, item, year*}, including the empty subset. A cube computation operator was first proposed and studied by Gray et al. [GCB+97].

Online analytical processing may need to access different cuboids for different queries. Therefore, it may seem like a good idea to compute in advance all or at least some of the cuboids in a data cube. Precomputation leads to fast response time and avoids some redundant computation. Most, if not all, OLAP products resort to some degree of precomputation of multidimensional aggregates.

A major challenge related to this precomputation, however, is that the required storage space may explode if all the cuboids in a data cube are precomputed, especially when the cube has many dimensions. The storage requirements are even more excessive when many of the dimensions have associated concept hierarchies, each with multiple levels. This problem is referred to as the **curse of dimensionality**. The extent of the curse of dimensionality is illustrated here.

"*How many cuboids are there in an* n*-dimensional data cube?*" If there were no hierarchies associated with each dimension, then the total number of cuboids for an n-dimensional data cube, as we have seen, is 2^n. However, in practice, many dimensions do have hierarchies. For example, *time* is usually explored not at only one conceptual level (e.g., *year*), but rather at multiple conceptual levels such as in the hierarchy "*day* < *month* < *quarter* < *year*." For an n-dimensional data cube, the total number of cuboids

that can be generated (including the cuboids generated by climbing up the hierarchies along each dimension) is

$$Total\ number\ of\ cuboids = \prod_{i=1}^{n}(L_i + 1), \tag{4.1}$$

where L_i is the number of levels associated with dimension i. One is added to L_i in Eq. (4.1) to include the *virtual* top level, all. (Note that generalizing to all is equivalent to the removal of the dimension.)

This formula is based on the fact that, at most, one abstraction level in each dimension will appear in a cuboid. For example, the time dimension as specified before has four conceptual levels, or five if we include the virtual level all. If the cube has 10 dimensions and each dimension has five levels (including all), the total number of cuboids that can be generated is $5^{10} \approx 9.8 \times 10^6$. The size of each cuboid also depends on the *cardinality* (i.e., number of distinct values) of each dimension. For example, if the *AllElectronics* branch in each city sold every item, there would be $|city| \times |item|$ tuples in the *city_item* group-by alone. As the number of dimensions, number of conceptual hierarchies, or cardinality increases, the storage space required for many of the group-by's will grossly exceed the (fixed) size of the input relation.

By now, you probably realize that it is unrealistic to precompute and materialize all of the cuboids that can possibly be generated for a data cube (i.e., from a base cuboid). If there are many cuboids, and these cuboids are large in size, a more reasonable option is *partial materialization*; that is, to materialize only *some* of the possible cuboids that can be generated.

Partial Materialization: Selected Computation of Cuboids

There are three choices for data cube materialization given a base cuboid:

1. **No materialization**: Do not precompute any of the "nonbase" cuboids. This leads to computing expensive multidimensional aggregates on-the-fly, which can be extremely slow.
2. **Full materialization**: Precompute all of the cuboids. The resulting lattice of computed cuboids is referred to as the *full cube*. This choice typically requires huge amounts of memory space in order to store all of the precomputed cuboids.
3. **Partial materialization**: Selectively compute a proper subset of the whole set of possible cuboids. Alternatively, we may compute a subset of the cube, which contains only those cells that satisfy some user-specified criterion, such as where the tuple count of each cell is above some threshold. We will use the term *subcube* to refer to the latter case, where only some of the cells may be precomputed for various cuboids. Partial materialization represents an interesting trade-off between storage space and response time.

The partial materialization of cuboids or subcubes should consider three factors: (1) identify the subset of cuboids or subcubes to materialize; (2) exploit the materialized cuboids or subcubes during query processing; and (3) efficiently update the materialized cuboids or subcubes during load and refresh.

The selection of the subset of cuboids or subcubes to materialize should take into account the queries in the workload, their frequencies, and their accessing costs. In addition, it should consider workload characteristics, the cost for incremental updates, and the total storage requirements. The selection must also consider the broad context of physical database design such as the generation and selection of indices. Several OLAP products have adopted heuristic approaches for cuboid and subcube selection. A popular approach is to materialize the cuboids set on which other frequently referenced cuboids are based. Alternatively, we can compute an **iceberg cube**, which is a data cube that stores only those cube cells with an aggregate value (e.g., count) that is above some minimum support threshold.

Another common strategy is to materialize a *shell cube*. This involves precomputing the cuboids for only a small number of dimensions (e.g., three to five) of a data cube. Queries on additional combinations of the dimensions can be computed on-the-fly. Because our aim in this chapter is to provide a solid introduction and overview of data warehousing for data mining, we defer our detailed discussion of cuboid selection and computation to Chapter 5, which studies various data cube computation methods in greater depth.

Once the selected cuboids have been materialized, it is important to take advantage of them during query processing. This involves several issues, such as how to determine the relevant cuboid(s) from among the candidate materialized cuboids, how to use available index structures on the materialized cuboids, and how to transform the OLAP operations onto the selected cuboid(s). These issues are discussed in Section 4.4.3 as well as in Chapter 5.

Finally, during load and refresh, the materialized cuboids should be updated efficiently. Parallelism and incremental update techniques for this operation should be explored.

4.4.2 Indexing OLAP Data: Bitmap Index and Join Index

To facilitate efficient data accessing, most data warehouse systems support index structures and materialized views (using cuboids). General methods to select cuboids for materialization were discussed in Section 4.4.1. In this subsection, we examine how to index OLAP data by *bitmap indexing* and *join indexing*.

The **bitmap indexing** method is popular in OLAP products because it allows quick searching in data cubes. The bitmap index is an alternative representation of the *record_ID (RID)* list. In the bitmap index for a given attribute, there is a distinct bit vector, ***Bv***, for each value v in the attribute's domain. If a given attribute's domain consists of n values, then n bits are needed for each entry in the bitmap index (i.e., there are n bit vectors). If the attribute has the value v for a given row in the data table, then the bit representing that value is set to 1 in the corresponding row of the bitmap index. All other bits for that row are set to 0.

Example 4.7 **Bitmap indexing.** In the *AllElectronics* data warehouse, suppose the dimension *item* at the top level has four values (representing item types): "*home entertainment,*" "*computer,*" "*phone,*" and "*security.*" Each value (e.g., "*computer*") is represented by a bit vector in the *item* bitmap index table. Suppose that the cube is stored as a relation table with 100,000 rows. Because the domain of *item* consists of four values, the bitmap index table requires four bit vectors (or lists), each with 100,000 bits. Figure 4.15 shows a base (data) table containing the dimensions *item* and *city*, and its mapping to bitmap index tables for each of the dimensions. ■

Base table

RID	*item*	*city*
R1	H	V
R2	C	V
R3	P	V
R4	S	V
R5	H	T
R6	C	T
R7	P	T
R8	S	T

item bitmap index table

RID	H	C	P	S
R1	1	0	0	0
R2	0	1	0	0
R3	0	0	1	0
R4	0	0	0	1
R5	1	0	0	0
R6	0	1	0	0
R7	0	0	1	0
R8	0	0	0	1

city bitmap index table

RID	V	T
R1	1	0
R2	1	0
R3	1	0
R4	1	0
R5	0	1
R6	0	1
R7	0	1
R8	0	1

Note: H for "home entertainment," C for "computer," P for "phone," S for "security," V for "Vancouver," T for "Toronto."

Figure 4.15 Indexing OLAP data using bitmap indices.

Bitmap indexing is advantageous compared to hash and tree indices. It is especially useful for low-cardinality domains because comparison, join, and aggregation operations are then reduced to bit arithmetic, which substantially reduces the processing time. Bitmap indexing leads to significant reductions in space and input/output (I/O) since a string of characters can be represented by a single bit. For higher-cardinality domains, the method can be adapted using compression techniques.

The **join indexing** method gained popularity from its use in relational database query processing. Traditional indexing maps the value in a given column to a list of rows having that value. In contrast, join indexing registers the joinable rows of two relations from a relational database. For example, if two relations $R(RID, A)$ and $S(B, SID)$ join on the attributes A and B, then the join index record contains the pair (RID, SID), where RID and SID are record identifiers from the R and S relations, respectively. Hence, the join index records can identify joinable tuples without performing costly join operations. Join indexing is especially useful for maintaining the relationship between a foreign key[2] and its matching primary keys, from the joinable relation.

The star schema model of data warehouses makes join indexing attractive for cross-table search, because the linkage between a fact table and its corresponding dimension tables comprises the fact table's foreign key and the dimension table's primary key. Join

[2] A set of attributes in a relation schema that forms a primary key for another relation schema is called a **foreign key**.

indexing maintains relationships between attribute values of a dimension (e.g., within a dimension table) and the corresponding rows in the fact table. Join indices may span multiple dimensions to form **composite join indices**. We can use join indices to identify subcubes that are of interest.

Example 4.8 **Join indexing.** In Example 3.4, we defined a star schema for *AllElectronics* of the form "*sales_star* [*time, item, branch, location*]: *dollars_sold* = sum *(sales_in_dollars).*" An example of a join index relationship between the *sales* fact table and the *location* and *item* dimension tables is shown in Figure 4.16. For example, the "Main Street" value in the *location* dimension table joins with tuples T57, T238, and T884 of the *sales* fact table. Similarly, the "Sony-TV" value in the *item* dimension table joins with tuples T57 and T459 of the *sales* fact table. The corresponding join index tables are shown in Figure 4.17.

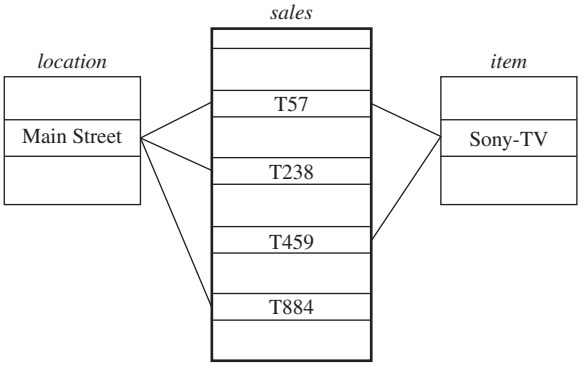


Figure 4.16 Linkages between a *sales* fact table and *location* and *item* dimension tables.

Join index table for *location/sales*

location	*sales_key*
. . .	. . .
Main Street	T57
Main Street	T238
Main Street	T884
. . .	. . .

Join index table for *item/sales*

item	*sales_key*
. . .	. . .
Sony-TV	T57
Sony-TV	T459
. . .	. . .

Join index table linking *location* and *item* to *sales*

location	*item*	*sales_key*
. . .	. . .	. . .
Main Street	Sony-TV	T57
. . .	. . .	. . .

Figure 4.17 Join index tables based on the linkages between the *sales* fact table and the *location* and *item* dimension tables shown in Figure 4.16.

Suppose that there are 360 time values, 100 items, 50 branches, 30 locations, and 10 million sales tuples in the *sales_star* data cube. If the *sales* fact table has recorded sales for only 30 items, the remaining 70 items will obviously not participate in joins. If join indices are not used, additional I/Os have to be performed to bring the joining portions of the fact table and the dimension tables together. ■

To further speed up query processing, the join indexing and the bitmap indexing methods can be integrated to form **bitmapped join indices**.

4.4.3 Efficient Processing of OLAP Queries

The purpose of materializing cuboids and constructing OLAP index structures is to speed up query processing in data cubes. Given materialized views, query processing should proceed as follows:

1. **Determine which operations should be performed on the available cuboids:** This involves transforming any selection, projection, roll-up (group-by), and drill-down operations specified in the query into corresponding SQL and/or OLAP operations. For example, slicing and dicing a data cube may correspond to selection and/or projection operations on a materialized cuboid.
2. **Determine to which materialized cuboid(s) the relevant operations should be applied:** This involves identifying all of the materialized cuboids that may potentially be used to answer the query, pruning the set using knowledge of "dominance" relationships among the cuboids, estimating the costs of using the remaining materialized cuboids, and selecting the cuboid with the least cost.

Example 4.9 **OLAP query processing.** Suppose that we define a data cube for *AllElectronics* of the form "*sales_cube* [*time, item, location*]: sum*(sales_in_dollars)*." The dimension hierarchies used are "*day* < *month* < *quarter* < *year*" for *time*; "*item_name* < *brand* < *type*" for *item*; and "*street* < *city* < *province_or_state* < *country*" for *location*.

Suppose that the query to be processed is on {*brand, province_or_state*}, with the selection constant "*year = 2010*." Also, suppose that there are four materialized cuboids available, as follows:

- cuboid 1: {*year, item_name, city*}
- cuboid 2: {*year, brand, country*}
- cuboid 3: {*year, brand, province_or_state*}
- cuboid 4: {*item_name, province_or_state*}, where *year* = 2010

"*Which of these four cuboids should be selected to process the query?*" Finer-granularity data cannot be generated from coarser-granularity data. Therefore, cuboid 2 cannot be used because *country* is a more general concept than *province_or_state*. Cuboids 1, 3, and 4 can be used to process the query because (1) they have the same set or a superset of the

dimensions in the query, (2) the selection clause in the query can imply the selection in the cuboid, and (3) the abstraction levels for the *item* and *location* dimensions in these cuboids are at a finer level than *brand* and *province_or_state*, respectively.

"*How would the costs of each cuboid compare if used to process the query?*" It is likely that using cuboid 1 would cost the most because both *item_name* and *city* are at a lower level than the *brand* and *province_or_state* concepts specified in the query. If there are not many *year* values associated with *items* in the cube, but there are several *item_names* for each *brand*, then cuboid 3 will be smaller than cuboid 4, and thus cuboid 3 should be chosen to process the query. However, if efficient indices are available for cuboid 4, then cuboid 4 may be a better choice. Therefore, some cost-based estimation is required to decide which set of cuboids should be selected for query processing. ■

4.4.4 OLAP Server Architectures: ROLAP versus MOLAP versus HOLAP

Logically, OLAP servers present business users with multidimensional data from data warehouses or data marts, without concerns regarding how or where the data are stored. However, the physical architecture and implementation of OLAP servers must consider data storage issues. Implementations of a warehouse server for OLAP processing include the following:

Relational OLAP (ROLAP) servers: These are the intermediate servers that stand in between a relational back-end server and client front-end tools. They use a *relational* or *extended-relational DBMS* to store and manage warehouse data, and OLAP middleware to support missing pieces. ROLAP servers include optimization for each DBMS back end, implementation of aggregation navigation logic, and additional tools and services. ROLAP technology tends to have greater scalability than MOLAP technology. The DSS server of Microstrategy, for example, adopts the ROLAP approach.

Multidimensional OLAP (MOLAP) servers: These servers support multidimensional data views through *array-based multidimensional storage engines*. They map multidimensional views directly to data cube array structures. The advantage of using a data cube is that it allows fast indexing to precomputed summarized data. Notice that with multidimensional data stores, the storage utilization may be low if the data set is sparse. In such cases, sparse matrix compression techniques should be explored (Chapter 5).

Many MOLAP servers adopt a two-level storage representation to handle dense and sparse data sets: Denser subcubes are identified and stored as array structures, whereas sparse subcubes employ compression technology for efficient storage utilization.

Hybrid OLAP (HOLAP) servers: The hybrid OLAP approach combines ROLAP and MOLAP technology, benefiting from the greater scalability of ROLAP and the faster computation of MOLAP. For example, a HOLAP server may allow large volumes

of detailed data to be stored in a relational database, while aggregations are kept in a separate MOLAP store. The Microsoft SQL Server 2000 supports a hybrid OLAP server.

Specialized SQL servers: To meet the growing demand of OLAP processing in relational databases, some database system vendors implement specialized SQL servers that provide advanced query language and query processing support for SQL queries over star and snowflake schemas in a read-only environment.

"*How are data actually stored in ROLAP and MOLAP architectures?*" Let's first look at ROLAP. As its name implies, ROLAP uses relational tables to store data for online analytical processing. Recall that the fact table associated with a base cuboid is referred to as a *base fact table*. The base fact table stores data at the abstraction level indicated by the join keys in the schema for the given data cube. Aggregated data can also be stored in fact tables, referred to as **summary fact tables**. Some summary fact tables store both base fact table data and aggregated data (see Example 3.10). Alternatively, separate summary fact tables can be used for each abstraction level to store only aggregated data.

Example 4.10 **A ROLAP data store.** Table 4.4 shows a summary fact table that contains both base fact data and aggregated data. The schema is "⟨*record_identifier (RID), item, . . . , day, month, quarter, year, dollars_sold*⟩," where *day, month, quarter*, and *year* define the sales date, and *dollars_sold* is the sales amount. Consider the tuples with an *RID* of 1001 and 1002, respectively. The data of these tuples are at the base fact level, where the sales dates are October 15, 2010, and October 23, 2010, respectively. Consider the tuple with an *RID* of 5001. This tuple is at a more general level of abstraction than the tuples 1001 and 1002. The *day* value has been generalized to all, so that the corresponding *time* value is October 2010. That is, the *dollars_sold* amount shown is an aggregation representing the entire month of October 2010, rather than just October 15 or 23, 2010. The special value all is used to represent subtotals in summarized data. ■

MOLAP uses multidimensional array structures to store data for online analytical processing. This structure is discussed in greater detail in Chapter 5.

Most data warehouse systems adopt a client-server architecture. A relational data store always resides at the data warehouse/data mart server site. A multidimensional data store can reside at either the database server site or the client site.

Table 4.4 Single Table for Base and Summary Facts

RID	*item*	. . .	*day*	*month*	*quarter*	*year*	*dollars_sold*
1001	TV	. . .	15	10	Q4	2010	250.60
1002	TV	. . .	23	10	Q4	2010	175.00
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
5001	TV	. . .	all	10	Q4	2010	45,786.08
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .

4.5 Data Generalization by Attribute-Oriented Induction

Conceptually, the data cube can be viewed as a kind of multidimensional data generalization. In general, *data generalization* summarizes data by replacing relatively low-level values (e.g., numeric values for an attribute *age*) with higher-level concepts (e.g., *young*, *middle-aged*, and *senior*), or by reducing the number of dimensions to summarize data in concept space involving fewer dimensions (e.g., removing *birth_date* and *telephone number* when summarizing the behavior of a group of students). Given the large amount of data stored in databases, it is useful to be able to describe concepts in concise and succinct terms at generalized (rather than low) levels of abstraction. Allowing data sets to be generalized at multiple levels of abstraction facilitates users in examining the general behavior of the data. Given the *AllElectronics* database, for example, instead of examining individual customer transactions, sales managers may prefer to view the data generalized to higher levels, such as summarized by customer groups according to geographic regions, frequency of purchases per group, and customer income.

This leads us to the notion of *concept description*, which is a form of data generalization. A concept typically refers to a data collection such as *frequent_buyers, graduate_students*, and so on. As a data mining task, concept description is not a simple enumeration of the data. Instead, **concept description** generates descriptions for data *characterization* and *comparison*. It is sometimes called **class description** when the concept to be described refers to a class of objects. **Characterization** provides a concise and succinct summarization of the given data collection, while concept or class **comparison** (also known as **discrimination**) provides descriptions comparing two or more data collections.

Up to this point, we have studied data cube (or OLAP) approaches to concept description using multidimensional, multilevel data generalization in data warehouses. "*Is data cube technology sufficient to accomplish all kinds of concept description tasks for large data sets?*" Consider the following cases.

- **Complex data types and aggregation**: Data warehouses and OLAP tools are based on a multidimensional data model that views data in the form of a data cube, consisting of dimensions (or attributes) and measures (aggregate functions). However, many current OLAP systems confine dimensions to non-numeric data and measures to numeric data. In reality, the database can include attributes of various data types, including numeric, non-numeric, spatial, text, or image, which ideally should be included in the concept description.

  Furthermore, the aggregation of attributes in a database may include sophisticated data types such as the collection of non-numeric data, the merging of spatial regions, the composition of images, the integration of texts, and the grouping of object pointers. Therefore, OLAP, with its restrictions on the possible dimension and measure types, represents a simplified model for data analysis. Concept description should handle complex data types of the attributes and their aggregations, as necessary.

- **User control versus automation**: Online analytical processing in data warehouses is a user-controlled process. The selection of dimensions and the application of OLAP operations (e.g., drill-down, roll-up, slicing, and dicing) are primarily directed and controlled by users. Although the control in most OLAP systems is quite user-friendly, users do require a good understanding of the role of each dimension. Furthermore, in order to find a satisfactory description of the data, users may need to specify a long sequence of OLAP operations. It is often desirable to have a more automated process that helps users determine which dimensions (or attributes) should be included in the analysis, and the degree to which the given data set should be generalized in order to produce an interesting summarization of the data.

This section presents an alternative method for concept description, called *attribute-oriented induction*, which works for complex data types and relies on a data-driven generalization process.

4.5.1 Attribute-Oriented Induction for Data Characterization

The **attribute-oriented induction** (**AOI**) approach to concept description was first proposed in 1989, a few years before the introduction of the data cube approach. The data cube approach is essentially based on *materialized views* of the data, which typically have been precomputed in a data warehouse. In general, it performs offline aggregation before an OLAP or data mining query is submitted for processing. On the other hand, the attribute-oriented induction approach is basically a *query-oriented*, generalization-based, online data analysis technique. Note that there is no inherent barrier distinguishing the two approaches based on online aggregation versus offline precomputation. Some aggregations in the data cube can be computed online, while offline precomputation of multidimensional space can speed up attribute-oriented induction as well.

The general idea of attribute-oriented induction is to first collect the task-relevant data using a database query and then perform generalization based on the examination of the number of each attribute's distinct values in the relevant data set. The generalization is performed by either *attribute removal* or *attribute generalization*. Aggregation is performed by merging identical generalized tuples and accumulating their respective counts. This reduces the size of the generalized data set. The resulting generalized relation can be mapped into different forms (e.g., charts or rules) for presentation to the user.

The following illustrates the process of attribute-oriented induction. We first discuss its use for characterization. The method is extended for the mining of class comparisons in Section 4.5.3.

Example 4.11 **A data mining query for characterization.** Suppose that a user wants to describe the general characteristics of graduate students in the *Big University* database, given the attributes *name, gender, major, birth_place, birth_date, residence, phone# (telephone*

number), and *gpa (grade_point_average)*. A data mining query for this characterization can be expressed in the data mining query language, DMQL, as follows:

```
use Big_University_DB
mine characteristics as "Science_Students"
in relevance to name, gender, major, birth_place, birth_date, residence,
        phone#, gpa
from student
where status in "graduate"
```

We will see how this example of a typical data mining query can apply attribute-oriented induction to the mining of characteristic descriptions.

First, **data focusing** should be performed *before* attribute-oriented induction. This step corresponds to the specification of the task-relevant data (i.e., data for analysis). The data are collected based on the information provided in the data mining query. Because a data mining query is usually relevant to only a portion of the database, selecting the relevant data set not only makes mining more efficient, but also derives more meaningful results than mining the entire database.

Specifying the set of relevant attributes (i.e., attributes for mining, as indicated in DMQL with the in relevance to clause) may be difficult for the user. A user may select only a few attributes that he or she feels are important, while missing others that could also play a role in the description. For example, suppose that the dimension *birth_place* is defined by the attributes *city*, *province_or_state*, and *country*. Of these attributes, let's say that the user has only thought to specify *city*. In order to allow generalization on the *birth_place* dimension, the other attributes defining this dimension should also be included. In other words, having the system automatically include *province_or_state* and *country* as relevant attributes allows *city* to be generalized to these higher conceptual levels during the induction process.

At the other extreme, suppose that the user may have introduced too many attributes by specifying all of the possible attributes with the clause in relevance to $*$. In this case, all of the attributes in the relation specified by the from clause would be included in the analysis. Many of these attributes are unlikely to contribute to an interesting description. A correlation-based analysis method (Section 3.3.2) can be used to perform attribute *relevance analysis* and filter out statistically irrelevant or weakly relevant attributes from the descriptive mining process. Other approaches such as attribute subset selection, are also described in Chapter 3.

Table 4.5 Initial Working Relation: A Collection of Task-Relevant Data

name	*gender*	*major*	*birth_place*	*birth_date*	*residence*	*phone#*	*gpa*
Jim Woodman	M	CS	Vancouver, BC, Canada	12-8-76	3511 Main St., Richmond	687-4598	3.67
Scott Lachance	M	CS	Montreal, Que, Canada	7-28-75	345 1st Ave., Richmond	253-9106	3.70
Laura Lee	F	Physics	Seattle, WA, USA	8-25-70	125 Austin Ave., Burnaby	420-5232	3.83
...	...	...	...	...	...	...	...

"*What does the* 'where *status* in "graduate"' *clause mean?*" The where clause implies that a concept hierarchy exists for the attribute *status*. Such a concept hierarchy organizes primitive-level data values for *status* (e.g., "M.Sc.," "M.A.," "M.B.A.," "Ph.D.," "B.Sc.," and "B.A.") into higher conceptual levels (e.g., "graduate" and "undergraduate"). This use of concept hierarchies does not appear in traditional relational query languages, yet is likely to become a common feature in data mining query languages.

The data mining query presented in Example 4.11 is transformed into the following relational query for the collection of the task-relevant data set:

```
use Big_University_DB
select name, gender, major, birth_place, birth_date, residence, phone#, gpa
from student
where status in {"M.Sc.," "M.A.," "M.B.A.," "Ph.D."}
```

The transformed query is executed against the relational database, *Big_University_DB*, and returns the data shown earlier in Table 4.5. This table is called the (task-relevant) **initial working relation**. It is the data on which induction will be performed. Note that each tuple is, in fact, a conjunction of attribute–value pairs. Hence, we can think of a tuple within a relation as a rule of conjuncts, and of induction on the relation as the generalization of these rules. ■

"*Now that the data are ready for attribute-oriented induction, how is attribute-oriented induction performed?*" The essential operation of attribute-oriented induction is *data generalization*, which can be performed in either of two ways on the initial working relation: *attribute removal* and *attribute generalization*.

Attribute removal is based on the following rule: *If there is a large set of distinct values for an attribute of the initial working relation, but either (case 1) there is no generalization operator on the attribute (e.g., there is no concept hierarchy defined for the attribute), or (case 2) its higher-level concepts are expressed in terms of other attributes, then the attribute should be removed from the working relation.*

Let's examine the reasoning behind this rule. An attribute–value pair represents a conjunct in a generalized tuple, or rule. The removal of a conjunct eliminates a constraint and thus generalizes the rule. If, as in case 1, there is a large set of distinct values for an attribute but there is no generalization operator for it, the attribute should be removed because it cannot be generalized. Preserving it would imply keeping a large number of disjuncts, which contradicts the goal of generating concise rules. On the other hand, consider case 2, where the attribute's higher-level concepts are expressed in terms of other attributes. For example, suppose that the attribute in question is *street*, with higher-level concepts that are represented by the attributes ⟨*city, province_or_state, country*⟩. The removal of *street* is equivalent to the application of a generalization operator. This rule corresponds to the generalization rule known as *dropping condition* in the machine learning literature on *learning from examples*.

Attribute generalization is based on the following rule: *If there is a large set of distinct values for an attribute in the initial working relation, and there exists a set of generalization operators on the attribute, then a generalization operator should be selected and applied*

to the attribute. This rule is based on the following reasoning. Use of a generalization operator to generalize an attribute value within a tuple, or rule, in the working relation will make the rule cover more of the original data tuples, thus generalizing the concept it represents. This corresponds to the generalization rule known as *climbing generalization trees* in *learning from examples*, or *concept tree ascension*.

Both rules–*attribute removal* and *attribute generalization*–claim that if there is a *large* set of distinct values for an attribute, further generalization should be applied. This raises the question: How large is "*a large set of distinct values for an attribute*" considered to be?

Depending on the attributes or application involved, a user may prefer some attributes to remain at a rather low abstraction level while others are generalized to higher levels. The control of how high an attribute should be generalized is typically quite subjective. The control of this process is called **attribute generalization control**. If the attribute is generalized "too high," it may lead to overgeneralization, and the resulting rules may not be very informative.

On the other hand, if the attribute is not generalized to a "sufficiently high level," then undergeneralization may result, where the rules obtained may not be informative either. Thus, a balance should be attained in attribute-oriented generalization. There are many possible ways to control a generalization process. We will describe two common approaches and illustrate how they work.

The first technique, called **attribute generalization threshold control**, either sets one generalization threshold for all of the attributes, or sets one threshold for each attribute. If the number of distinct values in an attribute is greater than the attribute threshold, further attribute removal or attribute generalization should be performed. Data mining systems typically have a default attribute threshold value generally ranging from 2 to 8 and should allow experts and users to modify the threshold values as well. If a user feels that the generalization reaches too high a level for a particular attribute, the threshold can be increased. This corresponds to drilling down along the attribute. Also, to further generalize a relation, the user can reduce an attribute's threshold, which corresponds to rolling up along the attribute.

The second technique, called **generalized relation threshold control**, sets a threshold for the generalized relation. If the number of (distinct) tuples in the generalized relation is greater than the threshold, further generalization should be performed. Otherwise, no further generalization should be performed. Such a threshold may also be preset in the data mining system (usually within a range of 10 to 30), or set by an expert or user, and should be adjustable. For example, if a user feels that the generalized relation is too small, he or she can increase the threshold, which implies drilling down. Otherwise, to further generalize a relation, the threshold can be reduced, which implies rolling up.

These two techniques can be applied in sequence: First apply the attribute threshold control technique to generalize each attribute, and then apply relation threshold control to further reduce the size of the generalized relation. No matter which generalization control technique is applied, the user should be allowed to adjust the generalization thresholds in order to obtain interesting concept descriptions.

In many database-oriented induction processes, users are interested in obtaining quantitative or statistical information about the data at different abstraction levels.

Thus, it is important to accumulate count and other aggregate values in the induction process. Conceptually, this is performed as follows. The aggregate function, count(), is associated with each database tuple. Its value for each tuple in the initial working relation is initialized to 1. Through attribute removal and attribute generalization, tuples within the initial working relation may be generalized, resulting in groups of *identical tuples*. In this case, all of the identical tuples forming a group should be merged into one tuple.

The count of this new, generalized tuple is set to the total number of tuples from the initial working relation that are represented by (i.e., merged into) the new generalized tuple. For example, suppose that by attribute-oriented induction, 52 data tuples from the initial working relation are all generalized to the same tuple, T. That is, the generalization of these 52 tuples resulted in 52 identical instances of tuple T. These 52 identical tuples are merged to form one instance of T, with a count that is set to 52. Other popular aggregate functions that could also be associated with each tuple include sum() and avg(). For a given generalized tuple, sum() contains the sum of the values of a given numeric attribute for the initial working relation tuples making up the generalized tuple. Suppose that tuple T contained sum(*units_sold*) as an aggregate function. The sum value for tuple T would then be set to the total number of units sold for each of the 52 tuples. The aggregate avg() (average) is computed according to the formula avg() = sum()/count().

Example 4.12 **Attribute-oriented induction.** Here we show how attribute-oriented induction is performed on the initial working relation of Table 4.5. For each attribute of the relation, the generalization proceeds as follows:

1. *name:* Since there are a large number of distinct values for *name* and there is no generalization operation defined on it, this attribute is removed.
2. *gender:* Since there are only two distinct values for *gender*, this attribute is retained and no generalization is performed on it.
3. *major:* Suppose that a concept hierarchy has been defined that allows the attribute *major* to be generalized to the values {arts&sciences, engineering, business}. Suppose also that the attribute generalization threshold is set to 5, and that there are more than 20 distinct values for *major* in the initial working relation. By attribute generalization and attribute generalization control, *major* is therefore generalized by climbing the given concept hierarchy.
4. *birth_place:* This attribute has a large number of distinct values; therefore, we would like to generalize it. Suppose that a concept hierarchy exists for *birth_place*, defined as "*city* < *province_or_state* < *country*." If the number of distinct values for *country* in the initial working relation is greater than the attribute generalization threshold, then *birth_place* should be removed, because even though a generalization operator exists for it, the generalization threshold would not be satisfied. If, instead, the number of distinct values for *country* is less than the attribute generalization threshold, then *birth_place* should be generalized to *birth_country*.
5. *birth_date:* Suppose that a hierarchy exists that can generalize *birth_date* to *age* and *age* to *age_range*, and that the number of age ranges (or intervals) is small with

Table 4.6 Generalized Relation Obtained by Attribute-Oriented Induction on Table 4.5's Data

gender	*major*	*birth_country*	*age_range*	*residence_city*	*gpa*	count
M	Science	Canada	20 – 25	Richmond	very_good	16
F	Science	Foreign	25 – 30	Burnaby	excellent	22
...	...	...	...	...	...	...

respect to the attribute generalization threshold. Generalization of *birth_date* should therefore take place.

6. *residence:* Suppose that *residence* is defined by the attributes *number, street, residence_city, residence_province_or_state*, and *residence_country*. The number of distinct values for *number* and *street* will likely be very high, since these concepts are quite low level. The attributes *number* and *street* should therefore be removed so that *residence* is then generalized to *residence_city*, which contains fewer distinct values.
7. *phone#:* As with the *name* attribute, *phone#* contains too many distinct values and should therefore be removed in generalization.
8. *gpa:* Suppose that a concept hierarchy exists for *gpa* that groups values for grade point average into numeric intervals like {3.75–4.0, 3.5–3.75, ...}, which in turn are grouped into descriptive values such as {"excellent", "very_good", ...}. The attribute can therefore be generalized.

The generalization process will result in groups of identical tuples. For example, the first two tuples of Table 4.5 both generalize to the same identical tuple (namely, the first tuple shown in Table 4.6). Such identical tuples are then merged into one, with their counts accumulated. This process leads to the generalized relation shown in Table 4.6.

Based on the vocabulary used in OLAP, we may view count() as a *measure*, and the remaining attributes as *dimensions*. Note that aggregate functions, such as sum(), may be applied to numeric attributes (e.g., *salary* and *sales*). These attributes are referred to as *measure attributes*. ■

4.5.2 Efficient Implementation of Attribute-Oriented Induction

"How is attribute-oriented induction actually implemented?" Section 4.5.1 provided an introduction to attribute-oriented induction. The general procedure is summarized in Figure 4.18. The efficiency of this algorithm is analyzed as follows:

- Step 1 of the algorithm is essentially a relational query to collect the task-relevant data into the **working relation**, W. Its processing efficiency depends on the query processing methods used. Given the successful implementation and commercialization of database systems, this step is expected to have good performance.
- Step 2 collects statistics on the working relation. This requires scanning the relation at most once. The cost for computing the minimum desired level and determining the mapping pairs, (v, v'), for each attribute is dependent on the number of distinct

Algorithm: Attribute-oriented induction. Mining generalized characteristics in a relational database given a user's data mining request.

Input:

- *DB*, a relational database;
- *DMQuery*, a data mining query;
- *a_list*, a list of attributes (containing attributes, a_i);
- *Gen*(a_i), a set of concept hierarchies or generalization operators on attributes, a_i;
- *a_gen_thresh*(a_i), attribute generalization thresholds for each a_i.

Output: *P*, a *Prime_generalized_relation.*

Method:

1. $W \leftarrow$ get_task_relevant_data (*DMQuery, DB*); // Let W, the working relation, hold the task-relevant data.
2. prepare_for_generalization (W); // This is implemented as follows.
   - **(a)** Scan W and collect the distinct values for each attribute, a_i. (*Note:* If W is very large, this may be done by examining a sample of W.)
   - **(b)** For each attribute a_i, determine whether a_i should be removed. If not, compute its minimum desired level L_i based on its given or default attribute threshold, and determine the mapping pairs (v, v'), where v is a distinct value of a_i in W, and v' is its corresponding generalized value at level L_i.
3. $P \leftarrow$ generalization (W),

   The *Prime_generalized_relation*, P, is derived by replacing each value v in W by its corresponding v' in the mapping while accumulating `count` and computing any other aggregate values.

   This step can be implemented efficiently using either of the two following variations:

   - **(a)** For each generalized tuple, insert the tuple into a sorted prime relation P by a binary search: if the tuple is already in P, simply increase its `count` and other aggregate values accordingly; otherwise, insert it into P.
   - **(b)** Since in most cases the number of distinct values at the prime relation level is small, the prime relation can be coded as an m-dimensional array, where m is the number of attributes in P, and each dimension contains the corresponding generalized attribute values. Each array element holds the corresponding `count` and other aggregation values, if any. The insertion of a generalized tuple is performed by measure aggregation in the corresponding array element.

Figure 4.18 Basic algorithm for attribute-oriented induction.

values for each attribute and is smaller than $|W|$, the number of tuples in the working relation. Notice that it may not be necessary to scan the working relation once, since if the working relation is large, a sample of such a relation will be sufficient to get statistics and determine which attributes should be generalized to a certain high level and which attributes should be removed. Moreover, such statistics may also be obtained in the process of extracting and generating a working relation in Step 1.

- Step 3 derives the **prime relation**, P. This is performed by scanning each tuple in the working relation and inserting generalized tuples into P. There are a total of $|W|$ tuples in W and p tuples in P. For each tuple, $\boldsymbol{t}$, in W, we substitute its attribute values based on the derived mapping pairs. This results in a generalized tuple, $\boldsymbol{t'}$. If variation (a) in Figure 4.18 is adopted, each $\boldsymbol{t'}$ takes $O(\log p)$ to find the location for the count increment or tuple insertion. Thus, the total time complexity is $O(|W| \times \log p)$ for all of the generalized tuples. If variation (b) is adopted, each $\boldsymbol{t'}$ takes $O(1)$ to find the tuple for the count increment. Thus, the overall time complexity is $O(N)$ for all of the generalized tuples.

Many data analysis tasks need to examine a good number of dimensions or attributes. This may involve *dynamically* introducing and testing additional attributes rather than just those specified in the mining query. Moreover, a user with little knowledge of the *truly* relevant data set may simply specify "in relevance to $*$" in the mining query, which includes all of the attributes in the analysis. Therefore, an advanced–concept description mining process needs to perform attribute relevance analysis on large sets of attributes to select the most relevant ones. This analysis may employ correlation measures or tests of statistical significance, as described in Chapter 3 on data preprocessing.

Example 4.13 **Presentation of generalization results.** Suppose that attribute-oriented induction was performed on a *sales* relation of the *AllElectronics* database, resulting in the generalized description of Table 4.7 for sales last year. The description is shown in the form of a generalized relation. Table 4.6 is another generalized relation example.

Such generalized relations can also be presented in the form of cross-tabulation forms, various kinds of graphic presentation (e.g., pie charts and bar charts), and quantitative characteristics rules (i.e., showing how different value combinations are distributed in the generalized relation). ■

Table 4.7 Generalized Relation for Last Year's Sales

location	*item*	*sales (in million dollars)*	count *(in thousands)*
Asia	TV	15	300
Europe	TV	12	250
North America	TV	28	450
Asia	computer	120	1000
Europe	computer	150	1200
North America	computer	200	1800

4.5.3 Attribute-Oriented Induction for Class Comparisons

In many applications, users may not be interested in having a single class (or concept) described or characterized, but prefer to mine a description that compares or distinguishes one class (or concept) from other comparable classes (or concepts). Class discrimination or comparison (hereafter referred to as **class comparison**) mines descriptions that distinguish a target class from its contrasting classes. Notice that the target and contrasting classes must be *comparable* in the sense that they share similar dimensions and attributes. For example, the three classes *person, address*, and *item* are not comparable. However, sales in the last three years are comparable classes, and so are, for example, computer science students versus physics students.

Our discussions on class characterization in the previous sections handle multilevel data summarization and characterization in a single class. The techniques developed can be extended to handle class comparison across several comparable classes. For example, the attribute generalization process described for class characterization can be modified so that the generalization is performed *synchronously* among all the classes compared. This allows the attributes in all of the classes to be generalized to the *same* abstraction levels.

Suppose, for instance, that we are given the *AllElectronics* data for sales in 2009 and in 2010 and want to compare these two classes. Consider the dimension *location* with abstractions at the *city*, *province_or_state*, and *country* levels. Data in each class should be generalized to the same *location* level. That is, they are all synchronously generalized to either the *city* level, the *province_or_state* level, or the *country* level. Ideally, this is more useful than comparing, say, the sales in Vancouver in 2009 with the sales in the United States in 2010 (i.e., where each set of sales data is generalized to a different level). The users, however, should have the option to overwrite such an automated, synchronous comparison with their own choices, when preferred.

"How is class comparison performed?" In general, the procedure is as follows:

1. **Data collection**: The set of relevant data in the database is collected by query processing and is partitioned respectively into a *target class* and one or a set of *contrasting classes*.
2. **Dimension relevance analysis**: If there are many dimensions, then dimension relevance analysis should be performed on these classes to select only the highly relevant dimensions for further analysis. Correlation or entropy-based measures can be used for this step (Chapter 3).
3. **Synchronous generalization**: Generalization is performed on the target class to the level controlled by a user- or expert-specified dimension threshold, which results in a **prime target class relation**. The concepts in the contrasting class(es) are generalized to the same level as those in the prime target class relation, forming the **prime contrasting class(es) relation**.
4. **Presentation of the derived comparison**: The resulting class comparison description can be visualized in the form of tables, graphs, and rules. This presentation usually includes a "contrasting" measure such as count% (percentage count) that reflects the

comparison between the target and contrasting classes. The user can adjust the comparison description by applying drill-down, roll-up, and other OLAP operations to the target and contrasting classes, as desired.

The preceding discussion outlines a general algorithm for mining comparisons in databases. In comparison with characterization, the previous algorithm involves synchronous generalization of the target class with the contrasting classes, so that classes are simultaneously compared at the same abstraction levels.

Example 4.14 mines a class comparison describing the graduate and undergraduate students at *Big University*.

Example 4.14 Mining a class comparison. Suppose that you would like to compare the general properties of the graduate and undergraduate students at *Big_University*, given the attributes *name, gender, major, birth_place, birth_date, residence, phone#*, and *gpa*.

This data mining task can be expressed in DMQL as follows:

```
use Big_University_DB
mine comparison as "grad_vs_undergrad_students"
in relevance to name, gender, major, birth_place, birth_date, residence,
        phone#, gpa
for "graduate_students"
where status in "graduate"
versus "undergraduate_students"
where status in "undergraduate"
analyze count%
from student
```

Let's see how this typical example of a data mining query for mining comparison descriptions can be processed.

First, the query is transformed into two relational queries that collect two sets of task-relevant data: one for the *initial target-class working relation* and the other for the *initial contrasting-class working relation*, as shown in Tables 4.8 and 4.9. This can also be viewed as the construction of a data cube, where the status {graduate, undergraduate} serves as one dimension, and the other attributes form the remaining dimensions.

Second, dimension relevance analysis can be performed, when necessary, on the two classes of data. After this analysis, irrelevant or weakly relevant dimensions (e.g., *name, gender, birth_place, residence*, and *phone#*) are removed from the resulting classes. Only the highly relevant attributes are included in the subsequent analysis.

Third, synchronous generalization is performed on the target class to the levels controlled by user- or expert-specified dimension thresholds, forming the *prime target class relation*. The contrasting class is generalized to the same levels as those in the prime target class relation, forming the *prime contrasting class(es) relation*, as presented in Tables 4.10 and 4.11. In comparison with undergraduate students, graduate students tend to be older and have a higher GPA in general.

Table 4.8 Initial Working Relations: The Target Class (Graduate Students)

name	*gender*	*major*	*birth_place*	*birth_date*	*residence*	*phone#*	*gpa*
Jim Woodman	M	CS	Vancouver, BC, Canada	12-8-76	3511 Main St., Richmond	687-4598	3.67
Scott Lachance	M	CS	Montreal, Que, Canada	7-28-75	345 1st Ave., Vancouver	253-9106	3.70
Laura Lee	F	Physics	Seattle, WA, USA	8-25-70	125 Austin Ave., Burnaby	420-5232	3.83
...	...	...	...	...	...	...	...

Table 4.9 Initial Working Relations: The Contrasting Class (Undergraduate Students)

name	*gender*	*major*	*birth_place*	*birth_date*	*residence*	*phone#*	*gpa*
Bob Schumann	M	Chemistry	Calgary, Alt, Canada	1-10-78	2642 Halifax St., Burnaby	294-4291	2.96
Amy Eau	F	Biology	Golden, BC, Canada	3-30-76	463 Sunset Cres., Vancouver	681-5417	3.52
...	...	...	...	...	...	...	...

Table 4.10 Prime Generalized Relation for the Target Class (Graduate Students)

major	*age_range*	*gpa*	count%
Science	21...25	good	5.53
Science	26...30	good	5.02
Science	over_30	very good	5.86
...	...	...	...
Business	over_30	excellent	4.68

Table 4.11 Prime Generalized Relation for the Contrasting Class (Undergraduate Students)

major	*age_range*	*gpa*	count%
Science	16...20	fair	5.53
Science	16...20	good	4.53
...	...	...	...
Science	26...30	good	2.32
...	...	...	...
Business	over_30	excellent	0.68

Finally, the resulting class comparison is presented in the form of tables, graphs, and/or rules. This visualization includes a contrasting measure (e.g., count%) that compares the target class and the contrasting class. For example, 5.02% of the graduate students majoring in science are between 26 and 30 years old and have a "good" GPA, while only 2.32% of undergraduates have these same characteristics. Drilling and other

OLAP operations may be performed on the target and contrasting classes as deemed necessary by the user in order to adjust the abstraction levels of the final description. ■

In summary, attribute-oriented induction for data characterization and generalization provides an alternative data generalization method in comparison to the data cube approach. It is not confined to relational data because such an induction can be performed on spatial, multimedia, sequence, and other kinds of data sets. In addition, there is no need to precompute a data cube because generalization can be performed online upon receiving a user's query.

Moreover, automated analysis can be added to such an induction process to automatically filter out irrelevant or unimportant attributes. However, because attribute-oriented induction automatically generalizes data to a higher level, it cannot efficiently support the process of drilling down to levels deeper than those provided in the generalized relation. The integration of data cube technology with attribute-oriented induction may provide a balance between precomputation and online computation. This would also support fast online computation when it is necessary to drill down to a level deeper than that provided in the generalized relation.

4.6 Summary

- A **data warehouse** is a *subject-oriented, integrated, time-variant*, and *nonvolatile* data collection organized in support of management decision making. Several factors distinguish data warehouses from operational databases. Because the two systems provide quite different functionalities and require different kinds of data, it is necessary to maintain data warehouses separately from operational databases.
- Data warehouses often adopt a **three-tier architecture**. The bottom tier is a *warehouse database server*, which is typically a relational database system. The middle tier is an *OLAP server*, and the top tier is a *client* that contains query and reporting tools.
- A data warehouse contains **back-end tools and utilities** for populating and refreshing the warehouse. These cover data extraction, data cleaning, data transformation, loading, refreshing, and warehouse management.
- Data warehouse **metadata** are data defining the warehouse objects. A metadata repository provides details regarding the warehouse structure, data history, the algorithms used for summarization, mappings from the source data to the warehouse form, system performance, and business terms and issues.
- A **multidimensional data model** is typically used for the design of corporate *data warehouses* and *departmental data marts*. Such a model can adopt a *star schema*, *snowflake schema*, or *fact constellation schema*. The core of the *multidimensional model* is the **data cube**, which consists of a large set of *facts* (or *measures*) and a number of *dimensions*. Dimensions are the entities or perspectives with respect to which an organization wants to keep records and are hierarchical in nature.

- A data cube consists of a **lattice of cuboids**, each corresponding to a different degree of summarization of the given multidimensional data.
- **Concept hierarchies** organize the values of attributes or dimensions into gradual abstraction levels. They are useful in mining at multiple abstraction levels.
- **Online analytical processing** can be performed in data warehouses/marts using the multidimensional data model. Typical OLAP operations include *roll-up*, and *drill-*(*down, across, through*), *slice-and-dice*, and *pivot* (*rotate*), as well as statistical operations such as ranking and computing moving averages and growth rates. OLAP operations can be implemented efficiently using the data cube structure.
- Data warehouses are used for *information processing* (querying and reporting), *analytical processing* (which allows users to navigate through summarized and detailed data by OLAP operations), and *data mining* (which supports knowledge discovery). OLAP-based data mining is referred to as **multidimensional data mining** (also known as exploratory multidimensional data mining, online analytical mining, or OLAM). It emphasizes the interactive and exploratory nature of data mining.
- OLAP servers may adopt a **relational OLAP** (**ROLAP**), a **multidimensional OLAP** (**MOLAP**), or a **hybrid OLAP** (**HOLAP**) implementation. A ROLAP server uses an extended relational DBMS that maps OLAP operations on multidimensional data to standard relational operations. A MOLAP server maps multidimensional data views directly to array structures. A HOLAP server combines ROLAP and MOLAP. For example, it may use ROLAP for historic data while maintaining frequently accessed data in a separate MOLAP store.
- **Full materialization** refers to the computation of all of the cuboids in the lattice defining a data cube. It typically requires an excessive amount of storage space, particularly as the number of dimensions and size of associated concept hierarchies grow. This problem is known as the **curse of dimensionality**. Alternatively, **partial materialization** is the selective computation of a subset of the cuboids or subcubes in the lattice. For example, an **iceberg cube** is a data cube that stores only those cube cells that have an aggregate value (e.g., `count`) above some minimum support threshold.
- OLAP query processing can be made more efficient with the use of indexing techniques. In **bitmap indexing**, each attribute has its own bitmap index table. Bitmap indexing reduces join, aggregation, and comparison operations to bit arithmetic. **Join indexing** registers the joinable rows of two or more relations from a relational database, reducing the overall cost of OLAP join operations. **Bitmapped join indexing**, which combines the bitmap and join index methods, can be used to further speed up OLAP query processing.
- **Data generalization** is a process that abstracts a large set of task-relevant data in a database from a relatively low conceptual level to higher conceptual levels. Data generalization approaches include data cube-based data aggregation and

attribute-oriented induction. **Concept description** is the most basic form of descriptive data mining. It describes a given set of task-relevant data in a concise and summarative manner, presenting interesting general properties of the data. Concept (or class) description consists of **characterization** and **comparison** (or **discrimination**). The former summarizes and describes a data collection, called the **target class**, whereas the latter summarizes and distinguishes one data collection, called the **target class**, from other data collection(s), collectively called the **contrasting class(es)**.

- **Concept characterization** can be implemented using **data cube (OLAP-based) approaches** and the **attribute-oriented induction approach**. These are attribute- or dimension-based generalization approaches. The **attribute-oriented induction approach** consists of the following techniques: *data focusing*, *data generalization by attribute removal or attribute generalization*, *count and aggregate value accumulation*, *attribute generalization control*, and *generalization data visualization*.
- **Concept comparison** can be performed using the attribute-oriented induction or data cube approaches in a manner similar to concept characterization. Generalized tuples from the target and contrasting classes can be quantitatively compared and contrasted.

4.7 Exercises

4.1 State why, for the integration of multiple heterogeneous information sources, many companies in industry prefer the *update-driven approach* (which constructs and uses data warehouses), rather than the *query-driven approach* (which applies wrappers and integrators). Describe situations where the query-driven approach is preferable to the update-driven approach.

4.2 Briefly compare the following concepts. You may use an example to explain your point(s).

(a) Snowflake schema, fact constellation, starnet query model

(b) Data cleaning, data transformation, refresh

(c) Discovery-driven cube, multifeature cube, virtual warehouse

4.3 Suppose that a data warehouse consists of the three dimensions *time*, *doctor*, and *patient*, and the two measures *count* and *charge*, where *charge* is the fee that a doctor charges a patient for a visit.

(a) Enumerate three classes of schemas that are popularly used for modeling data warehouses.

(b) Draw a schema diagram for the above data warehouse using one of the schema classes listed in (a).

(c) Starting with the base cuboid [*day*, *doctor*, *patient*], what specific *OLAP operations* should be performed in order to list the total fee collected by each doctor in 2010?

(d) To obtain the same list, write an SQL query assuming the data are stored in a relational database with the schema *fee* (*day, month, year, doctor, hospital, patient, count, charge*).

4.4 Suppose that a data warehouse for *Big_University* consists of the four dimensions *student, course, semester*, and *instructor*, and two measures *count* and *avg_grade*. At the lowest conceptual level (e.g., for a given student, course, semester, and instructor combination), the *avg_grade* measure stores the actual course grade of the student. At higher conceptual levels, *avg_grade* stores the average grade for the given combination.

(a) Draw a *snowflake schema* diagram for the data warehouse.

(b) Starting with the base cuboid [*student*, *course*, *semester*, *instructor*], what specific *OLAP operations* (e.g., roll-up from *semester* to *year*) should you perform in order to list the average grade of *CS* courses for each *Big_University* student.

(c) If each dimension has five levels (including all), such as "*student* < *major* < *status* < *university* < all", how many cuboids will this cube contain (including the base and apex cuboids)?

4.5 Suppose that a data warehouse consists of the four dimensions *date, spectator, location*, and *game*, and the two measures *count* and *charge*, where *charge* is the fare that a spectator pays when watching a game on a given date. Spectators may be students, adults, or seniors, with each category having its own charge rate.

(a) Draw a *star schema* diagram for the data warehouse.

(b) Starting with the base cuboid [*date*, *spectator*, *location*, *game*], what specific *OLAP operations* should you perform in order to list the total charge paid by student spectators at *GM_Place* in 2010?

(c) *Bitmap indexing* is useful in data warehousing. Taking this cube as an example, briefly discuss advantages and problems of using a bitmap index structure.

4.6 A data warehouse can be modeled by either a *star schema* or a *snowflake schema*. Briefly describe the similarities and the differences of the two models, and then analyze their advantages and disadvantages with regard to one another. Give your opinion of which might be more empirically useful and state the reasons behind your answer.

4.7 Design a data warehouse for a regional weather bureau. The weather bureau has about 1000 probes, which are scattered throughout various land and ocean locations in the region to collect basic weather data, including air pressure, temperature, and precipitation at each hour. All data are sent to the central station, which has collected such data for more than 10 years. Your design should facilitate efficient querying and online analytical processing, and derive general weather patterns in multidimensional space.

4.8 A popular data warehouse implementation is to construct a multidimensional database, known as a data cube. Unfortunately, this may often generate a huge, yet very sparse, multidimensional matrix.

(a) Present an example illustrating such a huge and sparse data cube.

(b) Design an implementation method that can elegantly overcome this sparse matrix problem. Note that you need to explain your data structures in detail and discuss the space needed, as well as how to retrieve data from your structures.

(c) Modify your design in (b) to handle *incremental data updates*. Give the reasoning behind your new design.

4.9 Regarding the *computation of measures* in a data cube:

(a) Enumerate three categories of measures, based on the kind of aggregate functions used in computing a data cube.

(b) For a data cube with the three dimensions *time, location*, and *item*, which category does the function *variance* belong to? Describe how to compute it if the cube is partitioned into many chunks.
Hint: The formula for computing *variance* is $\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x}_i)^2$, where $\bar{x}_i$ is the average of x_is.

(c) Suppose the function is "*top 10 sales*." Discuss how to efficiently compute this measure in a data cube.

4.10 Suppose a company wants to design a data warehouse to facilitate the analysis of moving vehicles in an online analytical processing manner. The company registers huge amounts of auto movement data in the format of (*Auto_ID, location, speed, time*). Each *Auto_ID* represents a vehicle associated with information (e.g., *vehicle_category, driver_category*), and each location may be associated with a street in a city. Assume that a street map is available for the city.

(a) Design such a data warehouse to facilitate effective online analytical processing in multidimensional space.

(b) The movement data may contain noise. Discuss how you would develop a method to automatically discover data records that were likely erroneously registered in the data repository.

(c) The movement data may be sparse. Discuss how you would develop a method that constructs a reliable data warehouse despite the sparsity of data.

(d) If you want to drive from A to B starting at a particular time, discuss how a system may use the data in this warehouse to work out a fast route.

4.11 Radio-frequency identification is commonly used to trace object movement and perform inventory control. An RFID reader can successfully read an RFID tag from a limited distance at any scheduled time. Suppose a company wants to design a data warehouse to facilitate the analysis of objects with RFID tags in an online analytical processing manner. The company registers huge amounts of RFID data in the format of *(RFID, at_location, time*), and also has some information about the objects carrying the RFID tag, for example, *(RFID, product_name, product_category, producer, date_produced, price*).

(a) Design a data warehouse to facilitate effective registration and online analytical processing of such data.

(b) The RFID data may contain lots of redundant information. Discuss a method that maximally reduces redundancy during data registration in the RFID data warehouse.

(c) The RFID data may contain lots of noise such as missing registration and misread IDs. Discuss a method that effectively cleans up the noisy data in the RFID data warehouse.

(d) You may want to perform online analytical processing to determine how many TV sets were shipped from the LA seaport to BestBuy in Champaign, IL, by *month*, *brand*, and *price_range*. Outline how this could be done efficiently if you were to store such RFID data in the warehouse.

(e) If a customer returns a jug of milk and complains that is has spoiled before its expiration date, discuss how you can investigate such a case in the warehouse to find out what the problem is, either in shipping or in storage.

4.12 In many applications, new data sets are incrementally added to the existing large data sets. Thus, an important consideration is whether a measure can be computed efficiently in an incremental manner. Use *count, standard deviation*, and *median* as examples to show that a distributive or algebraic measure facilitates efficient incremental computation, whereas a holistic measure does not.

4.13 Suppose that we need to record three measures in a data cube: min(), average(), and median(). Design an efficient computation and storage method for each measure given that the cube allows data to be *deleted incrementally* (i.e., in small portions at a time) from the cube.

4.14 In data warehouse technology, a multiple dimensional view can be implemented by a relational database technique (*ROLAP*), by a multidimensional database technique (*MOLAP*), or by a hybrid database technique (*HOLAP*).

(a) Briefly describe each implementation technique.

(b) For each technique, explain how each of the following functions may be implemented:

i. The generation of a data warehouse (including aggregation)

ii. Roll-up

iii. Drill-down

iv. Incremental updating

(c) Which implementation techniques do you prefer, and why?

4.15 Suppose that a data warehouse contains 20 dimensions, each with about five levels of granularity.

(a) Users are mainly interested in four particular dimensions, each having three frequently accessed levels for rolling up and drilling down. How would you design a data cube structure to support this preference efficiently?

(b) At times, a user may want to *drill through* the cube to the raw data for one or two particular dimensions. How would you support this feature?

4.16 A data cube, *C*, has *n* dimensions, and each dimension has exactly *p* distinct values in the base cuboid. Assume that there are no concept hierarchies associated with the dimensions.

(a) What is the *maximum number of cells* possible in the base cuboid?

(b) What is the *minimum number of cells* possible in the base cuboid?

(c) What is the *maximum number of cells* possible (including both base cells and aggregate cells) in the *C* data cube?

(d) What is the *minimum number of cells* possible in *C*?

4.17 What are the differences between the three main types of data warehouse usage: *information processing*, *analytical processing*, and *data mining*? Discuss the motivation behind *OLAP mining (OLAM)*.

4.8 Bibliographic Notes

There are a good number of introductory-level textbooks on data warehousing and OLAP technology—for example, Kimball, Ross, Thornthwaite, et al. [KRTM08]; Imhoff, Galemmo, and Geiger [IGG03]; and Inmon [Inm96]. Chaudhuri and Dayal [CD97] provide an early overview of data warehousing and OLAP technology. A set of research papers on materialized views and data warehouse implementations were collected in *Materialized Views: Techniques, Implementations, and Applications* by Gupta and Mumick [GM99].

The history of decision support systems can be traced back to the 1960s. However, the proposal to construct large data warehouses for multidimensional data analysis is credited to Codd [CCS93] who coined the term *OLAP* for *online analytical processing*. The OLAP Council was established in 1995. Widom [Wid95] identified several research problems in data warehousing. Kimball and Ross [KR02] provide an overview of the deficiencies of SQL regarding the ability to support comparisons that are common in the business world, and present a good set of application cases that require data warehousing and OLAP technology. For an overview of OLAP systems versus statistical databases, see Shoshani [Sho97].

Gray et al. [GCB+97] proposed the data cube as a relational aggregation operator generalizing group-by, crosstabs, and subtotals. Harinarayan, Rajaraman, and Ullman [HRU96] proposed a greedy algorithm for the partial materialization of cuboids in the computation of a data cube. Data cube computation methods have been investigated by numerous studies such as Sarawagi and Stonebraker [SS94]; Agarwal et al. [AAD+96]; Zhao, Deshpande, and Naughton [ZDN97]; Ross and Srivastava [RS97]; Beyer and Ramakrishnan [BR99]; Han, Pei, Dong, and Wang [HPDW01]; and Xin, Han, Li, and Wah [XHLW03]. These methods are discussed in depth in Chapter 5.

The concept of iceberg queries was first introduced in Fang, Shivakumar, Garcia-Molina et al. [FSGM+98]. The use of join indices to speed up relational query processing was proposed by Valduriez [Val87]. O'Neil and Graefe [OG95] proposed a bitmapped

join index method to speed up OLAP-based query processing. A discussion of the performance of bitmapping and other nontraditional index techniques is given in O'Neil and Quass [OQ97].

For work regarding the selection of materialized cuboids for efficient OLAP query processing, see, for example, Chaudhuri and Dayal [CD97]; Harinarayan, Rajaraman, and Ullman [HRU96]; and Sristava et al. [SDJL96]. Methods for cube size estimation can be found in Deshpande et al. [DNR+97], Ross and Srivastava [RS97], and Beyer and Ramakrishnan [BR99]. Agrawal, Gupta, and Sarawagi [AGS97] proposed operations for modeling multidimensional databases. Methods for answering queries quickly by online aggregation are described in Hellerstein, Haas, and Wang [HHW97] and Hellerstein et al. [HAC+99]. Techniques for estimating the top N queries are proposed in Carey and Kossman [CK98] and Donjerkovic and Ramakrishnan [DR99]. Further studies on intelligent OLAP and discovery-driven exploration of data cubes are presented in the bibliographic notes in Chapter 5.

This page intentionally left blank

5 Data Cube Technology

Data warehouse systems provide online analytical processing (OLAP) tools for interactive analysis of multidimensional data at varied granularity levels. OLAP tools typically use the *data cube* and a multidimensional data model to provide flexible access to summarized data. For example, a data cube can store precomputed measures (like count() and total_sales()) for multiple combinations of data dimensions (like *item*, *region*, and *customer*). Users can pose OLAP queries on the data. They can also interactively explore the data in a multidimensional way through OLAP operations like *drill-down* (to see more specialized data such as total sales per city) or *roll-up* (to see the data at a more generalized level such as total sales per country).

Although the data cube concept was originally intended for OLAP, it is also useful for data mining. **Multidimensional data mining** is an approach to data mining that integrates OLAP-based data analysis with knowledge discovery techniques. It is also known as *exploratory multidimensional data mining* and *online analytical mining* (*OLAM*). It searches for interesting patterns by exploring the data in multidimensional space. This gives users the freedom to dynamically focus on any subset of interesting dimensions. Users can interactively drill down or roll up to varying abstraction levels to find classification models, clusters, predictive rules, and outliers.

This chapter focuses on data cube technology. In particular, we study methods for data cube computation and methods for multidimensional data analysis. Precomputing a data cube (or parts of a data cube) allows for fast accessing of summarized data. Given the high dimensionality of most data, multidimensional analysis can run into performance bottlenecks. Therefore, it is important to study data cube computation techniques. Luckily, data cube technology provides many effective and scalable methods for cube computation. Studying these methods will also help in our understanding and further development of scalable methods for other data mining tasks such as the discovery of frequent patterns (Chapters 6 and 7).

We begin in Section 5.1 with preliminary concepts for cube computation. These summarize the data cube notion as a lattice of cuboids, and describe basic forms of cube materialization. General strategies for cube computation are given. Section 5.2 follows with an in-depth look at specific methods for data cube computation. We study both *full materialization* (i.e., where all the cuboids representing a data cube are precomputed

and thereby ready for use) and *partial cuboid materialization* (where, say, only the more "useful" parts of the data cube are precomputed). The *multiway array aggregation* method is detailed for full cube computation. Methods for partial cube computation, including *BUC*, *Star-Cubing*, and the use of *cube shell fragments*, are discussed.

In Section 5.3, we study cube-based query processing. The techniques described build on the standard methods of cube computation presented in Section 5.2. You will learn about *sampling cubes* for OLAP query answering on sampling data (e.g., survey data, which represent a sample or subset of a target data population of interest). In addition, you will learn how to compute *ranking cubes* for efficient top-k (ranking) query processing in large relational data sets.

In Section 5.4, we describe various ways to perform multidimensional data analysis using data cubes. *Prediction cubes* are introduced, which facilitate predictive modeling in multidimensional space. We discuss *multifeature cubes*, which compute complex queries involving multiple dependent aggregates at multiple granularities. You will also learn about the *exception-based discovery-driven exploration* of cube space, where visual cues are displayed to indicate discovered data exceptions at all aggregation levels, thereby guiding the user in the data analysis process.

5.1 Data Cube Computation: Preliminary Concepts

Data cubes facilitate the online analytical processing of multidimensional data. "*But how can we compute data cubes in advance, so that they are handy and readily available for query processing?*" This section contrasts full cube materialization (i.e., precomputation) versus various strategies for partial cube materialization. For completeness, we begin with a review of the basic terminology involving data cubes. We also introduce a cube cell notation that is useful for describing data cube computation methods.

5.1.1 Cube Materialization: Full Cube, Iceberg Cube, Closed Cube, and Cube Shell

Figure 5.1 shows a 3-D data cube for the dimensions *A*, *B*, and *C*, and an aggregate measure, *M*. Commonly used measures include count(), sum(), min(), max(), and total_sales(). A data cube is a lattice of cuboids. Each cuboid represents a group-by. *ABC* is the base cuboid, containing all three of the dimensions. Here, the aggregate measure, *M*, is computed for each possible combination of the three dimensions. The base cuboid is the least generalized of all the cuboids in the data cube. The most generalized cuboid is the apex cuboid, commonly represented as all. It contains one value—it aggregates measure *M* for all the tuples stored in the base cuboid. To drill down in the data cube, we move from the apex cuboid downward in the lattice. To roll up, we move from the base cuboid upward. For the purposes of our discussion in this chapter, we will always use the term *data cube* to refer to a lattice of cuboids rather than an individual cuboid.

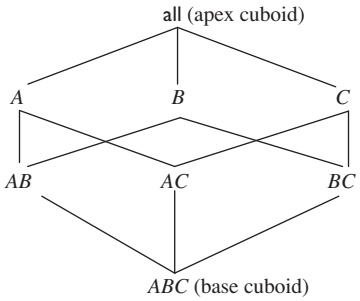


Figure 5.1 Lattice of cuboids making up a 3-D data cube with the dimensions *A*, *B*, and *C* for some aggregate measure, *M*.

A cell in the base cuboid is a **base cell**. A cell from a nonbase cuboid is an **aggregate cell**. An aggregate cell aggregates over one or more dimensions, where each aggregated dimension is indicated by a $*$ in the cell notation. Suppose we have an n-dimensional data cube. Let $a = (a_1, a_2, \ldots, a_n, measures)$ be a cell from one of the cuboids making up the data cube. We say that a is an ***m*-dimensional cell** (i.e., from an m-dimensional cuboid) if exactly m ($m \leq n$) values among $\{a_1, a_2, \ldots, a_n\}$ are *not* $*$. If $m = n$, then a is a base cell; otherwise, it is an aggregate cell (i.e., where $m < n$).

Example 5.1 **Base and aggregate cells.** Consider a data cube with the dimensions *month, city*, and *customer_group*, and the measure *sales*. (*Jan*, $*$, $*$, 2800) and ($*$, *Chicago*, $*$, 1200) are 1-D cells; (*Jan*, $*$, *Business*, 150) is a 2-D cell; and (*Jan*, *Chicago*, *Business*, 45) is a 3-D cell. Here, all base cells are 3-D, whereas 1-D and 2-D cells are aggregate cells. ■

An ancestor–descendant relationship may exist between cells. In an n-dimensional data cube, an i-D cell $a = (a_1, a_2, \ldots, a_n, measures_a)$ is an **ancestor** of a j-D cell $b = (b_1, b_2, \ldots, b_n, measures_b)$, and b is a **descendant** of a, if and only if (1) $i < j$, and (2) for $1 \leq k \leq n$, $a_k = b_k$ whenever $a_k \neq *$. In particular, cell a is called a **parent** of cell b, and b is a **child** of a, if and only if $j = i + 1$.

Example 5.2 **Ancestor and descendant cells.** Referring to Example 5.1, 1-D cell $a = (Jan, *, *,$ 2800) and 2-D cell $b = (Jan, *, Business, 150)$ are *ancestors* of 3-D cell $c = (Jan,$ *Chicago*, *Business*, 45); c is a *descendant* of both a and b; b is a *parent* of c; and c is a *child* of b. ■

To ensure fast OLAP, it is sometimes desirable to precompute the **full cube** (i.e., all the cells of all the cuboids for a given data cube). A method of full cube computation is given in Section 5.2.1. Full cube computation, however, is exponential to the number of dimensions. That is, a data cube of n dimensions contains 2^n cuboids. There are even

more cuboids if we consider concept hierarchies for each dimension.[1] In addition, the size of each cuboid depends on the cardinality of its dimensions. Thus, precomputation of the full cube can require huge and often excessive amounts of memory.

Nonetheless, full cube computation algorithms are important. **Individual** cuboids may be stored on secondary storage and accessed when necessary. Alternatively, we can use such algorithms to compute smaller cubes, consisting of a subset of the given set of dimensions, or a smaller range of possible values for some of the dimensions. In these cases, the smaller cube is a full cube for the given subset of dimensions and/or dimension values. A thorough understanding of full cube computation methods will help us develop efficient methods for computing partial cubes. Hence, it is important to explore scalable methods for computing all the cuboids making up a data cube, that is, for full materialization. These methods must take into consideration the limited amount of main memory available for cuboid computation, the total size of the computed data cube, as well as the time required for such computation.

Partial materialization of data cubes offers an interesting trade-off between storage space and response time for OLAP. Instead of computing the full cube, we can compute only a subset of the data cube's cuboids, or subcubes consisting of subsets of cells from the various cuboids.

Many cells in a cuboid may actually be of little or no interest to the data analyst. Recall that each cell in a full cube records an aggregate value such as count or sum. For many cells in a cuboid, the measure value will be zero. When the product of the cardinalities for the dimensions in a cuboid is large relative to the number of nonzero-valued tuples that are stored in the cuboid, then we say that the cuboid is **sparse**. If a cube contains many sparse cuboids, we say that the cube is **sparse**.

In many cases, a substantial amount of the cube's space could be taken up by a large number of cells with very low measure values. This is because the cube cells are often quite sparsely distributed within a multidimensional space. For example, a customer may only buy a few items in a store at a time. Such an event will generate only a few nonempty cells, leaving most other cube cells empty. In such situations, it is useful to materialize only those cells in a cuboid (group-by) with a measure value above some minimum threshold. In a data cube for sales, say, we may wish to materialize only those cells for which *count* $\geq$ *10* (i.e., where at least 10 tuples exist for the cell's given combination of dimensions), or only those cells representing *sales* $\geq$ $*100*. This not only saves processing time and disk space, but also leads to a more focused analysis. The cells that cannot pass the threshold are likely to be too trivial to warrant further analysis.

Such partially materialized cubes are known as **iceberg cubes**. The minimum threshold is called the **minimum support threshold**, or *minimum support* (*min_sup*), for short. By materializing only a fraction of the cells in a data cube, the result is seen as the "tip of the iceberg," where the "iceberg" is the potential full cube including all cells. An iceberg cube can be specified with an SQL query, as shown in Example 5.3.

[1] Eq. (4.1) of Section 4.4.1 gives the total number of cuboids in a data cube where each dimension has an associated concept hierarchy.

Example 5.3 **Iceberg cube.**

```
compute cube sales_iceberg as
select month, city, customer_group, count(*)
from salesInfo
cube by month, city, customer_group
having count(*) >= min_sup
```

The compute cube statement specifies the precomputation of the iceberg cube, *sales_iceberg*, with the dimensions *month*, *city*, and *customer_group*, and the aggregate measure count(). The input tuples are in the *salesInfo* relation. The cube by clause specifies that aggregates (group-by's) are to be formed for each of the possible subsets of the given dimensions. If we were computing the full cube, each group-by would correspond to a cuboid in the data cube lattice. The constraint specified in the having clause is known as the **iceberg condition**. Here, the iceberg measure is count(). Note that the iceberg cube computed here could be used to answer group-by queries on any combination of the specified dimensions of the form having count(*) >= v, where $v \geq min\_sup$. Instead of count(), the iceberg condition could specify more complex measures such as average().

If we were to omit the having clause, we would end up with the full cube. Let's call this cube *sales_cube*. The iceberg cube, *sales_iceberg*, excludes all the cells of *sales_cube* with a count that is less than *min_sup*. Obviously, if we were to set the minimum support to 1 in *sales_iceberg*, the resulting cube would be the full cube, *sales_cube*. ■

A naïve approach to computing an iceberg cube would be to first compute the full cube and then prune the cells that do not satisfy the iceberg condition. However, this is still prohibitively expensive. An efficient approach is to compute only the iceberg cube directly without computing the full cube. Sections 5.2.2 and 5.2.3 discuss methods for efficient iceberg cube computation.

Introducing iceberg cubes will lessen the burden of computing trivial aggregate cells in a data cube. However, we could still end up with a large number of uninteresting cells to compute. For example, suppose that there are 2 base cells for a database of 100 dimensions, denoted as $\{(a_1, a_2, a_3, \ldots, a_{100}) : 10, (a_1, a_2, b_3, \ldots, b_{100}) : 10\}$, where each has a cell count of 10. If the minimum support is set to 10, there will still be an impermissible number of cells to compute and store, although most of them are not interesting. For example, there are $2^{101} - 6$ distinct aggregate cells,[2] like $\{(a_1, a_2, a_3, a_4, \ldots, a_{99}, *) : 10, \ldots, (a_1, a_2, *, a_4, \ldots, a_{99}, a_{100}) : 10, \ldots, (a_1, a_2, a_3, *, \ldots, *, *) : 10\}$, but most of them do not contain much new information. If we ignore all the aggregate cells that can be obtained by replacing some constants by $*$'s while keeping the same measure value, there are only three distinct cells left: $\{(a_1, a_2, a_3, \ldots, a_{100}) : 10, (a_1, a_2, b_3, \ldots, b_{100}) : 10, (a_1, a_2, *, \ldots, *) : 20\}$. That is, out of $2^{101} - 4$ distinct base and aggregate cells, only three really offer valuable information.

[2]The proof is left as an exercise for the reader.

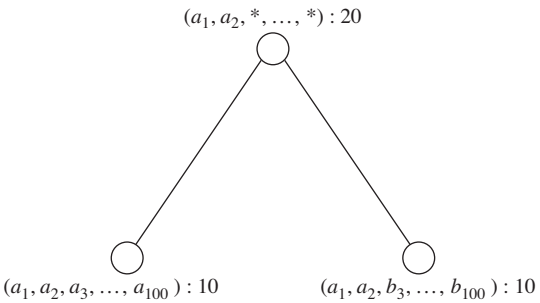


Figure 5.2 Three closed cells forming the lattice of a closed cube.

To systematically compress a data cube, we need to introduce the concept of *closed coverage*. A cell, c, is a *closed cell* if there exists no cell, d, such that d is a specialization (descendant) of cell c (i.e., where d is obtained by replacing $*$ in c with a non-$*$ value), and d has the same measure value as c. A **closed cube** is a data cube consisting of only closed cells. For example, the three cells derived in the preceding paragraph are the three closed cells of the data cube for the data set $\{(a_1, a_2, a_3, \ldots, a_{100}) : 10, (a_1, a_2, b_3, \ldots, b_{100}) : 10\}$. They form the lattice of a closed cube as shown in Figure 5.2. Other nonclosed cells can be derived from their corresponding closed cells in this lattice. For example, "$(a_1, *, *, \ldots, *) : 20$" can be derived from "$(a_1, a_2, *, \ldots, *) : 20$" because the former is a generalized nonclosed cell of the latter. Similarly, we have "$(a_1, a_2, b_3, *, \ldots, *) : 10$."

Another strategy for partial materialization is to precompute only the cuboids involving a small number of dimensions such as three to five. These cuboids form a **cube shell** for the corresponding data cube. Queries on additional combinations of the dimensions will have to be computed on-the-fly. For example, we could compute all cuboids with three dimensions or less in an n-dimensional data cube, resulting in a cube shell of size 3. This, however, can still result in a large number of cuboids to compute, particularly when n is large. Alternatively, we can choose to precompute only portions or *fragments* of the cube shell based on cuboids of interest. Section 5.2.4 discusses a method for computing **shell fragments** and explores how they can be used for efficient OLAP query processing.

5.1.2 General Strategies for Data Cube Computation

There are several methods for efficient data cube computation, based on the various kinds of cubes described in Section 5.1.1. In general, there are two basic data structures used for storing cuboids. The implementation of relational OLAP (ROLAP) uses relational tables, whereas multidimensional arrays are used in multidimensional OLAP (MOLAP). Although ROLAP and MOLAP may each explore different cube computation techniques, some optimization "tricks" can be shared among the different

data representations. The following are general optimization techniques for efficient computation of data cubes.

Optimization Technique 1: Sorting, hashing, and grouping. Sorting, hashing, and grouping operations should be applied to the dimension attributes to reorder and cluster related tuples.

In cube computation, aggregation is performed on the tuples (or cells) that share the same set of dimension values. Thus, it is important to explore sorting, hashing, and grouping operations to access and group such data together to facilitate computation of such aggregates.

To compute total sales by *branch*, *day*, and *item*, for example, it can be more efficient to sort tuples or cells by *branch*, and then by *day*, and then group them according to the *item* name. Efficient implementations of such operations in large data sets have been extensively studied in the database research community. Such implementations can be extended to data cube computation.

This technique can also be further extended to perform **shared-sorts** (i.e., sharing sorting costs across multiple cuboids when sort-based methods are used), or to perform **shared-partitions** (i.e., sharing the partitioning cost across multiple cuboids when hash-based algorithms are used).

Optimization Technique 2: Simultaneous aggregation and caching of intermediate results. In cube computation, it is efficient to compute higher-level aggregates from previously computed lower-level aggregates, rather than from the base fact table. Moreover, simultaneous aggregation from cached intermediate computation results may lead to the reduction of expensive disk input/output (I/O) operations.

To compute sales by *branch*, for example, we can use the intermediate results derived from the computation of a lower-level cuboid such as sales by *branch* and *day*. This technique can be further extended to perform **amortized scans** (i.e., computing as many cuboids as possible at the same time to amortize disk reads).

Optimization Technique 3: Aggregation from the smallest child when there exist multiple child cuboids. When there exist multiple child cuboids, it is usually more efficient to compute the desired parent (i.e., more generalized) cuboid from the smallest, previously computed child cuboid.

To compute a sales cuboid, C_{branch}, when there exist two previously computed cuboids, $C_{\{branch,year\}}$ and $C_{\{branch,item\}}$, for example, it is obviously more efficient to compute C_{branch} from the former than from the latter if there are many more distinct items than distinct years.

Many other optimization techniques may further improve computational efficiency. For example, string dimension attributes can be mapped to integers with values ranging from zero to the cardinality of the attribute.

In iceberg cube computation the following optimization technique plays a particularly important role.

Optimization Technique 4: The Apriori pruning method can be explored to compute iceberg cubes efficiently. The **Apriori property**,[3] in the context of data cubes, states as follows: *If a given cell does not satisfy minimum support, then no descendant of the cell (i.e., more specialized cell) will satisfy minimum support either.* This property can be used to substantially reduce the computation of iceberg cubes.

Recall that the specification of iceberg cubes contains an iceberg condition, which is a constraint on the cells to be materialized. A common iceberg condition is that the cells must satisfy a *minimum support* threshold such as a minimum count or sum. In this situation, the Apriori property can be used to prune away the exploration of the cell's descendants. For example, if the count of a cell, *c*, in a cuboid is less than a minimum support threshold, *v*, then the count of any of *c*'s descendant cells in the lower-level cuboids can never be greater than or equal to *v*, and thus can be pruned.

In other words, if a condition (e.g., the iceberg condition specified in the `having` clause) is violated for some cell *c*, then every descendant of *c* will also violate that condition. Measures that obey this property are known as **antimonotonic**.[4] This form of pruning was made popular in frequent pattern mining, yet also aids in data cube computation by cutting processing time and disk space requirements. It can lead to a more focused analysis because cells that cannot pass the threshold are unlikely to be of interest.

In the following sections, we introduce several popular methods for efficient cube computation that explore these optimization strategies.

5.2 Data Cube Computation Methods

Data cube computation is an essential task in data warehouse implementation. The precomputation of all or part of a data cube can greatly reduce the response time and enhance the performance of online analytical processing. However, such computation is challenging because it may require substantial computational time and storage space. This section explores efficient methods for data cube computation. Section 5.2.1 describes the *multiway array aggregation* (MultiWay) method for computing full cubes. Section 5.2.2 describes a method known as BUC, which computes iceberg cubes from the apex cuboid downward. Section 5.2.3 describes the Star-Cubing method, which integrates top-down and bottom-up computation.

Finally, Section 5.2.4 describes a shell-fragment cubing approach that computes shell fragments for efficient high-dimensional OLAP. To simplify our discussion, we exclude

[3]The Apriori property was proposed in the Apriori algorithm for association rule mining by Agrawal and Srikant [AS94b]. Many algorithms in association rule mining have adopted this property (see Chapter 6).

[4]**Antimonotone** is based on *condition violation*. This differs from **monotone**, which is based on *condition satisfaction*.

the cuboids that would be generated by climbing up any existing hierarchies for the dimensions. Those cube types can be computed by extension of the discussed methods. Methods for the efficient computation of closed cubes are left as an exercise for interested readers.

5.2.1 Multiway Array Aggregation for Full Cube Computation

The **multiway array aggregation** (or simply **MultiWay**) method computes a full data cube by using a multidimensional array as its basic data structure. It is a typical MOLAP approach that uses direct array addressing, where dimension values are accessed via the position or index of their corresponding array locations. Hence, MultiWay cannot perform any value-based reordering as an optimization technique. A different approach is developed for the array-based cube construction, as follows:

1. Partition the array into chunks. A **chunk** is a subcube that is small enough to fit into the memory available for cube computation. **Chunking** is a method for dividing an n-dimensional array into small n-dimensional chunks, where each chunk is stored as an object on disk. The chunks are compressed so as to remove wasted space resulting from *empty array cells*. A cell is *empty* if it does not contain any valid data (i.e., its cell count is 0). For instance, "*chunkID* + *offset*" can be used as a cell-addressing mechanism to **compress a sparse array structure** and when searching for cells within a chunk. Such a compression technique is powerful at handling sparse cubes, both on disk and in memory.
2. Compute aggregates by visiting (i.e., accessing the values at) cube cells. The order in which cells are visited can be optimized so as to *minimize the number of times that each cell must be revisited*, thereby reducing memory access and storage costs. The trick is to exploit this ordering so that portions of the aggregate cells in multiple cuboids can be computed simultaneously, and any unnecessary revisiting of cells is avoided.

This chunking technique involves "overlapping" some of the aggregation computations; therefore, it is referred to as multiway array aggregation. It performs **simultaneous aggregation**, that is, it computes aggregations simultaneously on multiple dimensions.

We explain this approach to array-based cube construction by looking at a concrete example.

Example 5.4 **Multiway array cube computation.** Consider a 3-D data array containing the three dimensions A, B, and C. The 3-D array is partitioned into small, memory-based chunks. In this example, the array is partitioned into 64 chunks as shown in Figure 5.3. Dimension A is organized into four equal-sized partitions: a_0, a_1, a_2, and a_3. Dimensions B and C are similarly organized into four partitions each. Chunks 1, 2, ..., 64 correspond to the subcubes $a_0b_0c_0$, $a_1b_0c_0$, ..., $a_3b_3c_3$, respectively. Suppose that the cardinality of

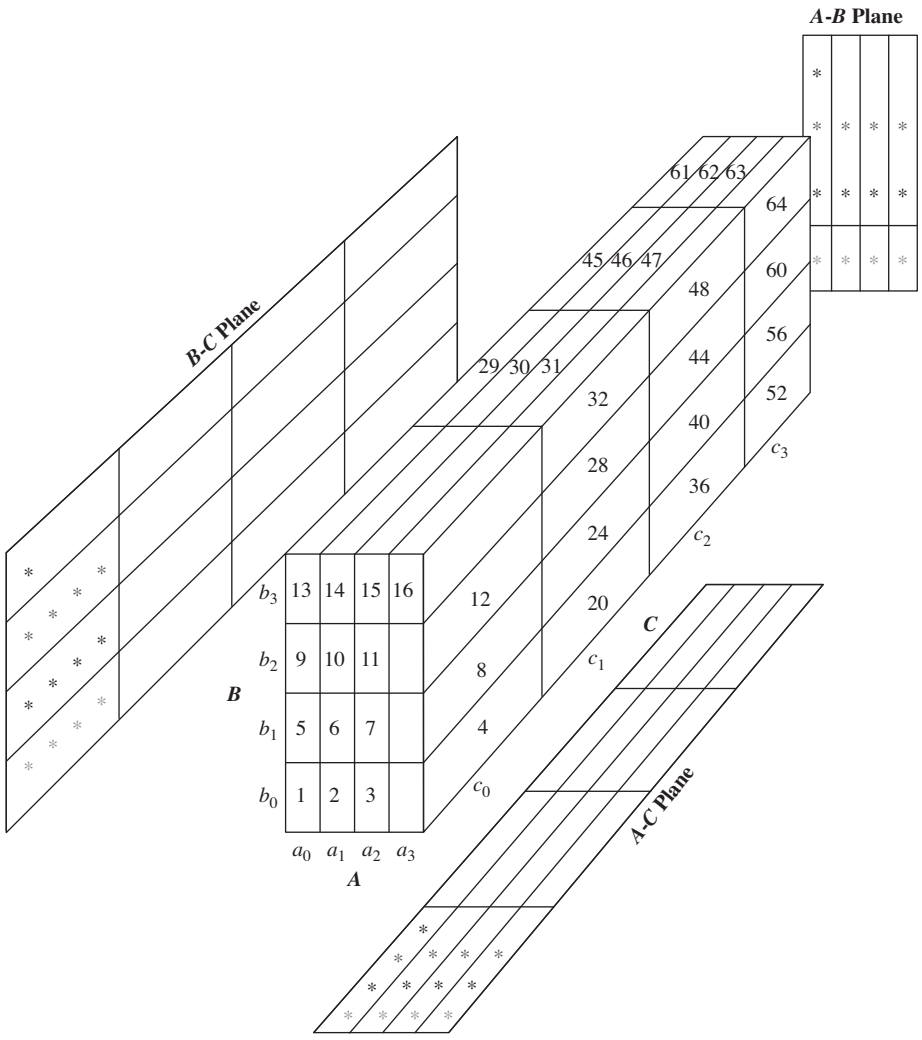


Figure 5.3 A 3-D array for the dimensions *A*, *B*, and *C*, organized into 64 *chunks*. Each chunk is small enough to fit into the memory available for cube computation. The ∗'s indicate the chunks from 1 to 13 that have been aggregated so far in the process.

the dimensions *A*, *B*, and *C* is 40, 400, and 4000, respectively. Thus, the size of the array for each dimension, *A*, *B*, and *C*, is also 40, 400, and 4000, respectively. The size of each partition in *A*, *B*, and *C* is therefore 10, 100, and 1000, respectively. Full materialization of the corresponding data cube involves the computation of all the cuboids defining this cube. The resulting full cube consists of the following cuboids:

- The base cuboid, denoted by *ABC* (from which all the other cuboids are directly or indirectly computed). This cube is already computed and corresponds to the given 3-D array.
- The 2-D cuboids, *AB*, *AC*, and *BC*, which respectively correspond to the group-by's *AB*, *AC*, and *BC*. These cuboids must be computed.
- The 1-D cuboids, *A*, *B*, and *C*, which respectively correspond to the group-by's *A*, *B*, and *C*. These cuboids must be computed.
- The 0-D (apex) cuboid, denoted by **all**, which corresponds to the group-by (); that is, there is no group-by here. This cuboid must be computed. It consists of only one value. If, say, the data cube measure is **count**, then the value to be computed is simply the total count of all the tuples in *ABC*.

Let's look at how the multiway array aggregation technique is used in this computation. There are many possible orderings with which chunks can be read into memory for use in cube computation. Consider the ordering labeled from 1 to 64, shown in Figure 5.3. Suppose we want to compute the b_0c_0 chunk of the *BC* cuboid. We allocate space for this chunk in *chunk memory*. By scanning *ABC* chunks 1 through 4, the b_0c_0 chunk is computed. That is, the cells for b_0c_0 are aggregated over a_0 to a_3. The chunk memory can then be assigned to the next chunk, b_1c_0, which completes its aggregation after the scanning of the next four *ABC* chunks: 5 through 8. Continuing in this way, the entire *BC* cuboid can be computed. Therefore, only *one BC* chunk needs to be in memory at a time, for the computation of all the *BC* chunks.

In computing the *BC* cuboid, we will have scanned each of the 64 chunks. "*Is there a way to avoid having to rescan all of these chunks for the computation of other cuboids such as AC and AB?*" The answer is, most definitely, *yes*. This is where the "multiway computation" or "simultaneous aggregation" idea comes in. For example, when chunk 1 (i.e., $a_0b_0c_0$) is being scanned (say, for the computation of the 2-D chunk b_0c_0 of *BC*, as described previously), all of the other 2-D chunks relating to $a_0b_0c_0$ can be simultaneously computed. That is, when $a_0b_0c_0$ is being scanned, each of the three chunks (b_0c_0, a_0c_0, and a_0b_0) on the three 2-D aggregation planes (*BC*, *AC*, and *AB*) should be computed then as well. In other words, multiway computation simultaneously aggregates to each of the 2-D planes while a 3-D chunk is in memory.

Now let's look at how different orderings of chunk scanning and of cuboid computation can affect the overall data cube computation efficiency. Recall that the size of the dimensions *A*, *B*, and *C* is 40, 400, and 4000, respectively. Therefore, the largest 2-D plane is *BC* (of size $400 \times 4000 = 1{,}600{,}000$). The second largest 2-D plane is *AC* (of size $40 \times 4000 = 160{,}000$). *AB* is the smallest 2-D plane (of size $40 \times 400 = 16{,}000$).

Suppose that the chunks are scanned in the order shown, from chunks 1 to 64. As previously mentioned, b_0c_0 is fully aggregated after scanning the row containing chunks 1 through 4; b_1c_0 is fully aggregated after scanning chunks 5 through 8, and so on. Thus, we need to scan four chunks of the 3-D array to *fully* compute one chunk of the *BC* cuboid (where *BC* is the largest of the 2-D planes). In other words, by scanning in this

order, one *BC* chunk is fully computed for each row scanned. In comparison, the complete computation of one chunk of the second largest 2-D plane, *AC*, requires scanning 13 chunks, given the ordering from 1 to 64. That is, a_0c_0 is fully aggregated only after the scanning of chunks 1, 5, 9, and 13.

Finally, the complete computation of one chunk of the smallest 2-D plane, *AB*, requires scanning 49 chunks. For example, a_0b_0 is fully aggregated after scanning chunks 1, 17, 33, and 49. Hence, *AB* requires the longest scan of chunks to complete its computation. To avoid bringing a 3-D chunk into memory more than once, the minimum memory requirement for holding all relevant 2-D planes in chunk memory, according to the chunk ordering of 1 to 64, is as follows: 40×400 (for the whole *AB* plane) + 40×1000 (for one column of the *AC* plane) + 100×1000 (for one *BC* plane chunk) = 16,000 + 40,000 + 100,000 = 156,000 memory units.

Suppose, instead, that the chunks are scanned in the order 1, 17, 33, 49, 5, 21, 37, 53, and so on. That is, suppose the scan is in the order of first aggregating toward the *AB* plane, and then toward the *AC* plane, and lastly toward the *BC* plane. The minimum memory requirement for holding 2-D planes in chunk memory would be as follows: 400×4000 (for the whole *BC* plane) + 40×1000 (for one *AC* plane row) + 10×100 (for one *AB* plane chunk) = 1,600,000 + 40,000 + 1000 = 1,641,000 memory units. Notice that this is *more than 10 times* the memory requirement of the scan ordering of 1 to 64.

Similarly, we can work out the minimum memory requirements for the multiway computation of the 1-D and 0-D cuboids. Figure 5.4 shows the most efficient way to compute 1-D cuboids. Chunks for 1-D cuboids *A* and *B* are computed during the computation of the smallest 2-D cuboid, *AB*. The smallest 1-D cuboid, *A*, will have all of its chunks allocated in memory, whereas the larger 1-D cuboid, *B*, will have only one chunk allocated in memory at a time. Similarly, chunk *C* is computed during the computation of the second smallest 2-D cuboid, *AC*, requiring only one chunk in memory at a time. Based on this analysis, we see that the most efficient ordering in this array cube computation is the chunk ordering of 1 to 64, with the stated memory allocation strategy. ■

Example 5.4 assumes that there is enough memory space for *one-pass* cube computation (i.e., to compute all of the cuboids from one scan of all the chunks). If there is insufficient memory space, the computation will require more than one pass through the 3-D array. In such cases, however, the basic principle of ordered chunk computation remains the same. MultiWay is most effective when the product of the cardinalities of dimensions is moderate and the data are not too sparse. When the dimensionality is high or the data are very sparse, the in-memory arrays become too large to fit in memory, and this method becomes infeasible.

With the use of appropriate sparse array compression techniques and careful ordering of the computation of cuboids, it has been shown by experiments that MultiWay array cube computation is significantly faster than traditional ROLAP (relational record-based) computation. Unlike ROLAP, the array structure of MultiWay does not require saving space to store search keys. Furthermore, MultiWay uses direct array addressing,

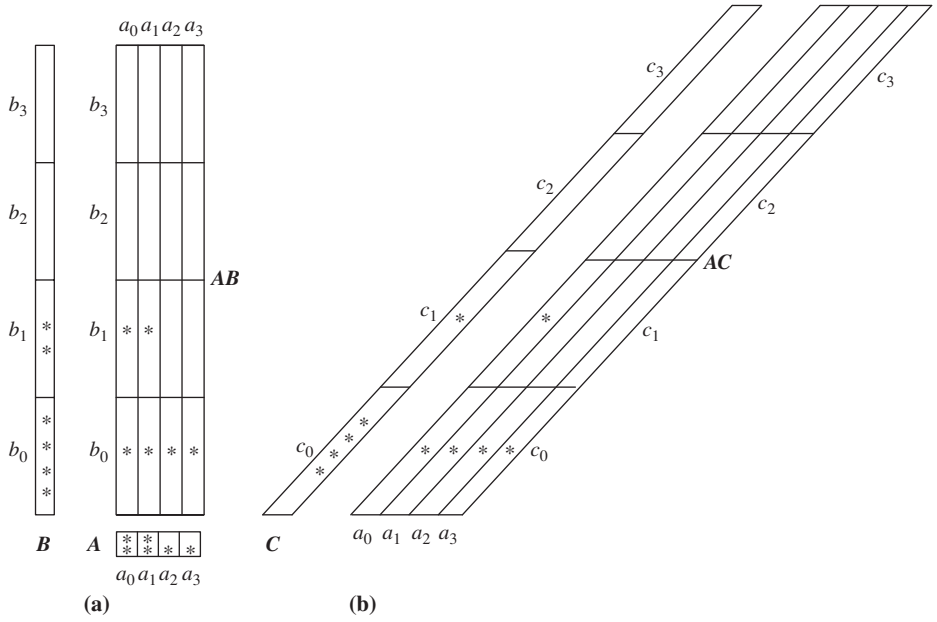


Figure 5.4 Memory allocation and computation order for computing Example 5.4's 1-D cuboids. (a) The 1-D cuboids, *A* and *B*, are aggregated during the computation of the smallest 2-D cuboid, *AB*. (b) The 1-D cuboid, *C*, is aggregated during the computation of the second smallest 2-D cuboid, *AC*. The ∗'s represent chunks that, so far, have been aggregated to.

which is faster than ROLAP's key-based addressing search strategy. For ROLAP cube computation, instead of cubing a table directly, it can be faster to convert the table to an array, cube the array, and then convert the result back to a table. However, this observation works only for cubes with a relatively small number of dimensions, because the number of cuboids to be computed is exponential to the number of dimensions.

"*What would happen if we tried to use MultiWay to compute iceberg cubes?*" Remember that the Apriori property states that if a given cell does not satisfy minimum support, then neither will any of its descendants. Unfortunately, MultiWay's computation starts from the base cuboid and progresses upward toward more generalized, ancestor cuboids. It cannot take advantage of Apriori pruning, which requires a parent node to be computed before its child (i.e., more specific) nodes. For example, if the count of a cell *c* in, say, *AB*, does not satisfy the minimum support specified in the iceberg condition, we cannot prune away cell *c*, because the count of *c*'s ancestors in the *A* or *B* cuboids may be greater than the minimum support, and their computation will need aggregation involving the count of *c*.

5.2.2 BUC: Computing Iceberg Cubes from the Apex Cuboid Downward

BUC is an algorithm for the computation of sparse and iceberg cubes. Unlike MultiWay, BUC constructs the cube from the apex cuboid toward the base cuboid. This allows BUC to share data partitioning costs. This processing order also allows BUC to prune during construction, using the Apriori property.

Figure 5.5 shows a lattice of cuboids, making up a 3-D data cube with the dimensions *A*, *B*, and *C*. The apex (0-D) cuboid, representing the concept all (i.e., $(*, *, *)$), is at the top of the lattice. This is the most aggregated or generalized level. The 3-D base cuboid, *ABC*, is at the bottom of the lattice. It is the least aggregated (most detailed or specialized) level. This representation of a lattice of cuboids, with the apex at the top and the base at the bottom, is commonly accepted in data warehousing. It consolidates the notions of *drill-down* (where we can move from a highly aggregated cell to lower, more detailed cells) and *roll-up* (where we can move from detailed, low-level cells to higher-level, more aggregated cells).

BUC stands for "Bottom-Up Construction." However, according to the lattice convention described before and used throughout this book, the BUC processing order is actually top-down! The BUC authors view a lattice of cuboids in the reverse order,

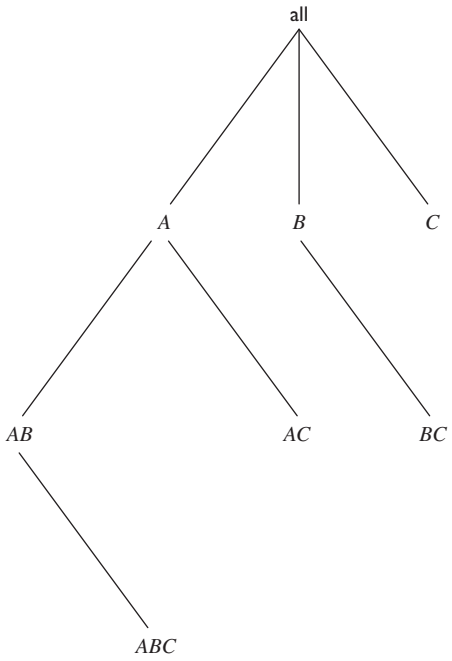


Figure 5.5 BUC's exploration for a 3-D data cube computation. Note that the computation starts from the apex cuboid.

with the apex cuboid at the bottom and the base cuboid at the top. In that view, BUC does bottom-up construction. However, because we adopt the application worldview where *drill-down* refers to drilling from the apex cuboid down toward the base cuboid, the exploration process of BUC is regarded as top-down. BUC's exploration for the computation of a 3-D data cube is shown in Figure 5.5.

The BUC algorithm is shown on the next page in Figure 5.6. We first give an explanation of the algorithm and then follow up with an example. Initially, the algorithm is called with the input relation (set of tuples). BUC aggregates the entire input (line 1) and writes the resulting total (line 3). (Line 2 is an optimization feature that is discussed later in our example.) For each dimension d (line 4), the input is partitioned on d (line 6). On return from Partition(), *dataCount* contains the total number of tuples for each distinct value of dimension d. Each distinct value of d *forms its own partition.* Line 8 iterates through each partition. Line 10 tests the partition for minimum support. That is, if the number of tuples in the partition satisfies (i.e., is $\geq$) the minimum support, then the partition becomes the input relation for a recursive call made to BUC, which computes the iceberg cube on the partitions for dimensions $d + 1$ to *numDims* (line 12).

Note that for a full cube (i.e., where minimum support in the having clause is 1), the minimum support condition is always satisfied. Thus, the recursive call descends one level deeper into the lattice. On return from the recursive call, we continue with the next partition for d. After all the partitions have been processed, the entire process is repeated for each of the remaining dimensions.

Example 5.5 BUC construction of an iceberg cube. Consider the iceberg cube expressed in SQL as follows:

```
compute cube iceberg_cube as
select A, B, C, D, count(*)
from R
cube by A, B, C, D
having count(*) >= 3
```

Let's see how BUC constructs the iceberg cube for the dimensions A, B, C, and D, where 3 is the minimum support count. Suppose that dimension A has four distinct values, a_1, a_2, a_3, a_4; B has four distinct values, b_1, b_2, b_3, b_4; C has two distinct values, c_1, c_2; and D has two distinct values, d_1, d_2. If we consider each group-by to be a *partition*, then we must compute every combination of the grouping attributes that satisfy the minimum support (i.e., that have three tuples).

Figure 5.7 illustrates how the input is partitioned first according to the different attribute values of dimension A, and then B, C, and D. To do so, BUC scans the input, aggregating the tuples to obtain a count for all, corresponding to the cell $(*, *, *, *)$. Dimension A is used to split the input into four partitions, one for each distinct value of A. The number of tuples (counts) for each distinct value of A is recorded in *dataCount.*

BUC uses the Apriori property to save time while searching for tuples that satisfy the iceberg condition. Starting with A dimension value, a_1, the a_1 partition is aggregated, creating one tuple for the A group-by, corresponding to the cell $(a_1, *, *, *)$.

Algorithm: BUC. Algorithm for the computation of sparse and iceberg cubes.

Input:

- *input*: the relation to aggregate;
- *dim*: the starting dimension for this iteration.

Globals:

- constant *numDims*: the total number of dimensions;
- constant *cardinality[numDims]*: the cardinality of each dimension;
- constant *min_sup*: the minimum number of tuples in a partition for it to be output;
- *outputRec*: the current output record;
- *dataCount[numDims]*: stores the size of each partition. *dataCount[i]* is a list of integers of size *cardinality[i]*.

Output: Recursively output the iceberg cube cells satisfying the minimum support.

Method:

```
(1)  Aggregate(input); // Scan input to compute measure, e.g., count. Place result in outputRec.
(2)  if input.count() == 1 then // Optimization
         WriteDescendants(input[0], dim); return;
     endif
(3)  write outputRec;
(4)  for (d = dim; d < numDims; d++) do //Partition each dimension
(5)      C = cardinality[d];
(6)      Partition(input, d, C, dataCount[d]); //create C partitions of data for dimension d
(7)      k = 0;
(8)      for (i = 0; i < C; i++) do // for each partition (each value of dimension d)
(9)              c = dataCount[d][i];
(10)             if c >= min_sup then // test the iceberg condition
(11)                     outputRec.dim[d] = input[k].dim[d];
(12)                     BUC(input[k..k + c − 1], d + 1); // aggregate on next dimension
(13)             endif
(14)             k +=c;
(15)     endfor
(16)     outputRec.dim[d] = all;
(17) endfor
```

Figure 5.6 BUC algorithm for sparse or iceberg cube computation. *Source:* Beyer and Ramakrishnan [BR99].

Suppose $(a_1, *, *, *)$ satisfies the minimum support, in which case a recursive call is made on the partition for a_1. BUC partitions a_1 on the dimension B. It checks the count of $(a_1, b_1, *, *)$ to see if it satisfies the minimum support. If it does, it outputs the aggregated tuple to the AB group-by and recurses on $(a_1, b_1, *, *)$ to partition on C, starting

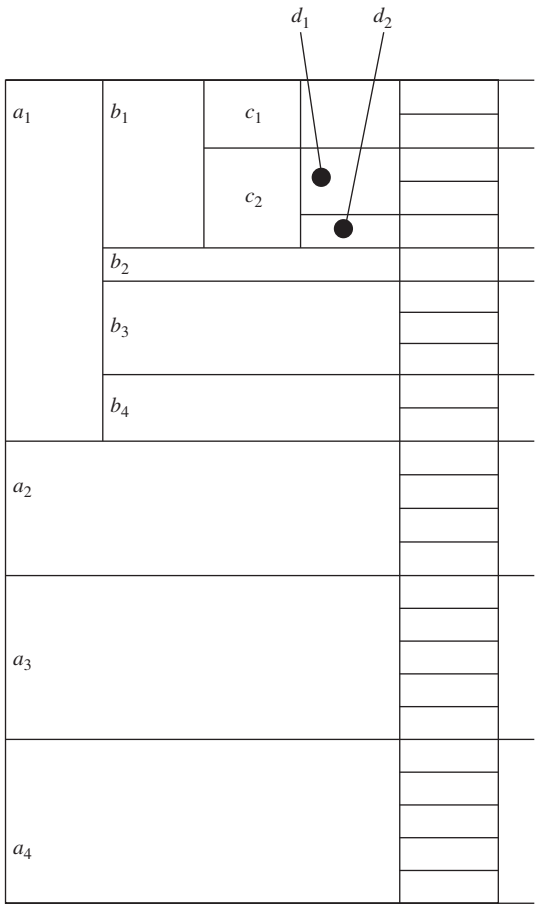


Figure 5.7 BUC partitioning snapshot given an example 4-D data set.

with c_1. Suppose the cell count for $(a_1, b_1, c_1, *)$ is 2, which does not satisfy the minimum support. According to the Apriori property, if a cell does not satisfy the minimum support, then neither can any of its descendants. Therefore, BUC prunes any further exploration of $(a_1, b_1, c_1, *)$. That is, it avoids partitioning this cell on dimension D. It backtracks to the a_1, b_1 partition and recurses on $(a_1, b_1, c_2, *)$, and so on. By checking the iceberg condition each time before performing a recursive call, BUC saves a great deal of processing time whenever a cell's count does not satisfy the minimum support.

The partition process is facilitated by a linear sorting method, CountingSort. CountingSort is fast because it does not perform any key comparisons to find partition boundaries. In addition, the counts computed during the sort can be reused to compute the group-by's in BUC. Line 2 is an optimization for partitions having a count of 1 such as $(a_1, b_2, *, *)$ in our example. To save on partitioning costs, the count is written

to each of the tuple's descendant group-by's. This is particularly useful since, in practice, many partitions have a single tuple. ■

The BUC performance is sensitive to the order of the dimensions and to skew in the data. Ideally, the most discriminating dimensions should be processed first. Dimensions should be processed in the order of decreasing cardinality. The higher the cardinality, the smaller the partitions, and thus the more partitions there will be, thereby providing BUC with a greater opportunity for pruning. Similarly, the more uniform a dimension (i.e., having less skew), the better it is for pruning.

BUC's major contribution is the idea of sharing partitioning costs. However, unlike MultiWay, it does not share the computation of aggregates between parent and child group-by's. For example, the computation of cuboid *AB* does not help that of *ABC*. The latter needs to be computed essentially from scratch.

5.2.3 Star-Cubing: Computing Iceberg Cubes Using a Dynamic Star-Tree Structure

In this section, we describe the **Star-Cubing** algorithm for computing iceberg cubes. Star-Cubing combines the strengths of the other methods we have studied up to this point. It integrates top-down and bottom-up cube computation and explores both multidimensional aggregation (similar to MultiWay) and Apriori-like pruning (similar to BUC). It operates from a data structure called a star-tree, which performs lossless data compression, thereby reducing the computation time and memory requirements.

The Star-Cubing algorithm explores both the bottom-up and top-down computation models as follows: On the global computation order, it uses the bottom-up model. However, it has a sublayer underneath based on the top-down model, which explores the notion of *shared dimensions*, as we shall see in the following. This integration allows the algorithm to aggregate on multiple dimensions while still partitioning parent group-by's and pruning child group-by's that do not satisfy the iceberg condition.

Star-Cubing's approach is illustrated in Figure 5.8 for a 4-D data cube computation. If we were to follow only the bottom-up model (similar to MultiWay), then the cuboids marked as pruned by Star-Cubing would still be explored. Star-Cubing is able to prune the indicated cuboids because it considers shared dimensions. *ACD*/*A* means cuboid *ACD* has shared dimension *A*, *ABD*/*AB* means cuboid *ABD* has shared dimension *AB*, *ABC*/*ABC* means cuboid *ABC* has shared dimension *ABC*, and so on. This comes from the generalization that all the cuboids in the subtree rooted at *ACD* include dimension *A*, all those rooted at *ABD* include dimensions *AB*, and all those rooted at *ABC* include dimensions *ABC* (even though there is only one such cuboid). We call these common dimensions the **shared dimensions** of those particular subtrees.

The introduction of shared dimensions facilitates shared computation. Because the shared dimensions are identified early on in the tree expansion, we can avoid recomputing them later. For example, cuboid *AB* extending from *ABD* in Figure 5.8 would actually be pruned because *AB* was already computed in *ABD*/*AB*. Similarly, cuboid

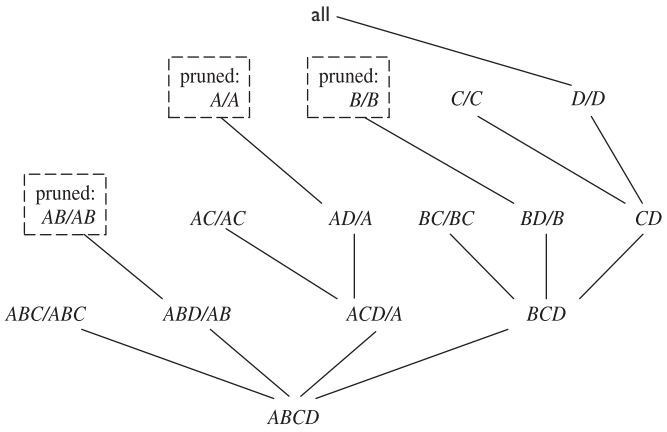


Figure 5.8 Star-Cubing: bottom-up computation with top-down expansion of shared dimensions.

A extending from *AD* would also be pruned because it was already computed in *ACD*/*A*.

Shared dimensions allow us to do Apriori-like pruning if the measure of an iceberg cube, such as *count*, is antimonotonic. That is, if the aggregate value on a shared dimension does not satisfy the iceberg condition, then *all the cells descending from this shared dimension cannot satisfy the iceberg condition either*. These cells and their descendants can be pruned because these descendant cells are, by definition, more specialized (i.e., contain more dimensions) than those in the shared dimension(s). The number of tuples covered by the descendant cells will be less than or equal to the number of tuples covered by the shared dimensions. Therefore, if the aggregate value on a shared dimension fails the iceberg condition, the descendant cells cannot satisfy it either.

Example 5.6 **Pruning shared dimensions.** If the value in the shared dimension *A* is a_1 and it fails to satisfy the iceberg condition, then the whole subtree rooted at a_1CD/a_1 (including a_1C/a_1C, a_1D/a_1, a_1/a_1) can be pruned because they are all more specialized versions of a_1. ■

To explain how the Star-Cubing algorithm works, we need to explain a few more concepts, namely, *cuboid trees*, *star-nodes*, and *star-trees*.

We use trees to represent individual cuboids. Figure 5.9 shows a fragment of the **cuboid tree** of the base cuboid, *ABCD*. Each level in the tree represents a dimension, and each node represents an attribute value. Each node has four fields: the attribute value, aggregate value, pointer to possible first child, and pointer to possible first sibling. Tuples in the cuboid are inserted one by one into the tree. A path from the root to a leaf node represents a tuple. For example, node c_2 in the tree has an aggregate (count) value of 5,

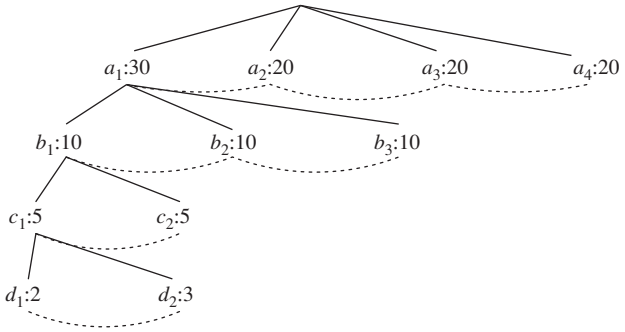


Figure 5.9 Base cuboid tree fragment.

which indicates that there are five cells of value $(a_1, b_1, c_2, *)$. This representation collapses the common prefixes to save memory usage and allows us to aggregate the values at internal nodes. With aggregate values at internal nodes, we can prune based on shared dimensions. For example, the *AB* cuboid tree can be used to prune possible cells in *ABD*.

If the single-dimensional aggregate on an attribute value *p* does not satisfy the iceberg condition, it is useless to distinguish such nodes in the iceberg cube computation. Thus, the node *p* can be replaced by $*$ so that the cuboid tree can be further compressed. We say that the node *p* in an attribute *A* is a **star-node** if the single-dimensional aggregate on *p* does not satisfy the iceberg condition; otherwise, *p* is a *non-star-node*. A cuboid tree that is compressed using star-nodes is called a **star-tree**.

Example 5.7 **Star-tree construction.** A base cuboid table is shown in Table 5.1. There are five tuples and four dimensions. The cardinalities for dimensions *A*, *B*, *C*, *D* are 2, 4, 4, 4, respectively. The one-dimensional aggregates for all attributes are shown in Table 5.2. Suppose $min\_sup = 2$ in the iceberg condition. Clearly, only attribute values a_1, a_2, b_1, c_3, d_4 satisfy the condition. All other values are below the threshold and thus become star-nodes. By collapsing star-nodes, the reduced base table is Table 5.3. Notice that the table contains two fewer rows and also fewer distinct values than Table 5.1.

Table 5.1 Base (Cuboid) Table: Before Star Reduction

A	*B*	*C*	*D*	count
a_1	b_1	c_1	d_1	1
a_1	b_1	c_4	d_3	1
a_1	b_2	c_2	d_2	1
a_2	b_3	c_3	d_4	1
a_2	b_4	c_3	d_4	1

Table 5.2 One-Dimensional Aggregates

Dimension	count = 1	count ≥ 2
A	—	$a_1(3)$, $a_2(2)$
B	b_2, b_3, b_4	$b_1(2)$
C	c_1, c_2, c_4	$c_3(2)$
D	d_1, d_2, d_3	$d_4(2)$

Table 5.3 Compressed Base Table: After Star Reduction

A	*B*	*C*	*D*	count
a_1	b_1	*	*	2
a_1	*	*	*	1
a_2	*	c_3	d_4	2

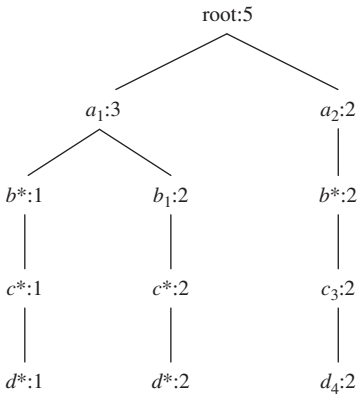


Figure 5.10 Compressed base table star-tree.

We use the reduced base table to construct the cuboid tree because it is smaller. The resultant star-tree is shown in Figure 5.10. ■

Now, let's see how the Star-Cubing algorithm uses star-trees to compute an iceberg cube. The algorithm is given later in Figure 5.13.

Example 5.8 **Star-Cubing.** Using the star-tree generated in Example 5.7 (Figure 5.10), we start the aggregation process by traversing in a bottom-up fashion. Traversal is depth-first. The first stage (i.e., the processing of the first branch of the tree) is shown in Figure 5.11. The leftmost tree in the figure is the base star-tree. Each attribute value is shown with its corresponding aggregate value. In addition, subscripts by the nodes in the tree show the

traversal order. The remaining four trees are *BCD*, *ACD*/*A*, *ABD*/*AB*, and *ABC*/*ABC*. They are the child trees of the base star-tree, and correspond to the level of 3-D cuboids above the base cuboid in Figure 5.8. The subscripts in them correspond to the same subscripts in the base tree—they denote the step or order in which they are created during the tree traversal. For example, when the algorithm is at step 1, the *BCD* child tree root is created. At step 2, the *ACD*/*A* child tree root is created. At step 3, the *ABD*/*AB* tree root and the $b*$ node in *BCD* are created.

When the algorithm has reached step 5, the trees in memory are exactly as shown in Figure 5.11. Because depth-first traversal has reached a leaf at this point, it starts backtracking. Before traversing back, the algorithm notices that all possible nodes in the base dimension (*ABC*) have been visited. This means the *ABC*/*ABC* tree is complete, so the count is output and the tree is destroyed. Similarly, upon moving back from $d*$ to $c*$ and seeing that $c*$ has no siblings, the count in *ABD*/*AB* is also output and the tree is destroyed.

When the algorithm is at $b*$ during the backtraversal, it notices that there exists a sibling in b_1. Therefore, it will keep *ACD*/*A* in memory and perform a depth-first search

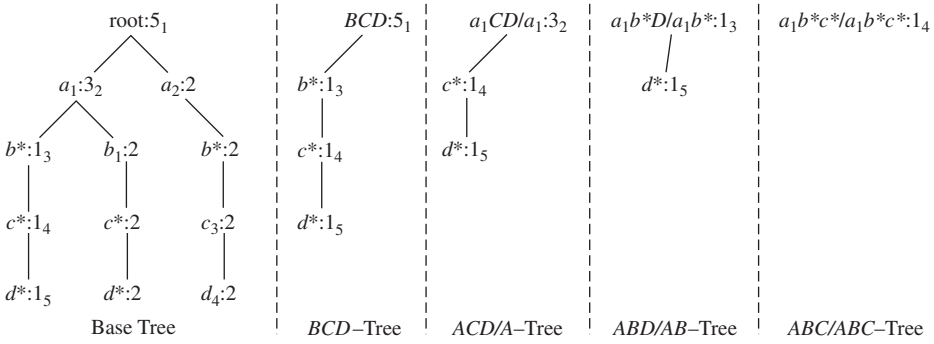


Figure 5.11 Aggregation stage one: processing the leftmost branch of the base tree.

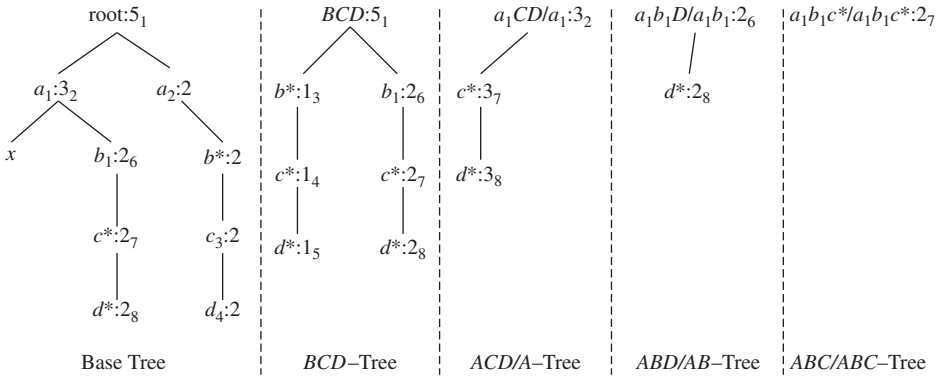


Figure 5.12 Aggregation stage two: processing the second branch of the base tree.

Algorithm: Star-Cubing. Compute iceberg cubes by Star-Cubing.

Input:

- *R*: a relational table
- *min_support*: minimum support threshold for the iceberg condition (taking count as the measure).

Output: The computed iceberg cube.

Method: Each star-tree corresponds to one cuboid tree node, and vice versa.

```
BEGIN
    scan R twice, create star-table S and star-tree T;
    output count of T.root;
  call starcubing(T, T.root);
END

procedure starcubing(T, cnode)// cnode: current node
{
(1)  for each non-null child C of T's cuboid tree
(2)       insert or aggregate cnode to the corresponding
              position or node in C's star-tree;
(3)  if (cnode.count ≥ min_support) then {
(4)       if (cnode ≠ root) then
(5)            output cnode.count;
(6)       if (cnode is a leaf) then
(7)            output cnode.count;
(8)       else { // initiate a new cuboid tree
(9)            create C_C as a child of T's cuboid tree;
(10)           let T_C be C_C's star-tree;
(11)           T_C.root's count = cnode.count;
(12)      }
(13) }
(14) if (cnode is not a leaf) then
(15)      starcubing(T, cnode.first_child);
(16) if (C_C is not null) then {
(17)      starcubing(T_C, T_C.root);
(18)      remove C_C from T's cuboid tree; }
(19) if (cnode has sibling) then
(20)      starcubing(T, cnode.sibling);
(21) remove T;
}
```

Figure 5.13 Star-Cubing algorithm.

on b_1 just as it did on $b*$. This traversal and the resultant trees are shown in Figure 5.12. The child trees ACD/A and ABD/AB are created again but now with the new values from the b_1 subtree. For example, notice that the aggregate count of $c*$ in the ACD/A tree has increased from 1 to 3. The trees that remained intact during the last traversal are reused and the new aggregate values are added on. For instance, another branch is added to the *BCD* tree.

Just like before, the algorithm will reach a leaf node at $d*$ and traverse back. This time, it will reach a_1 and notice that there exists a sibling in a_2. In this case, all child trees except *BCD* in Figure 5.12 are destroyed. Afterward, the algorithm will perform the same traversal on a_2. *BCD* continues to grow while the other subtrees start fresh with a_2 instead of a_1. ■

A node must satisfy two conditions in order to generate child trees: (1) the measure of the node must satisfy the iceberg condition; and (2) the tree to be generated must include at least one non-star-node (i.e., nontrivial). This is because if all the nodes were star-nodes, then none of them would satisfy *min_sup*. Therefore, it would be a complete waste to compute them. This pruning is observed in Figures 5.11 and 5.12. For example, the left subtree extending from node a_1 in the base tree in Figure 5.11 does not include any nonstar-nodes. Therefore, the a_1CD/a_1 subtree should not have been generated. It is shown, however, for illustration of the child tree generation process.

Star-Cubing is sensitive to the ordering of dimensions, as with other iceberg cube construction algorithms. For best performance, the dimensions are processed in order of decreasing cardinality. This leads to a better chance of early pruning, because the higher the cardinality, the smaller the partitions, and therefore the higher possibility that the partition will be pruned.

Star-Cubing can also be used for full cube computation. When computing the full cube for a dense data set, Star-Cubing's performance is comparable with MultiWay and is much faster than BUC. If the data set is sparse, Star-Cubing is significantly faster than MultiWay and faster than BUC, in most cases. For iceberg cube computation, Star-Cubing is faster than BUC, where the data are skewed and the speed-up factor increases as *min_sup* decreases.

5.2.4 Precomputing Shell Fragments for Fast High-Dimensional OLAP

Recall the reason that we are interested in precomputing data cubes: Data cubes facilitate fast OLAP in a multidimensional data space. However, a full data cube of high dimensionality needs massive storage space and unrealistic computation time. Iceberg cubes provide a more feasible alternative, as we have seen, wherein the iceberg condition is used to specify the computation of only a subset of the full cube's cells. However, although an iceberg cube is smaller and requires less computation time than its corresponding full cube, it is not an ultimate solution.

For one, the computation and storage of the iceberg cube can still be costly. For example, if the base cuboid cell, $(a_1, a_2, \ldots, a_{60})$, passes minimum support (or the iceberg

threshold), it will generate 2^{60} iceberg cube cells. Second, it is difficult to determine an appropriate iceberg threshold. Setting the threshold too low will result in a huge cube, whereas setting the threshold too high may invalidate many useful applications. Third, an iceberg cube cannot be incrementally updated. Once an aggregate cell falls below the iceberg threshold and is pruned, its measure value is lost. Any incremental update would require recomputing the cells from scratch. This is extremely undesirable for large real-life applications where incremental appending of new data is the norm.

One possible solution, which has been implemented in some commercial data warehouse systems, is to compute a thin **cube shell**. For example, we could compute all cuboids with three dimensions or less in a 60-dimensional data cube, resulting in a cube shell of size 3. The resulting cuboids set would require much less computation and storage than the full 60-dimensional data cube. However, there are two disadvantages to this approach. First, we would still need to compute $\binom{60}{3} + \binom{60}{2} + 60 = 36{,}050$ cuboids, each with many cells. Second, such a cube shell does not support high-dimensional OLAP because (1) it does not support OLAP on four or more dimensions, and (2) it cannot even support drilling along three dimensions, such as, say, (A_4, A_5, A_6), *on a subset of data* selected based on the constants provided in three *other* dimensions, such as (A_1, A_2, A_3), because this essentially requires the computation of the corresponding 6-D cuboid. (Notice that there is no cell in cuboid (A_4, A_5, A_6) computed for any particular constant set, such as (a_1, a_2, a_3), associated with dimensions (A_1, A_2, A_3).)

Instead of computing a cube shell, we can compute only portions or fragments of it. This section discusses the *shell fragment* approach for OLAP query processing. It is based on the following key observation about OLAP in high-dimensional space. Although a data cube may contain many dimensions, *most OLAP operations are performed on only a small number of dimensions at a time.* In other words, an OLAP query is likely to ignore many dimensions (i.e., treating them as irrelevant), fix some dimensions (e.g., using query constants as instantiations), and leave only a few to be manipulated (for drilling, pivoting, etc.). This is because it is neither realistic nor fruitful for anyone to comprehend the changes of thousands of cells involving tens of dimensions simultaneously in a high-dimensional space at the same time.

Instead, it is more natural to first locate some cuboids of interest and then drill along one or two dimensions to examine the changes of a few related dimensions. Most analysts will only need to examine, at any one moment, the combinations of a small number of dimensions. This implies that if multidimensional aggregates can be computed quickly on a *small number of dimensions inside a high-dimensional space*, we may still achieve fast OLAP without materializing the original high-dimensional data cube. Computing the full cube (or, often, even an iceberg cube or cube shell) can be excessive. Instead, a *semi-online computation model with certain preprocessing* may offer a more feasible solution. Given a base cuboid, some quick preparation computation can be done first (i.e., offline). After that, a query can then be computed online using the preprocessed data.

The shell fragment approach follows such a semi-online computation strategy. It involves two algorithms: one for computing cube shell fragments and the other for query processing with the cube fragments. The shell fragment approach can handle databases

Table 5.4 Original Database

TID	*A*	*B*	*C*	*D*	*E*
1	a_1	b_1	c_1	d_1	e_1
2	a_1	b_2	c_1	d_2	e_1
3	a_1	b_2	c_1	d_1	e_2
4	a_2	b_1	c_1	d_1	e_2
5	a_2	b_1	c_1	d_1	e_3

of high dimensionality and can quickly compute small local cubes online. It explores the *inverted index* data structure, which is popular in information retrieval and Web-based information systems.

The basic idea is as follows. Given a high-dimensional data set, we partition the dimensions into a set of disjoint dimension *fragments*, convert each fragment into its corresponding inverted index representation, and then construct *cube shell fragments* while keeping the inverted indices associated with the cube cells. Using the precomputed cubes' shell fragments, we can dynamically assemble and compute cuboid cells of the required data cube online. This is made efficient by set intersection operations on the inverted indices.

To illustrate the shell fragment approach, we use the tiny database of Table 5.4 as a running example. Let the cube measure be count(). Other measures will be discussed later. We first look at how to construct the inverted index for the given database.

Example 5.9 **Construct the inverted index.** For each attribute value in each dimension, list the tuple identifiers (*TIDs*) of all the tuples that have that value. For example, attribute value a_2 appears in tuples 4 and 5. The TID list for a_2 then contains exactly two items, namely 4 and 5. The resulting inverted index table is shown in Table 5.5. It retains all the original database's information. If each table entry takes one unit of memory, Tables 5.4 and 5.5 each takes 25 units, that is, the inverted index table uses the same amount of memory as the original database. ■

"*How do we compute shell fragments of a data cube?*" The shell fragment computation algorithm, **Frag-Shells**, is summarized in Figure 5.14. We first partition all the dimensions of the given data set into independent groups of dimensions, called *fragments* (line 1). We scan the base cuboid and construct an inverted index for each attribute (lines 2 to 6). Line 3 is for when the measure is other than the tuple count(), which will be described later. For each fragment, we compute the full *local* (i.e., fragment-based) data cube while retaining the inverted indices (lines 7 to 8). Consider a database of 60 dimensions, namely, $A_1, A_2, \ldots, A_{60}$. We can first partition the 60 dimensions into 20 fragments of size 3: (A_1, A_2, A_3), (A_4, A_5, A_6), …, (A_{58}, A_{59}, A_{60}). For each fragment, we compute its full data cube while recording the inverted indices. For example, in fragment (A_1, A_2, A_3), we would compute seven cuboids: A_1, A_2, A_3, A_1A_2, A_2A_3, A_1A_3, $A_1A_2A_3$. Furthermore, an inverted index

Table 5.5 Inverted Index

Attribute Value	*TID List*	*List Size*
a_1	{1, 2, 3}	3
a_2	{4, 5}	2
b_1	{1, 4, 5}	3
b_2	{2, 3}	2
c_1	{1, 2, 3, 4, 5}	5
d_1	{1, 3, 4, 5}	4
d_2	{2}	1
e_1	{1, 2}	2
e_2	{3, 4}	2
e_3	{5}	1

Algorithm: Frag-Shells. Compute shell fragments on a given high-dimensional base table (i.e., base cuboid).

Input: A base cuboid, B, of n dimensions, namely, $(A_1, \ldots, A_n)$.

Output:

- a set of fragment partitions, $\{P_1, \ldots, P_k\}$, and their corresponding (local) fragment cubes, $\{S_1, \ldots, S_k\}$, where P_i represents some set of dimension(s) and $P_1 \cup \ldots \cup P_k$ make up all the n dimensions
- an *ID_measure* array if the measure is not the tuple count, count()

Method:

(1) partition the set of dimensions $(A_1, \ldots, A_n)$ into
    a set of k fragments $P_1, \ldots, P_k$ (based on data & query distribution)
(2) scan base cuboid, B, once and do the following {
(3) insert each ⟨*TID, measure*⟩ into *ID_measure* array
(4) for each attribute value a_j of each dimension A_i
(5) build an inverted index entry: ⟨a_j, *TIDlist*⟩
(6) }
(7) for each fragment partition P_i
(8) build a local fragment cube, S_i, by intersecting their
     corresponding TIDlists and computing their measures

Figure 5.14 Shell fragment computation algorithm.

is retained for each cell in the cuboids. That is, for each cell, its associated TID list is recorded.

The benefit of computing local cubes of each shell fragment instead of computing the complete cube shell can be seen by a simple calculation. For a base cuboid of

60 dimensions, there are only $7 \times 20 = 140$ cuboids to be computed according to the preceding shell fragment partitioning. This is in contrast to the 36,050 cuboids computed for the cube shell of size 3 described earlier! Notice that the above fragment partitioning is based simply on the grouping of consecutive dimensions. A more desirable approach would be to partition based on popular dimension groupings. This information can be obtained from domain experts or the past history of OLAP queries.

Let's return to our running example to see how shell fragments are computed.

Example 5.10 **Compute shell fragments.** Suppose we are to compute the shell fragments of size 3. We first divide the five dimensions into two fragments, namely (A, B, C) and (D, E). For each fragment, we compute the full local data cube by intersecting the TID lists in Table 5.5 in a top-down depth-first order in the cuboid lattice. For example, to compute the cell $(a_1, b_2, *)$, we intersect the TID lists of a_1 and b_2 to obtain a new list of $\{2, 3\}$. Cuboid AB is shown in Table 5.6.

After computing cuboid AB, we can then compute cuboid ABC by intersecting all pairwise combinations between Table 5.6 and the row c_1 in Table 5.5. Notice that because cell (a_2, b_2) is empty, it can be effectively discarded in subsequent computations, based on the Apriori property. The same process can be applied to compute fragment (D, E), which is completely independent from computing (A, B, C). Cuboid DE is shown in Table 5.7. ■

If the measure in the iceberg condition is **count()** (as in tuple counting), there is no need to reference the original database for this because the *length* of the TID list is equivalent to the tuple count. "*Do we need to reference the original database if computing other measures such as* ***average()****?*" Actually, we can build and reference an *ID_measure*

Table 5.6 Cuboid *AB*

Cell	*Intersection*	*TID List*	*List Size*
(a_1, b_1)	$\{1, 2, 3\} \cap \{1, 4, 5\}$	$\{1\}$	1
(a_1, b_2)	$\{1, 2, 3\} \cap \{2, 3\}$	$\{2, 3\}$	2
(a_2, b_1)	$\{4, 5\} \cap \{1, 4, 5\}$	$\{4, 5\}$	2
(a_2, b_2)	$\{4, 5\} \cap \{2, 3\}$	$\{\}$	0

Table 5.7 Cuboid *DE*

Cell	*Intersection*	*TID List*	*List Size*
(d_1, e_1)	$\{1, 3, 4, 5\} \cap \{1, 2\}$	$\{1\}$	1
(d_1, e_2)	$\{1, 3, 4, 5\} \cap \{3, 4\}$	$\{3, 4\}$	2
(d_1, e_3)	$\{1, 3, 4, 5\} \cap \{5\}$	$\{5\}$	1
(d_2, e_1)	$\{2\} \cap \{1, 2\}$	$\{2\}$	1

array instead, which stores what we need to compute other measures. For example, to compute **average()**, we let the *ID_measure* array hold three elements, namely, (*TID*, **item_count**, **sum**), for each cell (line 3 of the shell fragment computation algorithm in Figure 5.14). The **average()** measure for each aggregate cell can then be computed by accessing only this *ID_measure* array, using **sum()/item_count()**. Considering a database with 10^6 tuples, each taking 4 bytes each for *TID*, **item_count**, and **sum**, the *ID_measure* array requires 12 MB, whereas the corresponding database of 60 dimensions will require $(60 + 3) \times 4 \times 10^6 = 252$ MB (assuming each attribute value takes 4 bytes). Obviously, *ID_measure* array is a more compact data structure and is more likely to fit in memory than the corresponding high-dimensional database.

To illustrate the design of the *ID_measure* array, let's look at Example 5.11.

Example 5.11 Computing cubes with the average() measure. Table 5.8 shows an example sales database where each tuple has two associated values, such as **item_count** and **sum**, where **item_count** is the count of items sold.

To compute a data cube for this database with the measure **average()**, we need to have a TID list for each cell: $\{TID_1, \ldots, TID_n\}$. Because each TID is uniquely associated with a particular set of measure values, all future computation just needs to fetch the measure values associated with the tuples in the list. In other words, by keeping an *ID_measure* array in memory for online processing, we can handle complex algebraic measures, such as average, variance, and standard deviation. Table 5.9 shows what exactly should be kept for our example, which is substantially smaller than the database itself. ■

Table 5.8 Database with Two Measure Values

TID	*A*	*B*	*C*	*D*	*E*	*item_count*	*sum*
1	a_1	b_1	c_1	d_1	e_1	5	70
2	a_1	b_2	c_1	d_2	e_1	3	10
3	a_1	b_2	c_1	d_1	e_2	8	20
4	a_2	b_1	c_1	d_1	e_2	5	40
5	a_2	b_1	c_1	d_1	e_3	2	30

Table 5.9 Table 5.8 *ID_measure* Array

TID	*item_count*	*sum*
1	5	70
2	3	10
3	8	20
4	5	40
5	2	30

The shell fragments are negligible in both storage space and computation time in comparison with the full data cube. Note that we can also use the Frag-Shells algorithm to compute the full data cube by including all the dimensions as a single fragment. Because the order of computation with respect to the cuboid lattice is top-down and depth-first (similar to that of BUC), the algorithm can perform Apriori pruning if applied to the construction of iceberg cubes.

"Once we have computed the shell fragments, how can they be used to answer OLAP queries?" Given the precomputed shell fragments, we can view the cube space as a virtual cube and perform OLAP queries related to the cube online. In general, two types of queries are possible: (1) *point query* and (2) *subcube query*.

In a **point query**, all of the *relevant* dimensions in the cube have been instantiated (i.e., there are no *inquired* dimensions in the relevant dimensions set). For example, in an n-dimensional data cube, $A_1A_2 \ldots A_n$, a point query could be in the form of $\langle A_1, A_5, A_9 : M? \rangle$, where $A_1 = \{a_{11}, a_{18}\}$, $A_5 = \{a_{52}, a_{55}, a_{59}\}$, $A_9 = a_{94}$, and M is the inquired measure for each corresponding cube cell. For a cube with a small number of dimensions, we can use $*$ to represent a "don't care" position where the corresponding dimension is *irrelevant*, that is, neither inquired nor instantiated. For example, in the query $\langle a_2, b_1, c_1, d_1, * : \text{count}() ? \rangle$ for the database in Table 5.4, the first four dimension values are instantiated to a_2, b_1, c_1, and d_1, respectively, while the last dimension is irrelevant, and count() (which is the tuple count by context) is the inquired measure.

In a **subcube query**, at least one of the *relevant* dimensions in the cube is *inquired*. For example, in an n-dimensional data cube $A_1A_2 \ldots A_n$, a subcube query could be in the form $\langle A_1, A_5?, A_9, A_{21}? : M? \rangle$, where $A_1 = \{a_{11}, a_{18}\}$ and $A_9 = a_{94}$, A_5 and A_{21} are the inquired dimensions, and M is the inquired measure. For a cube with a small number of dimensions, we can use $*$ for an irrelevant dimension and ? for an inquired one. For example, in the query $\langle a_2, ?, c_1, *, ? : \text{count}() ? \rangle$ we see that the first and third dimension values are instantiated to a_2 and c_1, respectively, while the fourth is irrelevant, and the second and the fifth are inquired. *A subcube query computes all possible value combinations of the inquired dimensions.* It essentially returns a local data cube consisting of the inquired dimensions.

"How can we use shell fragments to answer a point query?" Because a point query explicitly provides the instantiated variables set on the relevant dimensions set, we can make maximal use of the precomputed shell fragments by finding the *best fitting* (i.e., *dimension-wise completely matching*) fragments to fetch and intersect the associated TID lists.

Let the point query be of the form $\langle \alpha_i, \alpha_j, \alpha_k, \alpha_p : M? \rangle$, where α_i represents a set of instantiated values of dimension A_i, and so on for α_j, α_k, and α_p. First, we check the shell fragment schema to determine which dimensions among A_i, A_j, A_k, and A_p are in the same fragment(s). Suppose A_i and A_j are in the same fragment, while A_k and A_p are in two other fragments. We fetch the corresponding TID lists on the precomputed 2-D fragment for dimensions A_i and A_j using the instantiations α_i and α_j, and fetch the TID lists on the 1-D fragments for dimensions A_k and A_p using the instantiations α_k and α_p, respectively. The obtained TID lists are intersected to derive the TID list table. This table is then used to derive the specified measure (e.g., by taking the length of the TID lists

for tuple count(), or by fetching item_count() and sum() from the *ID_measure* array to compute average()) for the final set of cells.

Example 5.12 **Point query.** Suppose a user wants to compute the point query $\langle a_2, b_1, c_1, d_1, *: \text{count}()?\rangle$ for our database in Table 5.4 and that the shell fragments for the partitions (*A*, *B*, *C*) and (*D*, *E*) are precomputed as described in Example 5.10. The query is broken down into two subqueries based on the precomputed fragments: $\langle a_2, b_1, c_1, *, *\rangle$ and $\langle *, *, *, d_1, *\rangle$. The best-fit precomputed shell fragments for the two subqueries are *ABC* and *D*. The fetch of the TID lists for the two subqueries returns two lists: {4, 5} and {1, 3, 4, 5}. Their intersection is the list {4, 5}, which is of size 2. Thus, the final answer is count() = 2. ■

"How can we use shell fragments to answer a subcube query?" A subcube query returns a local data cube based on the instantiated and inquired dimensions. Such a data cube needs to be aggregated in a multidimensional way so that online analytical processing (drilling, dicing, pivoting, etc.) can be made available to users for flexible manipulation and analysis. Because instantiated dimensions usually provide highly selective constants that dramatically reduce the size of the valid TID lists, we should make maximal use of the precomputed shell fragments by finding the fragments that best fit the set of instantiated dimensions, and fetching and intersecting the associated TID lists to derive the reduced TID list. This list can then be used to intersect the best-fitting shell fragments consisting of the inquired dimensions. This will generate the relevant and inquired base cuboid, which can then be used to compute the relevant subcube on-the-fly using an efficient online cubing algorithm.

Let the subcube query be of the form $\langle \alpha_i, \alpha_j, A_k?, \alpha_p, A_q? : M?\rangle$, where α_i, α_j, and α_p represent a set of instantiated values of dimension A_i, A_j, and A_p, respectively, and A_k and A_q represent two inquired dimensions. First, we check the shell fragment schema to determine which dimensions among (1) A_i, A_j, and A_p, and (2) A_k and A_q are in the same fragment partition. Suppose A_i and A_j belong to the same fragment, as do A_k and A_q, but that A_p is in a different fragment. We fetch the corresponding TID lists in the precomputed 2-D fragment for A_i and A_j using the instantiations α_i and α_j, then fetch the TID list on the precomputed 1-D fragment for A_p using instantiation α_p, and then fetch the TID lists on the precomputed 2-D fragments for A_k and A_q, respectively, using no instantiations (i.e., all possible values). The obtained TID lists are intersected to derive the final TID lists, which are used to fetch the corresponding measures from the *ID_measure* array to derive the "base cuboid" of a 2-D subcube for two dimensions (A_k, A_q). A fast cube computation algorithm can be applied to compute this 2-D cube based on the derived base cuboid. The computed 2-D cube is then ready for OLAP operations.

Example 5.13 **Subcube query.** Suppose that a user wants to compute the subcube query, $\langle a_2, b_1, ?, *, ? : \text{count}()?\rangle$, for our database shown earlier in Table 5.4, and that the shell fragments have been precomputed as described in Example 5.10. The query can be broken into three best-fit fragments according to the instantiated and inquired dimensions: *AB*, *C*,

and E, where AB has the instantiation (a_2, b_1). The fetch of the TID lists for these partitions returns $(a_2, b_1) : \{4, 5\}$, $(c_1) : \{1, 2, 3, 4, 5\}$ and $\{(e_1 : \{1, 2\}), (e_2 : \{3, 4\}), (e_3 : \{5\})\}$, respectively. The intersection of these corresponding TID lists contains a cuboid with two tuples: $\{(c_1, e_2) : \{4\},$[5] $(c_1, e_3) : \{5\}\}$. This base cuboid can be used to compute the 2-D data cube, which is trivial. ■

For large data sets, a fragment size of 2 or 3 typically results in reasonable storage requirements for the shell fragments and for fast query response time. Querying with shell fragments is substantially faster than answering queries using precomputed data cubes that are stored on disk. In comparison to full cube computation, Frag-Shells is recommended if there are less than four inquired dimensions. Otherwise, more efficient algorithms, such as Star-Cubing, can be used for fast online cube computation. Frag-Shells can be easily extended to allow incremental updates, the details of which are left as an exercise.

5.3 Processing Advanced Kinds of Queries by Exploring Cube Technology

Data cubes are not confined to the simple multidimensional structure illustrated in the last section for typical business data warehouse applications. The methods described in this section further develop data cube technology for effective processing of advanced kinds of queries. Section 5.3.1 explores *sampling cubes*. This extension of data cube technology can be used to answer queries on *sample data*, such as survey data, which represent a sample or subset of a target data population of interest. Section 5.3.2 explains how *ranking cubes* can be computed to answer top-k queries, such as "find the top 5 cars," according to some user-specified criteria.

The basic data cube structure has been further extended for various sophisticated data types and new applications. Here we list some examples, such as *spatial data cubes* for the design and implementation of *geospatial data warehouses*, and *multimedia data cubes* for the multidimensional analysis of *multimedia data* (those containing images and videos). *RFID data cubes* handle the compression and multidimensional analysis of *RFID* (i.e., radio-frequency identification) data. *Text cubes* and *topic cubes* were developed for the application of vector-space models and generative language models, respectively, in the analysis of *multidimensional text databases* (which contain both structure attributes and narrative text attributes).

5.3.1 Sampling Cubes: OLAP-Based Mining on Sampling Data

When collecting data, we often collect only a *subset* of the data we would ideally like to gather. In statistics, this is known as collecting a **sample** of the data population.

[5] That is, the intersection of the TID lists for (a_2, b_1), (c_1), and (e_2) is $\{4\}$.

The resulting data are called **sample data**. Data are often sampled to save on costs, manpower, time, and materials. In many applications, the collection of the entire data population of interest is unrealistic. In the study of TV ratings or pre-election polls, for example, it is impossible to gather the opinion of *everyone* in the population. Most published ratings or polls rely on a data sample for analysis. The results are extrapolated for the entire population, and associated with certain statistical measures such as a *confidence interval*. The confidence interval tells us how reliable a result is. Statistical surveys based on sampling are a common tool in many fields like politics, healthcare, market research, and social and natural sciences.

"*How effective is OLAP on sample data*?" OLAP traditionally has the full data population on hand, yet with sample data, we have only a small subset. If we try to apply traditional OLAP tools to sample data, we encounter three challenges. First, sample data are often sparse in the multidimensional sense. When a user drills down on the data, it is easy to reach a point with very few or no samples even when the overall sample size is large. Traditional OLAP simply uses whatever data are available to compute a query answer. To extrapolate such an answer for a population based on a small sample could be misleading: A single outlier or a slight bias in the sampling can distort the answer significantly. Second, with sample data, statistical methods are used to provide a measure of reliability (e.g., a confidence interval) to indicate the quality of the query answer as it pertains to the population. Traditional OLAP is not equipped with such tools.

A *sampling cube* framework was introduced to tackle each of the preceding challenges.

Sampling Cube Framework

The **sampling cube** is a data cube structure that stores the sample data and their multidimensional aggregates. It supports OLAP on sample data. It calculates confidence intervals as a quality measure for any multidimensional query. Given a sample data relation (i.e., base cuboid) R, the sampling cube C_R typically computes the sample mean, sample standard deviation, and other task-specific measures.

In statistics, a *confidence interval* is used to indicate the reliability of an estimate. Suppose we want to estimate the mean age of all viewers of a given TV show. We have sample data (a subset) of this data population. Let's say our sample mean is 35 years. This becomes our estimate for the entire population of viewers as well, but how confident can we be that 35 is also the mean of the true population? It is unlikely that the sample mean will be exactly equal to the true population mean because of sampling error. Therefore, we need to qualify our estimate in some way to indicate the general magnitude of this error. This is typically done by computing a **confidence interval**, which is an *estimated value range with a given high probability of covering the true population value.* A confidence interval for our example could be "*the actual mean will not vary by +/− two standard deviations 95% of the time.*" (Recall that the standard deviation is just a number, which can be computed as shown in Section 2.2.2.) A confidence interval is always qualified by a particular *confidence level.* In our example, it is 95%.

The confidence interval is calculated as follows. Let x be a set of samples. The mean of the samples is denoted by $\bar{x}$, and the number of samples in x is denoted by l. Assuming

that the standard deviation of the population is unknown, the *sample* standard deviation of x is denoted by s. Given a desired confidence level, the **confidence interval** for $\bar{x}$ is

$$\bar{x} \pm t_c \hat{\sigma}_{\bar{x}}, \tag{5.1}$$

where t_c is the *critical t-value* associated with the confidence level and $\hat{\sigma}_{\bar{x}} = \frac{s}{\sqrt{l}}$ is the *estimated standard error of the mean*. To find the appropriate t_c, specify the desired confidence level (e.g., 95%) and also the *degree of freedom*, which is just $l - 1$.

The important thing to note is that the computation involved in computing a confidence interval is *algebraic*. Let's look at the three terms involved in Eq. (5.1). The first is the mean of the sample set, $\bar{x}$, which is algebraic; the second is the critical t-value, which is calculated by a lookup, and with respect to x, it depends on l, a distributive measure; and the third is $\hat{\sigma}_{\bar{x}} = \frac{s}{\sqrt{l}}$, which also turns out to be algebraic if one records the linear sum ($\sum_{i=1}^{l} x_i$) and squared sum ($\sum_{i=1}^{l} x_i^2$). Because the terms involved are either algebraic or distributive, the confidence interval computation is algebraic. Actually, since both the mean and confidence interval are algebraic, at every cell, exactly three values are sufficient to calculate them—all of which are either distributive or algebraic:

1. l
2. $sum = \sum_{i=1}^{l} x_i$
3. $squared\ sum = \sum_{i=1}^{l} x_i^2$

There are many efficient techniques for computing algebraic and distributive measures (Section 4.2.4). Therefore, any of the previously developed cubing algorithms can be used to efficiently construct a sampling cube.

Now that we have established that sampling cubes can be computed efficiently, our next step is to find a way of boosting the confidence of results obtained for queries on sample data.

Query Processing: Boosting Confidences for Small Samples

A query posed against a data cube can be either a *point query* or a *range query*. Without loss of generality, consider the case of a point query. Here, it corresponds to a cell in sampling cube C_R. The goal is to provide an accurate point estimate for the samples in that cell. Because the cube also reports the confidence interval associated with the sample mean, there is some measure of "reliability" to the returned answer. If the confidence interval is small, the reliability is deemed good; however, if the interval is large, the reliability is questionable.

"*What can we do to boost the reliability of query answers?*" Consider what affects the confidence interval size. There are two main factors: the variance of the sample data and the sample size. First, a rather large variance in the cell may indicate that the chosen cube

cell is poor for prediction. A better solution is probably to drill down on the query cell to a more specific one (i.e., asking more specific queries). Second, a small sample size can cause a large confidence interval. When there are very few samples, the corresponding t_c is large because of the small degree of freedom. This in turn could cause a large confidence interval. Intuitively, this makes sense. Suppose one is trying to figure out the average income of people in the United States. Just asking two or three people does not give much confidence to the returned response.

The best way to solve this small sample size problem is to get more data. Fortunately, there is usually an abundance of additional data available in the cube. The data do not match the query cell exactly; however, we can consider data from cells that are "close by." There are two ways to incorporate such data to enhance the reliability of the query answer: (1) *intracuboid query expansion*, where we consider nearby cells *within* the same cuboid, and (2) *intercuboid query expansion*, where we consider more general versions (from parent cuboids) of the query cell. Let's see how this works, starting with intracuboid query expansion.

Method 1. Intracuboid query expansion. Here, we expand the sample size by including nearby cells in the *same* cuboid as the queried cell, as shown in Figure 5.15(a). We just have to be careful that the new samples serve to increase the confidence in the answer without changing the query's semantics.

So, the first question is "*Which dimensions should be expanded?*" The best candidates should be the dimensions that are *uncorrelated* or *weakly correlated* with the measure

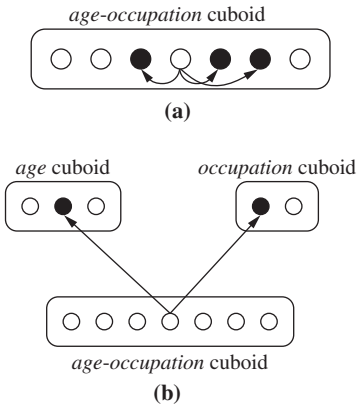


Figure 5.15 Query expansion within sampling cube: Given small data samples, both methods use strategies to boost the reliability of query answers by considering additional data cell values. (a) Intracuboid expansion considers nearby cells in the same cuboid as the queried cell. (b) Intercuboid expansion considers more general cells from parent cuboids.

value (i.e., the value to be predicted). Expanding within these dimensions will likely increase the sample size and not shift the query's answer. Consider an example of a 2-D query specifying *education* = "college" and *birth_month* = "July." Let the cube measure be *average income*. Intuitively, education has a high correlation to income while birth month does not. It would be harmful to expand the *education* dimension to include values such as "graduate" or "high school." They are likely to alter the final result. However, expansion in the *birth_month* dimension to include other month values could be helpful, because it is unlikely to change the result but will increase sampling size.

To mathematically measure the correlation of a dimension to the cube value, the correlation between the dimension's values and their aggregated cube measures is computed. *Pearson's correlation coefficient* for numeric data and the χ^2 correlation test for nominal data are popularly used correlation measures, although many other measures, such as *covariance*, can be used. (These measures were presented in Section 3.3.2.) A dimension that is strongly correlated with the value to be predicted should *not* be a candidate for expansion. Notice that since the correlation of a dimension with the cube measure is independent of a particular query, it should be precomputed and stored with the cube measure to facilitate efficient online analysis.

After selecting dimensions for expansion, the next question is "*Which values within these dimensions should the expansion use?*" This relies on the semantic knowledge of the dimensions in question. The goal should be to select semantically similar values to minimize the risk of altering the final result. Consider the *age* dimension—similarity of values in this dimension is clear. There is a definite (numeric) order to the values. Dimensions with numeric or ordinal (ranked) data (like *education*) have a definite ordering among data values. Therefore, we can select values that are close to the instantiated query value. For nominal data of a dimension that is organized in a multilevel hierarchy in a data cube (e.g., *location*), we should select those values located in the same branch of the tree (e.g., the same district or city).

By considering additional data during query expansion, we are aiming for a more accurate and reliable answer. As mentioned before, strongly correlated dimensions are precluded from expansion for this purpose. An additional strategy is to ensure that new samples share the "same" cube measure value (e.g., mean income) as the existing samples in the query cell. The two-sample ***t*-test** is a relatively simple statistical method that can be used to determine whether two samples have the same mean (or any other point estimate), where "same" means that they do not differ significantly. (It is described in greater detail in Section 8.5.5 on model selection using statistical tests of significance.)

The test determines whether two samples have the same mean (the null hypothesis) with the only assumption being that they are both normally distributed. The test fails if there is evidence that the two samples do not share the same mean. Furthermore, the test can be performed with a confidence level as an input. This allows the user to control how strict or loose the query expansion will be.

Example 5.14 shows how the intracuboid expansion strategies just described can be used to answer a query on sample data.

Table 5.10 Sample Customer Survey Data

gender	*age*	*education*	*occupation*	*income*
female	23	college	teacher	$85,000
female	40	college	programmer	$50,000
female	31	college	programmer	$52,000
female	50	graduate	teacher	$90,000
female	62	graduate	CEO	$500,000
male	25	high school	programmer	$50,000
male	28	high school	CEO	$250,000
male	40	college	teacher	$80,000
male	50	college	programmer	$45,000
male	57	graduate	programmer	$80,000

Example 5.14 **Intracuboid query expansion to answer a query on sample data.** Consider a book retailer trying to learn more about its customers' annual income levels. In Table 5.10, a sample of the survey data collected is shown.[6] In the survey, customers are segmented by four attributes, namely *gender*, *age*, *education*, and *occupation*.

Let a query on customer income be "$age = 25$," where the user specifies a 95% confidence level. Suppose this returns an *income* value of $50,000 with a rather large confidence interval.[7] Suppose also, that this confidence interval is larger than a preset threshold and that the *age* dimension was found to have little correlation with *income* in this data set. Therefore, intracuboid expansion starts within the *age* dimension. The nearest cell is "$age = 23$," which returns an *income* of $85,000. The two-sample t-test at the 95% confidence level passes so the query expands; it is now "$age = \{23, 25\}$" with a smaller confidence interval than initially. However, it is still larger than the threshold, so expansion continues to the next nearest cell: "$age = 28$," which returns an *income* of $250,000. The two sample t-test between this cell and the original query cell fails; as a result, it is ignored. Next, "$age = 31$" is checked and it passes the test.

The confidence interval of the three cells combined is now below the threshold and the expansion finishes at "$age = \{23, 25, 31\}$." The mean of the *income* values at these three cells is $\frac{85{,}000+50{,}000+52{,}000}{3} = \$62{,}333$, which is returned as the query answer. It has a smaller confidence interval, and thus is more reliable than the response of $50,000, which would have been returned if intracuboid expansion had not been considered. ■

Method 2. Intercuboid query expansion. In this case, the expansion occurs by looking to a *more general cell*, as shown in Figure 5.15(b). For example, the cell in the 2-D cuboid

[6] For the sake of illustration, ignore the fact that the sample size is too small to be statistically significant.

[7] For the sake of the example, suppose this is true even though there is only one sample. In practice, more points are needed to calculate a legitimate value.

age-occupation can use its parent in either of the 1-D cuboids, *age* or *occupation*. Think of intercuboid expansion as just an extreme case of intracuboid expansion, where *all* the cells within a dimension are used in the expansion. This essentially sets the dimension to $*$ and thus generalizes to a higher-level cuboid.

A k-dimensional cell has k direct parents in the cuboid lattice, where each parent is $(k-1)$-dimensional. There are *many* more ancestor cells in the data cube (e.g., if multiple dimensions are rolled up simultaneously). However, we choose only one parent here to make the search space tractable and to limit the change in the query's semantics. As with intracuboid query expansion, correlated dimensions are not allowed in intercuboid expansions. Within the uncorrelated dimensions, the two-sample t-test can be performed to confirm that the parent and the query cell share the same sample mean. If multiple parent cells pass the test, the test's confidence level can be adjusted progressively higher until only one passes. Alternatively, multiple parent cells can be used to boost the confidence simultaneously. The choice is application dependent.

Example 5.15 **Intercuboid expansion to answer a query on sample data.** Given the input relation in Table 5.10, let the query on *income* be "*occupation* = teacher $\wedge$ *gender* = male." There is only one sample in Table 5.10 that matches the query, and it has an *income* of \$80,000. Suppose the corresponding confidence interval is larger than a preset threshold. We use intercuboid expansion to find a more reliable answer. There are two parent cells in the data cube: "*gender* = male" and "*occupation* = teacher." By moving up to "*gender* = male" (and thus setting *occupation* to $*$), the mean *income* is \$101,000. A two sample t-test reveals that this parent's sample mean differs significantly from that of the original query cell, so it is ignored. Next, "*occupation* = teacher" is considered. It has a mean *income* of \$85,000 and passes the two-sample t-test. As a result, the query is expanded to "*occupation* = teacher" and an *income* value of \$85,000 is returned with acceptable reliability. ■

"*How can we determine which method to choose—intracuboid expansion or intercuboid expansion?*" This is difficult to answer without knowing the data and the application. A strategy for choosing between the two is to consider what the tolerance is for change in the query's semantics. This depends on the specific dimensions chosen in the query. For instance, the user might tolerate a bigger change in semantics for the *age* dimension than *education*. The difference in tolerance could be so large that the user is willing to set *age* to $*$ (i.e., intercuboid expansion) rather than letting *education* change at all. Domain knowledge is helpful here.

So far, our discussion has only focused on full materialization of the sampling cube. In many real-world problems, this is often impossible, especially for high-dimensional cases. Real-world survey data, for example, can easily contain over 50 variables (i.e., dimensions). The sampling cube size would grow exponentially with the number of dimensions. To handle high-dimensional data, a sampling cube method called *Sampling Cube Shell* was developed. It integrates the Frag-Shell method of Section 5.2.4 with the query expansion approach. The shell computes only a subset of the full sampling cube.

The subset should consist of relatively low-dimensional cuboids (that are commonly queried) and cuboids that offer the most benefit to the user. The details are left to interested readers as an exercise. The method was tested on both real and synthetic data and found to be efficient and effective in answering queries.

5.3.2 Ranking Cubes: Efficient Computation of Top-*k* Queries

The data cube helps not only online analytical processing of multidimensional queries but also search and data mining. In this section, we introduce a new cube structure called *Ranking Cube* and examine how it contributes to the efficient processing of *top-k queries*. Instead of returning a large set of indiscriminative answers to a query, a **top-*k* query** (or **ranking query**) returns only the best k results according to a user-specified preference.

The results are returned in ranked order so that the best is at the top. The user-specified preference generally consists of two components: a *selection condition* and a *ranking function*. Top-k queries are common in many applications like searching web databases, k-nearest-neighbor searches with approximate matches, and similarity queries in multimedia databases.

Example 5.16 A top-*k* query. Consider an online used-car database, R, that maintains the following information for each car: producer (e.g., Ford, Honda), model (e.g., Taurus, Accord), type (e.g., sedan, convertible), color (e.g., red, silver), transmission (e.g., auto, manual), price, mileage, and so on. A typical top-k query over this database is

Q_1: select *top* 5 * from R
where $producer =$ "Ford" and $type =$ "sedan"
order by $(price - 10K)^2 + (mileage - 30K)^2$ asc

Within the dimensions (or attributes) for R, *producer* and *type* are used here as **selection dimensions**. The **ranking function** is given in the order-by clause. It specifies the **ranking dimensions**, *price* and *mileage*. Q_1 searches for the top-5 sedans made by Ford. The entries found are ranked or sorted in ascending (asc) order, according to the ranking function. The ranking function is formulated so that entries that have price and mileage closest to the user's specified values of \$10K and 30K, respectively, appear toward the top of the list. ■

The database may have many dimensions that could be used for selection, describing, for example, whether a car has power windows, air conditioning, or a sunroof. Users may pick any subset of dimensions and issue a top-k query using their preferred ranking function. There are many other similar application scenarios. For example, when searching for hotels, ranking functions are often constructed based on price and distance to an area of interest. Selection conditions can be imposed on, say, the hotel location

district, the star rating, and whether the hotel offers complimentary treats or Internet access. The ranking functions may be linear, quadratic, or any other form.

As shown in the preceding examples, individual users may not only propose ad hoc ranking functions, but also have different data subsets of interest. Users often want to thoroughly study the data via *multidimensional analysis* of the top-k query results. For example, if unsatisfied by the top-5 results returned by Q_1, the user may roll up on the producer dimension to check the top-5 results on all sedans. The dynamic nature of the problem imposes a great challenge to researchers. OLAP requires offline precomputation so that multidimensional analysis can be performed on-the-fly, yet the ad hoc ranking functions prohibit full materialization. A natural compromise is to adopt a *semi-offline materialization* and *semi-online computation* model.

Suppose a relation R has selection dimensions $(A_1, A_2, \ldots, A_S)$ and ranking dimensions $(N_1, N_2, \ldots, N_R)$. Values in each ranking dimension can be partitioned into multiple intervals according to the data and expected query distributions. Regarding the price of used cars, for example, we may have, say, these four partitions (or value ranges): $\leq 5K$, $[5-10K)$, $[10-15K)$, and $\geq 15K$. A ranking cube can be constructed by performing multidimensional aggregations on selection dimensions. We can store the count for each partition of each ranking dimension, thereby making the cube "rank-aware." The top-k queries can be answered by first accessing the cells in the more preferred value ranges before consulting the cells in the less preferred value ranges.

Example 5.17 **Using a ranking cube to answer a top-k query.** Suppose Table 5.11 shows C_{MT}, a materialized (i.e., precomputed) cuboid of a ranking cube for used-car sales. The cuboid, C_{MT}, is for the selection dimensions *producer* and *type*. It shows the count and corresponding tuple IDs (TIDs) for various partitions of the ranking dimensions, *price* and *mileage*.

Query Q_1 can be answered by using a selection condition to select the appropriate selection dimension values (i.e., *producer* = "Ford" and *type* = "sedan") in cuboid C_{MT}. In addition, the ranking function "$(price - 10K)^2 + (mileage - 30K)^2$" is used to find the tuples that most closely match the user's criteria. If there are not enough matching tuples found in the closest matching cells, the next closest matching cells will need to be accessed. We may even drill down to the corresponding lower-level cells to see the count distributions of cells that match the ranking function and additional criteria regarding, say, model, maintenance situation, or other loaded features. Only users who really want to see more detailed information, such as interior photos, will need to access the physical records stored in the database. ■

Table 5.11 Cuboid of a Ranking Cube for Used-Car Sales

producer	*type*	*price*	*mileage*	*count*	*TIDs*
Ford	sedan	<5K	30–40K	7	$t_6, \ldots, t_{68}$
Ford	sedan	5–10K	30–40K	50	$t_{15}, \ldots, t_{152}$
Honda	sedan	10–15K	30–40K	20	$t_8, \ldots, t_{32}$
…	…	…	…	…	…

Most real-life top-k queries are likely to involve only a small subset of selection attributes. To support high-dimensional ranking cubes, we can carefully select the cuboids that need to be materialized. For example, we could choose to materialize only the 1-D cuboids that contain single-selection dimensions. This will achieve low space overhead and still have high performance when the number of selection dimensions is large. In some cases, there may exist many ranking dimensions to support multiple users with rather different preferences. For example, buyers may search for houses by considering various factors like price, distance to school or shopping, number of years old, floor space, and tax. In this case, a possible solution is to create multiple data partitions, each of which consists of a subset of the ranking dimensions. The query processing may need to search over a joint space involving multiple data partitions.

In summary, the general philosophy of ranking cubes is to materialize such cubes on the set of selection dimensions. Use of the interval-based partitioning in ranking dimensions makes the ranking cube efficient and flexible at supporting ad hoc user queries. Various implementation techniques and query optimization methods have been developed for efficient computation and query processing based on this framework.

5.4 Multidimensional Data Analysis in Cube Space

Data cubes create a flexible and powerful means to group and aggregate data subsets. They allow data to be explored in multiple dimensional combinations and at varying aggregate granularities. This capability greatly increases the analysis bandwidth and helps effective discovery of interesting patterns and knowledge from data. The use of cube space makes the data space both meaningful and tractable.

This section presents methods of multidimensional data analysis that make use of data cubes to organize data into intuitive regions of interest at varying granularities. Section 5.4.1 presents *prediction cubes*, a technique for multidimensional data mining that facilitates predictive modeling in multidimensional space. Section 5.4.2 describes how to construct *multifeature cubes*. These support complex analytical queries involving multiple dependent aggregates at multiple granularities. Finally, Section 5.4.3 describes an interactive method for users to systematically explore cube space. In such *exception-based, discovery-driven exploration*, interesting exceptions or anomalies in the data are automatically detected and marked for users with visual cues.

5.4.1 Prediction Cubes: Prediction Mining in Cube Space

Recently, researchers have turned their attention toward **multidimensional data mining** to uncover knowledge at varying dimensional combinations and granularities. Such mining is also known as *exploratory multidimensional data mining* and *online analytical data mining (OLAM)*. Multidimensional data space is huge. In preparing the data, how can we identify the interesting subspaces for exploration? To what granularities should we aggregate the data? Multidimensional data mining in cube space organizes data of

interest into intuitive regions at various granularities. It analyzes and mines the data by applying various data mining techniques systematically over these regions.

There are at least four ways in which OLAP-style analysis can be fused with data mining techniques:

1. *Use cube space to define the data space for mining.* Each region in cube space represents a subset of data over which we wish to find interesting patterns. Cube space is defined by a set of expert-designed, informative dimension hierarchies, not just arbitrary subsets of data. Therefore, the use of cube space makes the data space both meaningful and tractable.
2. *Use OLAP queries to generate features and targets for mining.* The features and even the targets (that we wish to learn to predict) can sometimes be naturally defined as OLAP aggregate queries over regions in cube space.
3. *Use data mining models as building blocks in a multistep mining process.* Multidimensional data mining in cube space may consist of multiple steps, where data mining models can be viewed as building blocks that are used to describe the behavior of interesting data sets, rather than the end results.
4. *Use data cube computation techniques to speed up repeated model construction.* Multidimensional data mining in cube space may require building a model for each candidate data space, which is usually too expensive to be feasible. However, by carefully sharing computation across model construction for different candidates based on data cube computation techniques, efficient mining is achievable.

In this subsection we study *prediction cubes*, an example of multidimensional data mining where the cube space is explored for prediction tasks. A **prediction cube** is a cube structure that stores prediction models in multidimensional data space and supports prediction in an OLAP manner. Recall that in a data cube, each cell value is an aggregate number (e.g., count) computed over the data subset in that cell. However, each cell value in a prediction cube is computed by evaluating a predictive model built on the data subset in that cell, thereby representing that subset's predictive behavior.

Instead of seeing prediction models as the end result, prediction cubes use prediction models as building blocks to define the interestingness of data subsets, that is, they identify data subsets that indicate more accurate prediction. This is best explained with an example.

Example 5.18 **Prediction cube for identification of interesting cube subspaces.** Suppose a company has a customer table with the attributes *time* (with two granularity levels: *month* and *year*), *location* (with two granularity levels: *state* and *country*), *gender*, *salary*, and one class-label attribute: *valued_customer*. A manager wants to analyze the decision process of whether a customer is highly valued with respect to *time* and *location*. In particular, he is interested in the question "*Are there times at and locations in which the value of a*

customer depended greatly on the customer's gender?" Notice that he believes *time* and *location* play a role in predicting valued customers, but at what granularity levels do they depend on *gender* for this task? For example, is performing analysis using {*month, country*} better than {*year, state*}?

Consider a data table **D** (e.g., the customer table). Let **X** be the attributes set for which no concept hierarchy has been defined (e.g., *gender, salary*). Let Y be the class-label attribute (e.g., *valued_customer*), and **Z** be the set of multilevel attributes, that is, attributes for which concept hierarchies have been defined (e.g., *time, location*). Let **V** be the set of attributes for which we would like to define their predictiveness. In our example, this set is {*gender*}. The predictiveness of **V** on a data subset can be quantified by the difference in accuracy between the model built on that subset using **X** to predict Y and the model built on that subset using $\mathbf{X} - \mathbf{V}$ (e.g., {*salary*}) to predict Y. The intuition is that, if the difference is large, **V** must play an important role in the prediction of class label Y.

Given a set of attributes, **V**, and a learning algorithm, the prediction cube at granularity $\langle l_1, \ldots, l_d \rangle$ (e.g., $\langle year, state \rangle$) is a d-dimensional array, in which the value in each cell (e.g., [2010, Illinois]) is the predictiveness of **V** evaluated on the subset defined by the cell (e.g., the records in the customer table with *time* in 2010 and *location* in Illinois). ■

Supporting OLAP roll-up and drill-down operations on a prediction cube is a computational challenge requiring the materialization of cell values at many different granularities. For simplicity, we can consider only full materialization. A naïve way to fully materialize a prediction cube is to exhaustively build models and evaluate them for each cell and granularity. This method is very expensive if the base data set is large. An ensemble method called **Probability-Based Ensemble (PBE)** was developed as a more feasible alternative. It requires model construction for only the finest-grained cells. OLAP-style bottom-up aggregation is then used to generate the values of the coarser-grained cells.

The prediction of a predictive model can be seen as finding a class label that maximizes a scoring function. The PBE method was developed to approximately make the scoring function of any predictive model distributively decomposable. In our discussion of data cube measures in Section 4.2.4, we showed that distributive and algebraic measures can be computed efficiently. Therefore, if the scoring function used is distributively or algebraically decomposable, prediction cubes can also be computed with efficiency. In this way, the PBE method reduces prediction cube computation to data cube computation.

For example, previous studies have shown that the naïve Bayes classifier has an algebraically decomposable scoring function, and the kernel density–based classifier has a distributively decomposable scoring function.[8] Therefore, either of these could be used

[8] Naïve Bayes classifiers are detailed in Chapter 8. Kernel density–based classifiers, such as support vector machines, are described in Chapter 9.

to implement prediction cubes efficiently. The PBE method presents a novel approach to multidimensional data mining in cube space.

5.4.2 Multifeature Cubes: Complex Aggregation at Multiple Granularities

Data cubes facilitate the answering of analytical or mining-oriented queries as they allow the computation of aggregate data at multiple granularity levels. Traditional data cubes are typically constructed on commonly used dimensions (e.g., *time*, *location*, and *product*) using simple measures (e.g., count( ), average( ), and sum()). In this section, you will learn a newer way to define data cubes called **multifeature cubes**. Multifeature cubes enable more in-depth analysis. They can compute more complex queries of which the measures depend on groupings of multiple aggregates at varying granularity levels. The queries posed can be much more elaborate and task-specific than traditional queries, as we shall illustrate in the next examples. Many complex data mining queries can be answered by multifeature cubes without significant increase in computational cost, in comparison to cube computation for simple queries with traditional data cubes.

To illustrate the idea of multifeature cubes, let's first look at an example of a query on a simple data cube.

Example 5.19 **A simple data cube query.** Let the query be "*Find the total sales in 2010, broken down by item, region, and month, with subtotals for each dimension.*" To answer this query, a traditional data cube is constructed that aggregates the total sales at the following eight different granularity levels: {(*item, region, month*), (*item, region*), (*item, month*), (*month, region*), (*item*), (*month*), (*region*), ()}, where () represents all. This data cube is simple in that it does not involve any dependent aggregates. ■

To illustrate what is meant by "dependent aggregates," let's examine a more complex query, which can be computed with a multifeature cube.

Example 5.20 **A complex query involving dependent aggregates.** Suppose the query is "*Grouping by all subsets of* {*item, region, month*}, *find the maximum price in 2010 for each group and the total sales among all maximum price tuples.*"

The specification of such a query using standard SQL can be long, repetitive, and difficult to optimize and maintain. Alternatively, it can be specified concisely using an extended SQL syntax as follows:

```
select      item, region, month, max(price), sum(R.sales)
from        Purchases
where       year = 2010
cube by     item, region, month: R
such that   R.price = max(price)
```

The tuples representing purchases in 2010 are first selected. The cube by clause computes aggregates (or group-by's) for all possible combinations of the attributes *item,*

region, and *month*. It is an *n*-dimensional generalization of the group-by clause. The attributes specified in the cube by clause are the **grouping attributes**. Tuples with the same value on all grouping attributes form one group. Let the groups be $g_1, \ldots, g_r$. For each group of tuples g_i, the maximum price max_{g_i} among the tuples forming the group is computed. The variable R is a **grouping variable**, ranging over all tuples in group g_i that have a price equal to max_{g_i} (as specified in the such that clause). The sum of sales of the tuples in g_i that R ranges over is computed and returned with the values of the grouping attributes of g_i.

The resulting cube is a multifeature cube in that it supports complex data mining queries for which multiple dependent aggregates are computed at a variety of granularities. For example, the sum of sales returned in this query is dependent on the set of maximum price tuples for each group. In general, multifeature cubes give users the flexibility to define sophisticated, task-specific cubes on which multidimensional aggregation and OLAP-based mining can be performed. ■

"How can multifeature cubes be computed efficiently?" The computation of a multifeature cube depends on the types of aggregate functions used in the cube. In Chapter 4, we saw that aggregate functions can be categorized as either distributive, algebraic, or holistic. Multifeature cubes can be organized into the same categories and computed efficiently by minor extension of the cube computation methods in Section 5.2.

5.4.3 Exception-Based, Discovery-Driven Cube Space Exploration

As studied in previous sections, a data cube may have a large number of cuboids, and each cuboid may contain a large number of (aggregate) cells. With such an overwhelmingly large space, it becomes a burden for users to even just browse a cube, let alone think of exploring it thoroughly. Tools need to be developed to assist users in intelligently exploring the huge aggregated space of a data cube.

In this section, we describe a **discovery-driven approach** to exploring cube space. Precomputed measures indicating data exceptions are used to guide the user in the data analysis process, at all aggregation levels. We hereafter refer to these measures as *exception indicators*. Intuitively, an **exception** is a data cube cell value that is significantly different from the value anticipated, based on a statistical model. The model considers variations and patterns in the measure value across *all the dimensions* to which a cell belongs. For example, if the analysis of *item-sales* data reveals an increase in sales in December in comparison to all other months, this may seem like an exception in the time dimension. However, it is not an exception if the *item* dimension is considered, since there is a similar increase in sales for other items during December.

The model considers exceptions hidden at all aggregated group-by's of a data cube. Visual cues, such as background color, are used to reflect each cell's degree of exception, based on the precomputed exception indicators. Efficient algorithms have been proposed for cube construction, as discussed in Section 5.2. The computation of exception indicators can be overlapped with cube construction, so that the overall construction of data cubes for discovery-driven exploration is efficient.

Three measures are used as exception indicators to help identify data anomalies. These measures indicate the degree of surprise that the quantity in a cell holds, with respect to its expected value. The measures are computed and associated with every cell, for all aggregation levels. They are as follows:

- **SelfExp**: This indicates the degree of surprise of the cell value, relative to other cells at the same aggregation level.
- **InExp**: This indicates the degree of surprise somewhere beneath the cell, if we were to drill down from it.
- **PathExp**: This indicates the degree of surprise for each drill-down path from the cell.

The use of these measures for discovery-driven exploration of data cubes is illustrated in Example 5.21.

Example 5.21 **Discovery-driven exploration of a data cube.** Suppose that you want to analyze the monthly sales at *AllElectronics* as a percentage difference from the previous month. The dimensions involved are *item*, *time*, and *region*. You begin by studying the data aggregated over all items and sales regions for each month, as shown in Figure 5.16.

To view the exception indicators, you click on a button marked highlight exceptions on the screen. This translates the SelfExp and InExp values into visual cues, displayed with each cell. Each cell's background color is based on its SelfExp value. In addition, a box is drawn around each cell, where the thickness and color of the box are functions of its InExp value. Thick boxes indicate high InExp values. In both cases, the darker the color, the greater the degree of exception. For example, the dark, thick boxes for sales during July, August, and September signal the user to explore the lower-level aggregations of these cells by drilling down.

Drill-downs can be executed along the aggregated *item* or *region* dimensions. "*Which path has more exceptions?*" you wonder. To find this out, you select a cell of interest and trigger a path exception module that colors each dimension based on the PathExp value of the cell. This value reflects that path's degree of surprise. Suppose that the path along *item* contains more exceptions.

A drill-down along *item* results in the cube slice of Figure 5.17, showing the sales over time for each item. At this point, you are presented with many different sales values to analyze. By clicking on the highlight exceptions button, the visual cues are displayed, bringing focus to the exceptions. Consider the sales difference of 41% for "*Sony*

Sum of sales	*Month*											
	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Total		1%	–1%	0%	1%	3%	–1%	–9%	–1%	2%	–4%	3%

Figure 5.16 Change in sales over time.

Avg. sales	*Month*											
Item	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Sony b/w printer		9%	–8%	2%	–5%	14%	–4%	0%	41%	–13%	–15%	–11%
Sony color printer		0%	0%	3%	2%	4%	–10%	–13%	0%	4%	–6%	4%
HP b/w printer		–2%	1%	2%	3%	8%	0%	–12%	–9%	3%	–3%	6%
HP color printer		0%	0%	–2%	1%	0%	–1%	–7%	–2%	1%	–4%	1%
IBM desktop computer		1%	–2%	–1%	–1%	3%	3%	–10%	4%	1%	–4%	–1%
IBM laptop computer		0%	0%	–1%	3%	4%	2%	–10%	–2%	0%	–9%	3%
Toshiba desktop computer		–2%	–5%	1%	1%	–1%	1%	5%	–3%	–5%	–1%	–1%
Toshiba laptop computer		1%	0%	3%	0%	–2%	–2%	–5%	3%	2%	–1%	0%
Logitech mouse		3%	–2%	–1%	0%	4%	6%	–11%	2%	1%	–4%	0%
Ergo-way mouse		0%	0%	2%	3%	1%	–2%	–2%	–5%	0%	–5%	8%

Figure 5.17 Change in sales for each *item-time* combination.

Avg. sales	*Month*											
Region	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
North		–1%	–3%	–1%	0%	3%	4%	–7%	1%	0%	–3%	–3%
South		–1%	1%	–9%	6%	–1%	–39%	9%	–34%	4%	1%	7%
East		–1%	–2%	2%	–3%	1%	18%	–2%	11%	–3%	–2%	–1%
West		4%	0%	–1%	–3%	5%	1%	–18%	8%	5%	–8%	1%

Figure 5.18 Change in sales for the item *IBM desktop computer* per region.

b/w printers" in September. This cell has a dark background, indicating a high SelfExp value, meaning that the cell is an exception. Consider now the sales difference of −15% for "*Sony b/w printers*" in November and of −11% in December. The −11% value for December is marked as an exception, while the −15% value is not, even though −15% is a bigger deviation than −11%. This is because the exception indicators consider all the dimensions that a cell is in. Notice that the December sales of most of the other items have a large positive value, while the November sales do not. Therefore, by considering the cell's position in the cube, the sales difference for "*Sony b/w printers*" in December is exceptional, while the November sales difference of this item is not.

The InExp values can be used to indicate exceptions at lower levels that are not visible at the current level. Consider the cells for "*IBM desktop computers*" in July and September. These both have a dark, thick box around them, indicating high InExp values. You may decide to further explore the sales of "*IBM desktop computers*" by drilling down along *region*. The resulting sales difference by *region* is shown in Figure 5.18, where the highlight exceptions option has been invoked. The visual cues displayed make it easy to instantly notice an exception for the sales of "*IBM desktop computers*" in the southern region, where such sales have decreased by −39% and −34% in July and September,

respectively. These detailed exceptions were far from obvious when we were viewing the data as an *item-time* group-by, aggregated over *region* in Figure 5.17. Thus, the InExp value is useful for searching for exceptions at lower-level cells of the cube. ■

"*How are the exception values computed?*" The SelfExp, InExp, and PathExp measures are based on a statistical method for table analysis. They take into account all of the group-by's (aggregations) in which a given cell value participates. A cell value is considered an exception based on how much it differs from its expected value, where its expected value is determined with a statistical model. The difference between a given cell value and its expected value is called a **residual**. Intuitively, the larger the residual, the more the given cell value is an exception. The comparison of residual values requires us to scale the values based on the expected standard deviation associated with the residuals. A cell value is therefore considered an exception if its scaled residual value exceeds a prespecified threshold. The SelfExp, InExp, and PathExp measures are based on this scaled residual.

The expected value of a given cell is a function of the higher-level group-by's of the given cell. For example, given a cube with the three dimensions *A*, *B*, and *C*, the expected value for a cell at the *i*th position in *A*, the *j*th position in *B*, and the *k*th position in *C* is a function of γ, γ_i^A, γ_j^B, γ_k^C, γ_{ij}^{AB}, γ_{ik}^{AC}, and γ_{jk}^{BC}, which are coefficients of the statistical model used. The coefficients reflect how different the values at more detailed levels are, based on generalized impressions formed by looking at higher-level aggregations. In this way, the exception quality of a cell value is based on the exceptions of the values below it. Thus, when seeing an exception, it is natural for the user to further explore the exception by drilling down.

"*How can the data cube be efficiently constructed for discovery-driven exploration?*" This computation consists of three phases. The first step involves the computation of the aggregate values defining the cube, such as `sum` or `count`, over which exceptions will be found. The second phase consists of model fitting, in which the coefficients mentioned before are determined and used to compute the standardized residuals. This phase can be overlapped with the first phase because the computations involved are similar. The third phase computes the SelfExp, InExp, and PathExp values, based on the standardized residuals. This phase is computationally similar to phase 1. Therefore, the computation of data cubes for discovery-driven exploration can be done efficiently.

5.5 Summary

- **Data cube computation and exploration** play an essential role in data warehousing and are important for flexible data mining in multidimensional space.
- A data cube consists of a **lattice of cuboids**. Each cuboid corresponds to a different degree of summarization of the given multidimensional data. **Full materialization** refers to the computation of all the cuboids in a data cube lattice. **Partial materialization** refers to the selective computation of a subset of the cuboid cells in the

lattice. Iceberg cubes and shell fragments are examples of partial materialization. An **iceberg cube** is a data cube that stores only those cube cells that have an aggregate value (e.g., count) above some minimum support threshold. For **shell fragments** of a data cube, only some cuboids involving a small number of dimensions are computed, and queries on additional combinations of the dimensions can be computed on-the-fly.

- There are several efficient **data cube computation methods**. In this chapter, we discussed four cube computation methods in detail: (1) **MultiWay** array aggregation for materializing full data cubes in sparse-array-based, bottom-up, shared computation; (2) **BUC** for computing iceberg cubes by exploring ordering and sorting for efficient top-down computation; (3) **Star-Cubing** for computing iceberg cubes by integrating top-down and bottom-up computation using a star-tree structure; and (4) **shell-fragment cubing**, which supports high-dimensional OLAP by precomputing only the partitioned cube shell fragments.
- **Multidimensional data mining in cube space** is the integration of knowledge discovery with multidimensional data cubes. It facilitates systematic and focused knowledge discovery in large structured and semi-structured data sets. It will continue to endow analysts with tremendous flexibility and power at multidimensional and multigranularity exploratory analysis. This is a vast open area for researchers to build powerful and sophisticated data mining mechanisms.
- Techniques for processing advanced queries have been proposed that take advantage of cube technology. These include **sampling cubes** for multidimensional analysis on sampling data, and **ranking cubes** for efficient top-k (ranking) query processing in large relational data sets.
- This chapter highlighted three approaches to multidimensional data analysis with data cubes. **Prediction cubes** compute prediction models in multidimensional cube space. They help users identify interesting data subsets at varying degrees of granularity for effective prediction. **Multifeature cubes** compute complex queries involving multiple dependent aggregates at multiple granularities. **Exception-based, discovery-driven exploration** of cube space displays visual cues to indicate discovered data exceptions at all aggregation levels, thereby guiding the user in the data analysis process.

5.6 Exercises

5.1 Assume that a 10-D base cuboid contains only three base cells: (1) (a_1, d_2, d_3, d_4, …, d_9, d_{10}), (2) (d_1, b_2, d_3, d_4, …, d_9, d_{10}), and (3) (d_1, d_2, c_3, d_4, …, d_9, d_{10}), where $a_1 \neq d_1$, $b_2 \neq d_2$, and $c_3 \neq d_3$. The measure of the cube is count( ).

(a) How many *nonempty* cuboids will a full data cube contain?

(b) How many *nonempty* aggregate (i.e., nonbase) cells will a full cube contain?

(c) How many *nonempty* aggregate cells will an iceberg cube contain if the condition of the iceberg cube is "count ≥ 2"?

(d) A cell, c, is a *closed cell* if there exists no cell, d, such that d is a specialization of cell c (i.e., d is obtained by replacing a $*$ in c by a non-$*$ value) and d has the same measure value as c. A *closed cube* is a data cube consisting of only closed cells. How many closed cells are in the full cube?

5.2 There are several typical cube computation methods, such as *MultiWay* [ZDN97], *BUC* [BR99], and *Star-Cubing* [XHLW03]. Briefly describe these three methods (i.e., use one or two lines to outline the key points), and compare their feasibility and performance under the following conditions:

(a) Computing a dense full cube of low dimensionality (e.g., less than eight dimensions).

(b) Computing an iceberg cube of around 10 dimensions with a highly skewed data distribution.

(c) Computing a sparse iceberg cube of high dimensionality (e.g., over 100 dimensions).

5.3 Suppose a data cube, C, has D dimensions, and the base cuboid contains k distinct tuples.

(a) Present a formula to calculate the minimum number of cells that the cube, C, may contain.

(b) Present a formula to calculate the maximum number of cells that C may contain.

(c) Answer parts (a) and (b) as if the count in each cube cell must be no less than a threshold, v.

(d) Answer parts (a) and (b) as if only closed cells are considered (with the minimum count threshold, v).

5.4 Suppose that a base cuboid has three dimensions, A, B, C, with the following number of cells: $|A| = 1,000,000$, $|B| = 100$, and $|C| = 1000$. Suppose that each dimension is evenly partitioned into 10 portions for *chunking*.

(a) Assuming each dimension has only one level, draw the complete lattice of the cube.

(b) If each cube cell stores one measure with four bytes, what is the total size of the computed cube if the cube is *dense*?

(c) State the order for computing the chunks in the cube that requires the least amount of space, and compute the total amount of main memory space required for computing the 2-D planes.

5.5 Often, the aggregate *count* value of many cells in a large data cuboid is zero, resulting in a huge, yet sparse, multidimensional matrix.

(a) Design an implementation method that can elegantly overcome this sparse matrix problem. Note that you need to explain your data structures in detail and discuss the space needed, as well as how to retrieve data from your structures.

(b) Modify your design in (a) to handle *incremental data updates*. Give the reasoning behind your new design.

5.6 When computing a cube of high dimensionality, we encounter the inherent *curse of dimensionality* problem: There exists a huge number of subsets of combinations of dimensions.

(a) Suppose that there are only two base cells, $\{(a_1, a_2, a_3, \ldots, a_{100})$ and $(a_1, a_2, b_3, \ldots, b_{100})\}$, in a 100-D base cuboid. Compute the number of nonempty aggregate cells. Comment on the storage space and time required to compute these cells.

(b) Suppose we are to compute an iceberg cube from (a). If the minimum support count in the iceberg condition is 2, how many aggregate cells will there be in the iceberg cube? Show the cells.

(c) Introducing iceberg cubes will lessen the burden of computing trivial aggregate cells in a data cube. However, even with iceberg cubes, we could still end up having to compute a large number of trivial uninteresting cells (i.e., with small counts). Suppose that a database has 20 tuples that map to (or cover) the two following base cells in a 100-D base cuboid, each with a cell count of 10: $\{(a_1, a_2, a_3, \ldots, a_{100}) : 10, (a_1, a_2, b_3, \ldots, b_{100}) : 10\}$.

i. Let the minimum support be 10. How many distinct aggregate cells will there be like the following: $\{(a_1, a_2, a_3, a_4, \ldots, a_{99}, *) : 10, \ldots, (a_1, a_2, *, a_4, \ldots, a_{99}, a_{100}) : 10, \ldots, (a_1, a_2, a_3, *, \ldots, *, *) : 10\}$?

ii. If we ignore all the aggregate cells that can be obtained by replacing some constants with $*$'s while keeping the same measure value, how many distinct cells remain? What are the cells?

5.7 Propose an algorithm that computes *closed iceberg cubes* efficiently.

5.8 Suppose that we want to compute an iceberg cube for the dimensions, *A*, *B*, *C*, *D*, where we wish to materialize all cells that satisfy a minimum support count of at least *v*, and where *cardinality(A) < cardinality(B) < cardinality(C) < cardinality(D)*. Show the *BUC* processing tree (which shows the order in which the BUC algorithm explores a data cube's lattice, starting from **all**) for the construction of this iceberg cube.

5.9 Discuss how you might extend the *Star-Cubing* algorithm to compute iceberg cubes where the iceberg condition tests for an **avg** that is no bigger than some value, *v*.

5.10 A flight data warehouse for a travel agent consists of six dimensions: *traveler, departure (city), departure_time, arrival, arrival_time*, and *flight*; and two measures: **count()** and **avg_fare()**, where **avg_fare()** stores the concrete fare at the lowest level but the average fare at other levels.

(a) Suppose the cube is fully materialized. Starting with the *base cuboid* [*traveler*, *departure*, *departure_time*, *arrival*, *arrival_time*, *flight*], what specific OLAP operations (e.g., roll-up *flight* to *airline*) should one perform to list the average fare per month for *each business traveler* who flies American Airlines (*AA*) from Los Angeles in 2009?

(b) Suppose we want to compute a data cube where the condition is that the minimum number of records is 10 and the average fare is over $500. Outline an efficient cube computation method (based on common sense about flight data distribution).

5.11 **(Implementation project)** There are four typical data cube computation methods: MultiWay [ZDN97], BUC [BR99], H-Cubing [HPDW01], and Star-Cubing [XHLW03].

(a) Implement any one of these cube computation algorithms and describe your implementation, experimentation, and performance. Find another student who has implemented a different algorithm on the same platform (e.g., C++ on Linux) and compare your algorithm performance with his or hers.

Input:

i. An n-dimensional base cuboid table (for $n < 20$), which is essentially a relational table with n attributes.

ii. An iceberg condition: count $(C) \geq k$, where k is a positive integer as a parameter.

Output:

i. The set of computed cuboids that satisfy the iceberg condition, in the order of your output generation.

ii. Summary of the set of cuboids in the form of "*cuboid ID*: the number of nonempty cells," sorted in alphabetical order of cuboids (e.g., *A*: 155, *AB*: 120, *ABC*: 22, *ABCD*: 4, *ABCE*: 6, *ABD*: 36), where the number after : represents the number of nonempty cells. (This is used to quickly check the correctness of your results.)

(b) Based on your implementation, discuss the following:

i. What challenging computation problems are encountered as the number of dimensions grows large?

ii. How can iceberg cubing solve the problems of part (a) for some data sets (and characterize such data sets)?

iii. Give one simple example to show that sometimes iceberg cubes cannot provide a good solution.

(c) Instead of computing a high-dimensionality data cube, we may choose to materialize the cuboids that have only a small number of dimension combinations. For example, for a 30-D data cube, we may only compute the 5-D cuboids for every possible 5-D combination. The resulting cuboids form a *shell cube*. Discuss how easy or hard it is to modify your cube computation algorithm to facilitate such computation.

5.12 The *sampling cube* was proposed for multidimensional analysis of sampling data (e.g., survey data). In many real applications, sampling data can be of high dimensionality (e.g., it is not unusual to have more than 50 dimensions in a survey data set).

(a) How can we construct an efficient and scalable high-dimensional sampling cube in large sampling data sets?

(b) Design an efficient incremental update algorithm for such a high-dimensional sampling cube.

(c) Discuss how to support quality drill-down given that some low-level cells may be empty or contain too few data for reliable analysis.

5.13 The *ranking cube* was proposed for efficient computation of top-k (ranking) queries in relational databases. Recently, researchers have proposed another kind of query, called a *skyline query*. A *skyline query* returns all the objects p_i such that p_i is not dominated by any other object p_j, where dominance is defined as follows. Let the value of p_i on dimension d be $v(p_i, d)$. We say p_i is dominated by p_j if and only if for each preference dimension d, $v(p_j, d) \leq v(p_i, d)$, and there is at least one d where the equality does not hold.

(a) Design a ranking cube so that skyline queries can be processed efficiently.

(b) Skyline queries are sometimes too strict to be desirable to some users. One may generalize the concept of skyline into *generalized skyline* as follows: *Given a d-dimensional database and a query q, the* ***generalized skyline*** *is the set of the following objects: (1) the skyline objects and (2) the nonskyline objects that are ϵ-neighbors of a skyline object, where r is an ϵ-neighbor of an object p if the distance between p and r is no more than ϵ*. Design a ranking cube to process generalized skyline queries efficiently.

5.14 The ranking cube was designed to support top-k (ranking) queries in relational database systems. However, ranking queries are also posed to data warehouses, where ranking is on multidimensional aggregates instead of on measures of base facts. For example, consider a product manager who is analyzing a sales database that stores the nationwide sales history, organized by location and time. To make investment decisions, the manager may pose the following query: "*What are the top-10 (state, year) cells having the largest total product sales?*" He may further drill down and ask, "*What are the top-10 (city, month) cells?*" Suppose the system can perform such partial materialization to derive two types of materialized cuboids: a *guiding cuboid* and a *supporting cuboid*, where the former contains a number of guiding cells that provide concise, high-level data statistics to guide the ranking query processing, whereas the latter provides inverted indices for efficient online aggregation.

(a) Derive an efficient method for computing such aggregate ranking cubes.

(b) Extend your framework to handle more advanced measures. One such example could be as follows. Consider an organization donation database, where donors are grouped by "*age*," "*income*," and other attributes. Interesting questions include: "*Which age and income groups have made the top*-k *average amount of donation (per donor)?*" and "*Which income group of donors has the largest standard deviation in the donation amount?*"

5.15 The *prediction cube* is a good example of multidimensional data mining in cube space.

(a) Propose an efficient algorithm that computes prediction cubes in a given multidimensional database.

(b) For what kind of classification models can your algorithm be applied? Explain.

5.16 *Multifeature cubes* allow us to construct interesting data cubes based on rather sophisticated query conditions. Can you construct the following multifeature cube by translating the following user requests into queries using the form introduced in this textbook?

(a) Construct a smart shopper cube where a shopper is smart if at least 10% of the goods she buys in each shopping trip are on sale.

(b) Construct a data cube for best-deal products where best-deal products are those products for which the price is the lowest for this product in the given month.

5.17 *Discovery-driven cube exploration* is a desirable way to mark interesting points among a large number of cells in a data cube. Individual users may have different views on whether a point should be considered interesting enough to be marked. Suppose one would like to mark those objects of which the absolute value of z score is over 2 in every row and column in a d-dimensional plane.

(a) Derive an efficient computation method to identify such points during the data cube computation.

(b) Suppose a partially materialized cube has $(d-1)$-dimensional and $(d+1)$-dimensional cuboids materialized but not the d-dimensional one. Derive an efficient method to mark those $(d-1)$-dimensional cells with d-dimensional children that contain such marked points.

5.7 Bibliographic Notes

Efficient computation of multidimensional aggregates in data cubes has been studied by many researchers. Gray, Chaudhuri, Bosworth, et al. [GCB+97] proposed *cube-by* as a relational aggregation operator generalizing group-by, crosstabs, and subtotals, and categorized data cube measures into three categories: *distributive*, *algebraic*, and *holistic*. Harinarayan, Rajaraman, and Ullman [HRU96] proposed a greedy algorithm for the partial materialization of cuboids in the computation of a data cube. Sarawagi and Stonebraker [SS94] developed a chunk-based computation technique for the efficient organization of large multidimensional arrays. Agarwal, Agrawal, Deshpande, et al. [AAD+96] proposed several guidelines for efficient computation of multidimensional aggregates for ROLAP servers.

The chunk-based MultiWay array aggregation method for data cube computation in MOLAP was proposed in Zhao, Deshpande, and Naughton [ZDN97]. Ross and Srivastava [RS97] developed a method for computing sparse data cubes. Iceberg queries are first described in Fang, Shivakumar, Garcia-Molina, et al. [FSGM+98]. BUC, a scalable method that computes iceberg cubes from the apex cuboid downwards, was introduced by Beyer and Ramakrishnan [BR99]. Han, Pei, Dong, and Wang [HPDW01] introduced an H-Cubing method for computing iceberg cubes with complex measures using an H-tree structure.

The Star-Cubing method for computing iceberg cubes with a dynamic star-tree structure was introduced by Xin, Han, Li, and Wah [XHLW03]. MM-Cubing, an efficient

iceberg cube computation method that factorizes the lattice space was developed by Shao, Han, and Xin [SHX04]. The shell-fragment-based cubing approach for efficient high-dimensional OLAP was proposed by Li, Han, and Gonzalez [LHG04].

Aside from computing iceberg cubes, another way to reduce data cube computation is to materialize condensed, dwarf, or quotient cubes, which are variants of closed cubes. Wang, Feng, Lu, and Yu proposed computing a reduced data cube, called a *condensed cube* [WLFY02]. Sismanis, Deligiannakis, Roussopoulos, and Kotids proposed computing a compressed data cube, called a *dwarf cube* [SDRK02]. Lakeshmanan, Pei, and Han proposed a *quotient cube* structure to summarize a data cube's semantics [LPH02], which has been further extended to a *qc-tree structure* by Lakshmanan, Pei, and Zhao [LPZ03]. An *aggregation-based* approach, called C-Cubing (i.e., *Closed-Cubing*), has been developed by Xin, Han, Shao, and Liu [XHSL06], which performs efficient closed-cube computation by taking advantage of a new algebraic measure *closedness*.

There are also various studies on the computation of compressed data cubes by approximation, such as *quasi-cubes* by Barbara and Sullivan [BS97]; *wavelet cubes* by Vitter, Wang, and Iyer [VWI98]; *compressed cubes* for query approximation on continuous dimensions by Shanmugasundaram, Fayyad, and Bradley [SFB99]; using log-linear models to compress data cubes by Barbara and Wu [BW00]; and OLAP over uncertain and imprecise data by Burdick, Deshpande, Jayram, et al. [BDJ+05].

For works regarding the selection of materialized cuboids for efficient OLAP query processing, see Chaudhuri and Dayal [CD97]; Harinarayan, Rajaraman, and Ullman [HRU96]; Srivastava, Dar, Jagadish, and Levy [SDJL96]; Gupta [Gup97], Baralis, Paraboschi, and Teniente [BPT97]; and Shukla, Deshpande, and Naughton [SDN98]. Methods for cube size estimation can be found in Deshpande, Naughton, Ramasamy, et al. [DNR+97], Ross and Srivastava [RS97], and Beyer and Ramakrishnan [BR99]. Agrawal, Gupta, and Sarawagi [AGS97] proposed operations for modeling multidimensional databases.

Data cube modeling and computation have been extended well beyond relational data. Computation of *stream cubes* for multidimensional stream data analysis has been studied by Chen, Dong, Han, et al. [CDH+02]. Efficient computation of *spatial data cubes* was examined by Stefanovic, Han, and Koperski [SHK00], efficient OLAP in spatial data warehouses was studied by Papadias, Kalnis, Zhang, and Tao [PKZT01], and a map cube for visualizing spatial data warehouses was proposed by Shekhar, Lu, Tan, et al. [SLT+01]. A multimedia data cube was constructed in MultiMediaMiner by Zaiane, Han, Li, et al. [ZHL+98]. For analysis of multidimensional text databases, *TextCube*, based on the vector space model, was proposed by Lin, Ding, Han, et al. [LDH+08], and *TopicCube*, based on a topic modeling approach, was proposed by Zhang, Zhai, and Han [ZZH09]. *RFID Cube* and *FlowCube* for analyzing RFID data were proposed by Gonzalez, Han, Li, et al. [GHLK06, GHL06].

The *sampling cube* was introduced for analyzing sampling data by Li, Han, Yin, et al. [LHY+08]. The *ranking cube* was proposed by Xin, Han, Cheng, and Li [XHCL06] for efficient processing of ranking (top-k) queries in databases. This methodology has been extended by Wu, Xin, and Han [WXH08] to *ARCube*, which supports the ranking of aggregate queries in partially materialized data cubes. It has also been extended by

Wu, Xin, Mei, and Han [WXMH09] to *PromoCube*, which supports promotion query analysis in multidimensional space.

The discovery-driven exploration of OLAP data cubes was proposed by Sarawagi, Agrawal, and Megiddo [SAM98]. Further studies on integration of OLAP with data mining capabilities for intelligent exploration of multidimensional OLAP data were done by Sarawagi and Sathe [SS01]. The construction of multifeature data cubes is described by Ross, Srivastava, and Chatziantoniou [RSC98]. Methods for answering queries quickly by online aggregation are described by Hellerstein, Haas, and Wang [HHW97] and Hellerstein, Avnur, Chou, et al. [HAC+99]. A cube-gradient analysis problem, called *cubegrade*, was first proposed by Imielinski, Khachiyan, and Abdulghani [IKA02]. An efficient method for multidimensional constrained gradient analysis in data cubes was studied by Dong, Han, Lam, et al. [DHL+01].

Mining cube space, or integration of knowledge discovery and OLAP cubes, has been studied by many researchers. The concept of online analytical mining (OLAM), or OLAP mining, was introduced by Han [Han98]. Chen, Dong, Han, et al. developed a *regression cube* for regression-based multidimensional analysis of time-series data [CDH+02, CDH+06]. Fagin, Guha, Kumar, et al. [FGK+05] studied data mining in *multistructured databases*. B.-C. Chen, L. Chen, Lin, and Ramakrishnan [CCLR05] proposed *prediction cubes*, which integrate prediction models with data cubes to discover interesting data subspaces for facilitated prediction. Chen, Ramakrishnan, Shavlik, and Tamma [CRST06] studied the use of data mining models as building blocks in a multistep mining process, and the use of cube space to intuitively define the space of interest for predicting global aggregates from local regions. Ramakrishnan and Chen [RC07] presented an organized picture of exploratory mining in cube space.

6 Mining Frequent Patterns, Associations, and Correlations: Basic Concepts and Methods

Imagine that you are a sales manager at *AllElectronics*, and you are talking to a customer who recently bought a PC and a digital camera from the store. What should you recommend to her next? Information about which products are frequently purchased by your customers following their purchases of a PC and a digital camera in sequence would be very helpful in making your recommendation. Frequent patterns and association rules are the knowledge that you want to mine in such a scenario.

Frequent patterns are patterns (e.g., itemsets, subsequences, or substructures) that appear frequently in a data set. For example, a set of items, such as milk and bread, that appear frequently together in a transaction data set is a *frequent itemset*. A subsequence, such as buying first a PC, then a digital camera, and then a memory card, if it occurs frequently in a shopping history database, is a (*frequent*) *sequential pattern*. A *substructure* can refer to different structural forms, such as subgraphs, subtrees, or sublattices, which may be combined with itemsets or subsequences. If a substructure occurs frequently, it is called a (*frequent*) *structured pattern*. Finding frequent patterns plays an essential role in mining associations, correlations, and many other interesting relationships among data. Moreover, it helps in data classification, clustering, and other data mining tasks. Thus, frequent pattern mining has become an important data mining task and a focused theme in data mining research.

In this chapter, we introduce the basic concepts of frequent patterns, associations, and correlations (Section 6.1) and study how they can be mined efficiently (Section 6.2). We also discuss how to judge whether the patterns found are interesting (Section 6.3). In Chapter 7, we extend our discussion to advanced methods of frequent pattern mining, which mine more complex forms of frequent patterns and consider user preferences or constraints to speed up the mining process.

6.1 Basic Concepts

Frequent pattern mining searches for recurring relationships in a given data set. This section introduces the basic concepts of frequent pattern mining for the discovery of

interesting associations and correlations between itemsets in transactional and relational databases. We begin in Section 6.1.1 by presenting an example of market basket analysis, the earliest form of frequent pattern mining for association rules. The basic concepts of mining frequent patterns and associations are given in Section 6.1.2.

6.1.1 Market Basket Analysis: A Motivating Example

Frequent itemset mining leads to the discovery of associations and correlations among items in large transactional or relational data sets. With massive amounts of data continuously being collected and stored, many industries are becoming interested in mining such patterns from their databases. The discovery of interesting correlation relationships among huge amounts of business transaction records can help in many business decision-making processes such as catalog design, cross-marketing, and customer shopping behavior analysis.

A typical example of frequent itemset mining is **market basket analysis**. This process analyzes customer buying habits by finding associations between the different items that customers place in their "shopping baskets" (Figure 6.1). The discovery of these associations can help retailers develop marketing strategies by gaining insight into which items are frequently purchased together by customers. For instance, if customers are buying milk, how likely are they to also buy bread (and what kind of bread) on the same trip

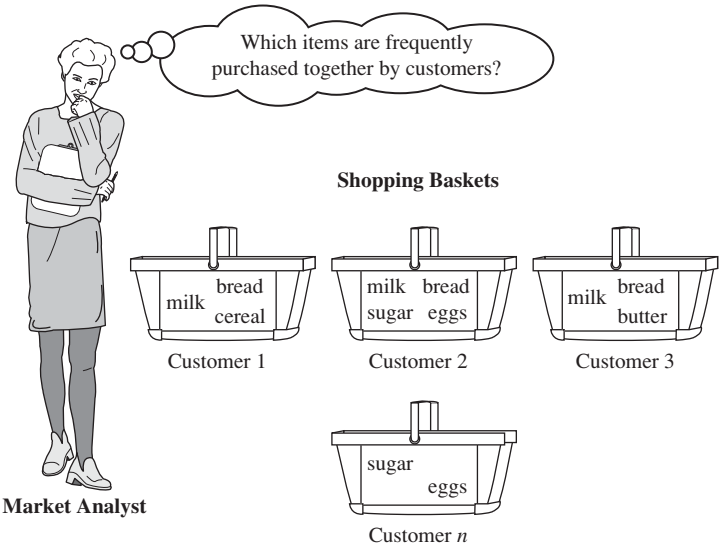


Figure 6.1 Market basket analysis.

to the supermarket? This information can lead to increased sales by helping retailers do selective marketing and plan their shelf space.

Let's look at an example of how market basket analysis can be useful.

Example 6.1 **Market basket analysis.** Suppose, as manager of an *AllElectronics* branch, you would like to learn more about the buying habits of your customers. Specifically, you wonder, "*Which groups or sets of items are customers likely to purchase on a given trip to the store?*" To answer your question, market basket analysis may be performed on the retail data of customer transactions at your store. You can then use the results to plan marketing or advertising strategies, or in the design of a new catalog. For instance, market basket analysis may help you design different store layouts. In one strategy, items that are frequently purchased together can be placed in proximity to further encourage the combined sale of such items. If customers who purchase computers also tend to buy antivirus software at the same time, then placing the hardware display close to the software display may help increase the sales of both items.

In an alternative strategy, placing hardware and software at opposite ends of the store may entice customers who purchase such items to pick up other items along the way. For instance, after deciding on an expensive computer, a customer may observe security systems for sale while heading toward the software display to purchase antivirus software, and may decide to purchase a home security system as well. Market basket analysis can also help retailers plan which items to put on sale at reduced prices. If customers tend to purchase computers and printers together, then having a sale on printers may encourage the sale of printers *as well as* computers. ■

If we think of the universe as the set of items available at the store, then each item has a Boolean variable representing the presence or absence of that item. Each basket can then be represented by a Boolean vector of values assigned to these variables. The Boolean vectors can be analyzed for buying patterns that reflect items that are frequently *associated* or purchased together. These patterns can be represented in the form of **association rules**. For example, the information that customers who purchase computers also tend to buy antivirus software at the same time is represented in the following association rule:

$$computer \Rightarrow antivirus\_software\ [support = 2\%, confidence = 60\%]. \qquad (6.1)$$

Rule **support** and **confidence** are two measures of rule interestingness. They respectively reflect the usefulness and certainty of discovered rules. A support of 2% for Rule (6.1) means that 2% of all the transactions under analysis show that computer and antivirus software are purchased together. A confidence of 60% means that 60% of the customers who purchased a computer also bought the software. Typically, association rules are considered interesting if they satisfy both a **minimum support threshold** and a **minimum confidence threshold**. These thresholds can be a set by users or domain experts. Additional analysis can be performed to discover interesting statistical correlations between associated items.

6.1.2 Frequent Itemsets, Closed Itemsets, and Association Rules

Let $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$ be an itemset. Let D, the task-relevant data, be a set of database transactions where each transaction T is a nonempty itemset such that $T \subseteq \mathcal{I}$. Each transaction is associated with an identifier, called a *TID*. Let A be a set of items. A transaction T is said to contain A if $A \subseteq T$. An association rule is an implication of the form $A \Rightarrow B$, where $A \subset \mathcal{I}$, $B \subset \mathcal{I}$, $A \neq \emptyset$, $B \neq \emptyset$, and $A \cap B = \phi$. The rule $A \Rightarrow B$ holds in the transaction set D with **support** s, where s is the percentage of transactions in D that contain $A \cup B$ (i.e., the *union* of sets A and B say, or, both A and B). This is taken to be the probability, $P(A \cup B)$.[1] The rule $A \Rightarrow B$ has **confidence** c in the transaction set D, where c is the percentage of transactions in D containing A that also contain B. This is taken to be the conditional probability, $P(B|A)$. That is,

$$support(A \Rightarrow B) = P(A \cup B) \tag{6.2}$$

$$confidence(A \Rightarrow B) = P(B|A). \tag{6.3}$$

Rules that satisfy both a minimum support threshold (*min_sup*) and a minimum confidence threshold (*min_conf*) are called **strong**. By convention, we write support and confidence values so as to occur between 0% and 100%, rather than 0 to 1.0.

A set of items is referred to as an **itemset**.[2] An itemset that contains k items is a ***k*-itemset**. The set {*computer, antivirus_software*} is a 2-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. Note that the itemset support defined in Eq. (6.2) is sometimes referred to as *relative support*, whereas the occurrence frequency is called the **absolute support**. If the relative support of an itemset I satisfies a prespecified **minimum support threshold** (i.e., the absolute support of I satisfies the corresponding **minimum support count threshold**), then I is a **frequent** itemset.[3] The set of frequent k-itemsets is commonly denoted by L_k.[4]

From Eq. (6.3), we have

$$confidence(A \Rightarrow B) = P(B|A) = \frac{support(A \cup B)}{support(A)} = \frac{support\_count(A \cup B)}{support\_count(A)}. \tag{6.4}$$

[1] Notice that the notation $P(A \cup B)$ indicates the probability that a transaction contains the *union* of sets A and B (i.e., it contains every item in A and B). This should not be confused with $P(A \text{ or } B)$, which indicates the probability that a transaction contains either A or B.

[2] In the data mining research literature, "itemset" is more commonly used than "item set."

[3] In early work, itemsets satisfying minimum support were referred to as **large**. This term, however, is somewhat confusing as it has connotations of the number of items in an itemset rather than the frequency of occurrence of the set. Hence, we use the more recent term **frequent**.

[4] Although the term **frequent** is preferred over **large**, for historic reasons frequent k-itemsets are still denoted as L_k.

Equation (6.4) shows 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.

Additional interestingness measures can be applied for the discovery of correlation relationships between associated items, as will be discussed in Section 6.3. Because the second step is much less costly than the first, the overall performance of mining association rules is determined by the first step.

A major challenge in mining frequent itemsets from a large data set is the fact that such mining often generates a huge number of itemsets satisfying the minimum support (*min_sup*) threshold, especially when *min_sup* is set low. This is because if an itemset is frequent, each of its subsets is frequent as well. A long itemset will contain a combinatorial number of shorter, frequent sub-itemsets. For example, a frequent itemset of length 100, such as $\{a_1, a_2, \ldots, a_{100}\}$, contains $\binom{100}{1} = 100$ frequent 1-itemsets: $\{a_1\}$, $\{a_2\}$, $\ldots$, $\{a_{100}\}$; $\binom{100}{2}$ frequent 2-itemsets: $\{a_1, a_2\}$, $\{a_1, a_3\}, \ldots, \{a_{99}, a_{100}\}$; and so on. The total number of frequent itemsets that it contains is thus

$$\binom{100}{1} + \binom{100}{2} + \cdots + \binom{100}{100} = 2^{100} - 1 \approx 1.27 \times 10^{30}. \tag{6.5}$$

This is too huge a number of itemsets for any computer to compute or store. To overcome this difficulty, we introduce the concepts of *closed frequent itemset* and *maximal frequent itemset*.

An itemset X is **closed** in a data set D if there exists no proper super-itemset Y[5] such that Y has the same support count as X in D. An itemset X is a **closed frequent itemset** in set D if X is both closed and frequent in D. An itemset X is a **maximal frequent itemset** (or **max-itemset**) in a data set D if X is frequent, and there exists no super-itemset Y such that $X \subset Y$ and Y is frequent in D.

Let $\mathcal{C}$ be the set of closed frequent itemsets for a data set D satisfying a minimum support threshold, *min_sup*. Let $\mathcal{M}$ be the set of maximal frequent itemsets for D satisfying *min_sup*. Suppose that we have the support count of each itemset in $\mathcal{C}$ and $\mathcal{M}$. Notice that $\mathcal{C}$ and its count information can be used to derive the whole set of frequent itemsets.

[5] Y is a proper super-itemset of X if X is a proper sub-itemset of Y, that is, if $X \subset Y$. In other words, every item of X is contained in Y but there is at least one item of Y that is not in X.

Thus, we say that $\mathcal{C}$ contains complete information regarding its corresponding frequent itemsets. On the other hand, $\mathcal{M}$ registers only the support of the maximal itemsets. It usually does not contain the complete support information regarding its corresponding frequent itemsets. We illustrate these concepts with Example 6.2.

Example 6.2 **Closed and maximal frequent itemsets.** Suppose that a transaction database has only two transactions: $\{\langle a_1, a_2, \ldots, a_{100}\rangle; \langle a_1, a_2, \ldots, a_{50}\rangle\}$. Let the minimum support count threshold be $min\_sup = 1$. We find two closed frequent itemsets and their support counts, that is, $\mathcal{C} = \{\{a_1, a_2, \ldots, a_{100}\} : 1; \{a_1, a_2, \ldots, a_{50}\} : 2\}$. There is only one maximal frequent itemset: $\mathcal{M} = \{\{a_1, a_2, \ldots, a_{100}\} : 1\}$. Notice that we cannot include $\{a_1, a_2, \ldots, a_{50}\}$ as a maximal frequent itemset because it has a frequent superset, $\{a_1, a_2, \ldots, a_{100}\}$. Compare this to the preceding where we determined that there are $2^{100} - 1$ frequent itemsets, which are too many to be enumerated!

The set of closed frequent itemsets contains complete information regarding the frequent itemsets. For example, from $\mathcal{C}$, we can derive, say, (1) $\{a_2, a_{45} : 2\}$ since $\{a_2, a_{45}\}$ is a sub-itemset of the itemset $\{a_1, a_2, \ldots, a_{50} : 2\}$; and (2) $\{a_8, a_{55} : 1\}$ since $\{a_8, a_{55}\}$ is not a sub-itemset of the previous itemset but of the itemset $\{a_1, a_2, \ldots, a_{100} : 1\}$. However, from the maximal frequent itemset, we can only assert that both itemsets ($\{a_2, a_{45}\}$ and $\{a_8, a_{55}\}$) are frequent, but we cannot assert their actual support counts. ■

6.2 Frequent Itemset Mining Methods

In this section, you will learn methods for mining the simplest form of frequent patterns such as those discussed for market basket analysis in Section 6.1.1. We begin by presenting **Apriori**, the basic algorithm for finding frequent itemsets (Section 6.2.1). In Section 6.2.2, we look at how to generate strong association rules from frequent itemsets. Section 6.2.3 describes several variations to the Apriori algorithm for improved efficiency and scalability. Section 6.2.4 presents pattern-growth methods for mining frequent itemsets that confine the subsequent search space to only the data sets containing the current frequent itemsets. Section 6.2.5 presents methods for mining frequent itemsets that take advantage of the vertical data format.

6.2.1 Apriori Algorithm: Finding Frequent Itemsets by Confined Candidate Generation

Apriori is a seminal algorithm proposed by R. Agrawal and R. Srikant in 1994 for mining frequent itemsets for Boolean association rules [AS94b]. The name of the algorithm is based on the fact that the algorithm uses *prior knowledge* of frequent itemset properties, as we shall see later. Apriori employs an iterative approach known as a *level-wise* search, where k-itemsets are used to explore $(k+1)$-itemsets. First, the set of frequent 1-itemsets is found by scanning the database to accumulate the count for each item, and

collecting those items that satisfy minimum support. The resulting set is denoted by L_1. Next, L_1 is used to find L_2, the set of frequent 2-itemsets, which is used to find L_3, and so on, until no more frequent k-itemsets can be found. The finding of each L_k requires one full scan of the database.

To improve the efficiency of the level-wise generation of frequent itemsets, an important property called the **Apriori property** is used to reduce the search space.

Apriori property: *All nonempty subsets of a frequent itemset must also be frequent.*

The Apriori property is based on the following observation. By definition, if an itemset I does not satisfy the minimum support threshold, *min_sup*, then I is not frequent, that is, $P(I) < min\_sup$. If an item A is added to the itemset I, then the resulting itemset (i.e., $I \cup A$) cannot occur more frequently than I. Therefore, $I \cup A$ is not frequent either, that is, $P(I \cup A) < min\_sup$.

This property belongs to a special category of properties called **antimonotonicity** in the sense that *if a set cannot pass a test, all of its supersets will fail the same test as well.* It is called *antimonotonicity* because the property is monotonic in the context of failing a test.[6]

"*How is the Apriori property used in the algorithm?*" To understand this, let us look at how L_{k-1} is used to find L_k for $k \geq 2$. A two-step process is followed, consisting of **join** and **prune** actions.

1. **The join step**: To find L_k, a set of **candidate** k-itemsets is generated by joining L_{k-1} with itself. This set of candidates is denoted C_k. Let l_1 and l_2 be itemsets in L_{k-1}. The notation $l_i[j]$ refers to the jth item in l_i (e.g., $l_1[k-2]$ refers to the second to the last item in l_1). For efficient implementation, Apriori assumes that items within a transaction or itemset are sorted in lexicographic order. For the $(k-1)$-itemset, l_i, this means that the items are sorted such that $l_i[1] < l_i[2] < \cdots < l_i[k-1]$. The join, $L_{k-1} \bowtie L_{k-1}$, is performed, where members of L_{k-1} are joinable if their first $(k-2)$ items are in common. That is, members l_1 and l_2 of L_{k-1} are joined if $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge \cdots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$. The condition $l_1[k-1] < l_2[k-1]$ simply ensures that no duplicates are generated. The resulting itemset formed by joining l_1 and l_2 is $\{l_1[1], l_1[2], \ldots, l_1[k-2], l_1[k-1], l_2[k-1]\}$.
2. **The prune step**: C_k is a superset of L_k, that is, its members may or may not be frequent, but all of the frequent k-itemsets are included in C_k. A database scan to determine the count of each candidate in C_k would result in the determination of L_k (i.e., all candidates having a count no less than the minimum support count are frequent by definition, and therefore belong to L_k). C_k, however, can be huge, and so this could involve heavy computation. To reduce the size of C_k, the Apriori property

[6]The Apriori property has many applications. For example, it can also be used to prune search during data cube computation (Chapter 5).

is used as follows. Any $(k-1)$-itemset that is not frequent cannot be a subset of a frequent k-itemset. Hence, if any $(k-1)$-subset of a candidate k-itemset is not in L_{k-1}, then the candidate cannot be frequent either and so can be removed from C_k. This **subset testing** can be done quickly by maintaining a hash tree of all frequent itemsets.

Example 6.3 **Apriori.** Let's look at a concrete example, based on the *AllElectronics* transaction database, D, of Table 6.1. There are nine transactions in this database, that is, $|D| = 9$. We use Figure 6.2 to illustrate the Apriori algorithm for finding frequent itemsets in D.

1. In the first iteration of the algorithm, each item is a member of the set of candidate 1-itemsets, C_1. The algorithm simply scans all of the transactions to count the number of occurrences of each item.
2. Suppose that the minimum support count required is 2, that is, $min\_sup = 2$. (Here, we are referring to *absolute* support because we are using a support count. The corresponding relative support is $2/9 = 22\%$.) The set of frequent 1-itemsets, L_1, can then be determined. It consists of the candidate 1-itemsets satisfying minimum support. In our example, all of the candidates in C_1 satisfy minimum support.
3. To discover the set of frequent 2-itemsets, L_2, the algorithm uses the join $L_1 \bowtie L_1$ to generate a candidate set of 2-itemsets, C_2.[7] C_2 consists of $\binom{|L_1|}{2}$ 2-itemsets. Note that no candidates are removed from C_2 during the prune step because each subset of the candidates is also frequent.

Table 6.1 Transactional Data for an *AllElectronics* Branch

TID	*List of item_IDs*
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

[7] $L_1 \bowtie L_1$ is equivalent to $L_1 \times L_1$, since the definition of $L_k \bowtie L_k$ requires the two joining itemsets to share $k-1 = 0$ items.

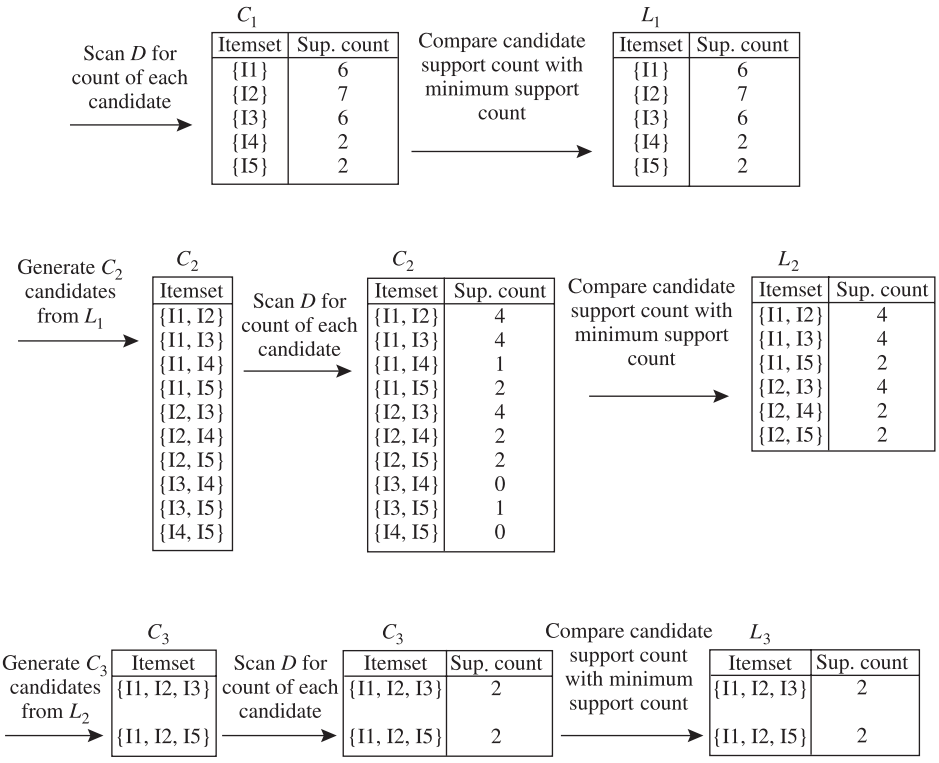


Figure 6.2 Generation of the candidate itemsets and frequent itemsets, where the minimum support count is 2.

4. Next, the transactions in D are scanned and the support count of each candidate itemset in C_2 is accumulated, as shown in the middle table of the second row in Figure 6.2.
5. The set of frequent 2-itemsets, L_2, is then determined, consisting of those candidate 2-itemsets in C_2 having minimum support.
6. The generation of the set of the candidate 3-itemsets, C_3, is detailed in Figure 6.3. From the join step, we first get $C_3 = L_2 \bowtie L_2 = \{\{I1, I2, I3\}, \{I1, I2, I5\}, \{I1, I3, I5\}, \{I2, I3, I4\}, \{I2, I3, I5\}, \{I2, I4, I5\}\}$. Based on the Apriori property that all subsets of a frequent itemset must also be frequent, we can determine that the four latter candidates cannot possibly be frequent. We therefore remove them from C_3, thereby saving the effort of unnecessarily obtaining their counts during the subsequent scan of D to determine L_3. Note that when given a candidate k-itemset, we only need to check if its $(k-1)$-subsets are frequent since the Apriori algorithm uses a level-wise

(**a**) Join: $C_3 = L_2 \bowtie L_2 = \{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\}$
$\bowtie \{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\}$
$= \{\{I1, I2, I3\}, \{I1, I2, I5\}, \{I1, I3, I5\}, \{I2, I3, I4\}, \{I2, I3, I5\}, \{I2, I4, I5\}\}$.

(**b**) Prune using the Apriori property: All nonempty subsets of a frequent itemset must also be frequent. Do any of the candidates have a subset that is not frequent?

- The 2-item subsets of {I1, I2, I3} are {I1, I2}, {I1, I3}, and {I2, I3}. All 2-item subsets of {I1, I2, I3} are members of L_2. Therefore, keep {I1, I2, I3} in C_3.
- The 2-item subsets of {I1, I2, I5} are {I1, I2}, {I1, I5}, and {I2, I5}. All 2-item subsets of {I1, I2, I5} are members of L_2. Therefore, keep {I1, I2, I5} in C_3.
- The 2-item subsets of {I1, I3, I5} are {I1, I3}, {I1, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I1, I3, I5} from C_3.
- The 2-item subsets of {I2, I3, I4} are {I2, I3}, {I2, I4}, and {I3, I4}. {I3, I4} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I4} from C_3.
- The 2-item subsets of {I2, I3, I5} are {I2, I3}, {I2, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I5} from C_3.
- The 2-item subsets of {I2, I4, I5} are {I2, I4}, {I2, I5}, and {I4, I5}. {I4, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I4, I5} from C_3.

(**c**) Therefore, $C_3 = \{\{I1, I2, I3\}, \{I1, I2, I5\}\}$ after pruning.

Figure 6.3 Generation and pruning of candidate 3-itemsets, C_3, from L_2 using the Apriori property.

search strategy. The resulting pruned version of C_3 is shown in the first table of the bottom row of Figure 6.2.

7. The transactions in D are scanned to determine L_3, consisting of those candidate 3-itemsets in C_3 having minimum support (Figure 6.2).

8. The algorithm uses $L_3 \bowtie L_3$ to generate a candidate set of 4-itemsets, C_4. Although the join results in {{I1, I2, I3, I5}}, itemset {I1, I2, I3, I5} is pruned because its subset {I2, I3, I5} is not frequent. Thus, $C_4 = \phi$, and the algorithm terminates, having found all of the frequent itemsets. ■

Figure 6.4 shows pseudocode for the Apriori algorithm and its related procedures. Step 1 of Apriori finds the frequent 1-itemsets, L_1. In steps 2 through 10, L_{k-1} is used to generate candidates C_k to find L_k for $k \geq 2$. The apriori_gen procedure generates the candidates and then uses the Apriori property to eliminate those having a subset that is not frequent (step 3). This procedure is described later. Once all of the candidates have been generated, the database is scanned (step 4). For each transaction, a subset function is used to find all subsets of the transaction that are candidates (step 5), and the count for each of these candidates is accumulated (steps 6 and 7). Finally, all the candidates satisfying the minimum support (step 9) form the set of frequent itemsets, L (step 11).

Algorithm: Apriori. Find frequent itemsets using an iterative level-wise approach based on candidate generation.

Input:

- D, a database of transactions;
- *min_sup*, the minimum support count threshold.

Output: L, frequent itemsets in D.

Method:

```
(1)   L1 = find_frequent_1-itemsets(D);
(2)   for (k = 2; Lk−1 ≠ φ; k++) {
(3)       Ck = apriori_gen(Lk−1);
(4)       for each transaction t ∈ D { // scan D for counts
(5)           Ct = subset(Ck, t); // get the subsets of t that are candidates
(6)           for each candidate c ∈ Ct
(7)               c.count++;
(8)       }
(9)       Lk = {c ∈ Ck|c.count ≥ min_sup}
(10)  }
(11)  return L = ∪k Lk;

procedure apriori_gen(Lk−1:frequent (k − 1)-itemsets)
(1)   for each itemset l1 ∈ Lk−1
(2)       for each itemset l2 ∈ Lk−1
(3)           if (l1[1] = l2[1]) ∧ (l1[2] = l2[2])
                  ∧... ∧ (l1[k − 2] = l2[k − 2]) ∧ (l1[k − 1] < l2[k − 1]) then {
(4)               c = l1 ⋈ l2; // join step: generate candidates
(5)               if has_infrequent_subset(c, Lk−1) then
(6)                   delete c; // prune step: remove unfruitful candidate
(7)               else add c to Ck;
(8)           }
(9)   return Ck;

procedure has_infrequent_subset(c: candidate k-itemset;
          Lk−1: frequent (k − 1)-itemsets); // use prior knowledge
(1)   for each (k − 1)-subset s of c
(2)       if s ∉ Lk−1 then
(3)           return TRUE;
(4)   return FALSE;
```

Figure 6.4 Apriori algorithm for discovering frequent itemsets for mining Boolean association rules.

A procedure can then be called to generate association rules from the frequent itemsets. Such a procedure is described in Section 6.2.2.

The apriori_gen procedure performs two kinds of actions, namely, **join** and **prune**, as described before. In the join component, L_{k-1} is joined with L_{k-1} to generate potential candidates (steps 1–4). The prune component (steps 5–7) employs the Apriori property to remove candidates that have a subset that is not frequent. The test for infrequent subsets is shown in procedure has_infrequent_subset.

6.2.2 Generating Association Rules from Frequent Itemsets

Once the frequent itemsets from transactions in a database D have been found, it is straightforward to generate strong association rules from them (where *strong* association rules satisfy both minimum support and minimum confidence). This can be done using Eq. (6.4) for confidence, which we show again here for completeness:

$$confidence(A \Rightarrow B) = P(B|A) = \frac{support\_count(A \cup B)}{support\_count(A)}.$$

The conditional probability is expressed in terms of itemset support count, where $support\_count(A \cup B)$ is the number of transactions containing the itemsets $A \cup B$, and $support\_count(A)$ is the number of transactions containing the itemset A. Based on this equation, association rules can be generated as follows:

- For each frequent itemset l, generate all nonempty subsets of l.
- For every nonempty subset s of l, output the rule "$s \Rightarrow (l - s)$" if $\frac{support\_count(l)}{support\_count(s)} \geq min\_conf$, where min_conf is the minimum confidence threshold.

Because the rules are generated from frequent itemsets, each one automatically satisfies the minimum support. Frequent itemsets can be stored ahead of time in hash tables along with their counts so that they can be accessed quickly.

Example 6.4 **Generating association rules.** Let's try an example based on the transactional data for *AllElectronics* shown before in Table 6.1. The data contain frequent itemset $X = \{I1, I2, I5\}$. What are the association rules that can be generated from X? The nonempty subsets of X are {I1, I2}, {I1, I5}, {I2, I5}, {I1}, {I2}, and {I5}. The resulting association rules are as shown below, each listed with its confidence:

$$\begin{aligned}
&\{I1, I2\} \Rightarrow I5, && confidence = 2/4 = 50\% \\
&\{I1, I5\} \Rightarrow I2, && confidence = 2/2 = 100\% \\
&\{I2, I5\} \Rightarrow I1, && confidence = 2/2 = 100\% \\
&I1 \Rightarrow \{I2, I5\}, && confidence = 2/6 = 33\% \\
&I2 \Rightarrow \{I1, I5\}, && confidence = 2/7 = 29\% \\
&I5 \Rightarrow \{I1, I2\}, && confidence = 2/2 = 100\%
\end{aligned}$$

If the minimum confidence threshold is, say, 70%, then only the second, third, and last rules are output, because these are the only ones generated that are strong. Note that, unlike conventional classification rules, association rules can contain more than one conjunct in the right side of the rule. ■

6.2.3 Improving the Efficiency of Apriori

"*How can we further improve the efficiency of Apriori-based mining?*" Many variations of the Apriori algorithm have been proposed that focus on improving the efficiency of the original algorithm. Several of these variations are summarized as follows:

Create hash table H_2 using hash function $h(x, y) = ((order\ of\ x) \times 10 + (order\ of\ y))\ mod\ 7$

H_2

bucket address	0	1	2	3	4	5	6
bucket count	2	2	4	2	2	4	4
bucket contents	{I1, I4} {I3, I5}	{I1, I5} {I1, I5}	{I2, I3} {I2, I3} {I2, I3} {I2, I3}	{I2, I4} {I2, I4}	{I2, I5} {I2, I5}	{I1, I2} {I1, I2} {I1, I2} {I1, I2}	{I1, I3} {I1, I3} {I1, I3} {I1, I3}

Figure 6.5 Hash table, H_2, for candidate 2-itemsets. This hash table was generated by scanning Table 6.1's transactions while determining L_1. If the minimum support count is, say, 3, then the itemsets in buckets 0, 1, 3, and 4 cannot be frequent and so they should not be included in C_2.

Hash-based technique (hashing itemsets into corresponding buckets): A hash-based technique can be used to reduce the size of the candidate k-itemsets, C_k, for $k > 1$. For example, when scanning each transaction in the database to generate the frequent 1-itemsets, L_1, we can generate all the 2-itemsets for each transaction, hash (i.e., map) them into the different *buckets* of a *hash table* structure, and increase the corresponding bucket counts (Figure 6.5). A 2-itemset with a corresponding bucket count in the hash table that is below the support threshold cannot be frequent and thus should be removed from the candidate set. Such a hash-based technique may substantially reduce the number of candidate k-itemsets examined (especially when $k = 2$).

Transaction reduction (reducing the number of transactions scanned in future iterations): A transaction that does not contain any frequent k-itemsets cannot contain any frequent $(k+1)$-itemsets. Therefore, such a transaction can be marked or removed from further consideration because subsequent database scans for j-itemsets, where $j > k$, will not need to consider such a transaction.

Partitioning (partitioning the data to find candidate itemsets): A partitioning technique can be used that requires just two database scans to mine the frequent itemsets (Figure 6.6). It consists of two phases. In phase I, the algorithm divides the transactions of D into n nonoverlapping partitions. If the minimum relative support threshold for transactions in D is *min_sup*, then the minimum support count for a partition is *min_sup* $\times$ *the number of transactions in that partition.* For each partition, all the *local frequent itemsets* (i.e., the itemsets frequent within the partition) are found.

A local frequent itemset may or may not be frequent with respect to the entire database, D. However, *any itemset that is potentially frequent with respect to D must occur as a frequent itemset in at least one of the partitions.*[8] Therefore, all local frequent itemsets are candidate itemsets with respect to D. The collection of frequent itemsets from all partitions forms the *global candidate itemsets* with respect to D. In phase II,

[8]The proof of this property is left as an exercise (see Exercise 6.3d).

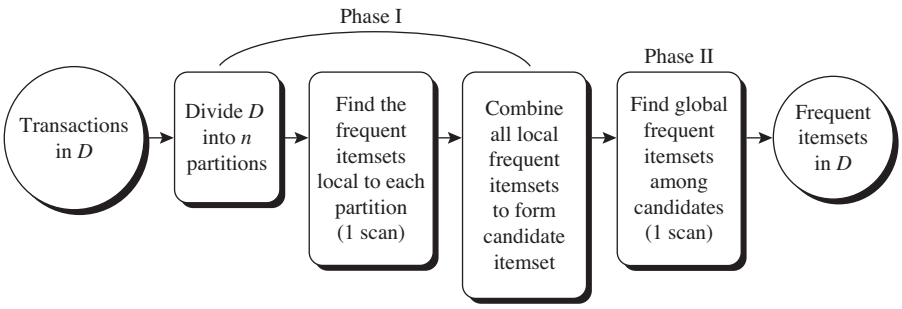


Figure 6.6 Mining by partitioning the data.

a second scan of D is conducted in which the actual support of each candidate is assessed to determine the global frequent itemsets. Partition size and the number of partitions are set so that each partition can fit into main memory and therefore be read only once in each phase.

Sampling (mining on a subset of the given data): The basic idea of the sampling approach is to pick a random sample S of the given data D, and then search for frequent itemsets in S instead of D. In this way, we trade off some degree of accuracy against efficiency. The S sample size is such that the search for frequent itemsets in S can be done in main memory, and so only one scan of the transactions in S is required overall. Because we are searching for frequent itemsets in S rather than in D, it is possible that we will miss some of the global frequent itemsets.

To reduce this possibility, we use a lower support threshold than minimum support to find the frequent itemsets local to S (denoted L^S). The rest of the database is then used to compute the actual frequencies of each itemset in L^S. A mechanism is used to determine whether all the global frequent itemsets are included in L^S. If L^S actually contains all the frequent itemsets in D, then only one scan of D is required. Otherwise, a second pass can be done to find the frequent itemsets that were missed in the first pass. The sampling approach is especially beneficial when efficiency is of utmost importance such as in computationally intensive applications that must be run frequently.

Dynamic itemset counting (adding candidate itemsets at different points during a scan): A dynamic itemset counting technique was proposed in which the database is partitioned into blocks marked by start points. In this variation, new candidate itemsets can be added at any start point, unlike in Apriori, which determines new candidate itemsets only immediately before each complete database scan. The technique uses the count-so-far as the lower bound of the actual count. If the count-so-far passes the minimum support, the itemset is added into the frequent itemset collection and can be used to generate longer candidates. This leads to fewer database scans than with Apriori for finding all the frequent itemsets.

Other variations are discussed in the next chapter.

6.2.4 A Pattern-Growth Approach for Mining Frequent Itemsets

As we have seen, in many cases the Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain. However, it can suffer from two nontrivial costs:

- *It may still need to generate a huge number of candidate sets.* For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets.
- *It may need to repeatedly scan the whole database and check a large set of candidates by pattern matching.* It is costly to go over each transaction in the database to determine the support of the candidate itemsets.

"Can we design a method that mines the complete set of frequent itemsets without such a costly candidate generation process?" An interesting method in this attempt is called **frequent pattern growth,** or simply **FP-growth**, which adopts a *divide-and-conquer* strategy as follows. First, it compresses the database representing frequent items into a **frequent pattern tree,** or **FP-tree**, which retains the itemset association information. It then divides the compressed database into a set of *conditional databases* (a special kind of projected database), each associated with one frequent item or "pattern fragment," and mines each database separately. For each "pattern fragment," only its associated data sets need to be examined. Therefore, this approach may substantially reduce the size of the data sets to be searched, along with the "growth" of patterns being examined. You will see how it works in Example 6.5.

Example 6.5 **FP-growth (finding frequent itemsets without candidate generation).** We reexamine the mining of transaction database, *D*, of Table 6.1 in Example 6.3 using the frequent pattern growth approach.

The first scan of the database is the same as Apriori, which derives the set of frequent items (1-itemsets) and their support counts (frequencies). Let the minimum support count be 2. The set of frequent items is sorted in the order of descending support count. This resulting set or *list* is denoted by *L*. Thus, we have $L = \{\{\text{I2: 7}\}, \{\text{I1: 6}\}, \{\text{I3: 6}\}, \{\text{I4: 2}\}, \{\text{I5: 2}\}\}$.

An FP-tree is then constructed as follows. First, create the root of the tree, labeled with "null." Scan database *D* a second time. The items in each transaction are processed in *L* order (i.e., sorted according to descending support count), and a branch is created for each transaction. For example, the scan of the first transaction, "T100: I1, I2, I5," which contains three items (I2, I1, I5 in *L* order), leads to the construction of the first branch of the tree with three nodes, ⟨I2: 1⟩, ⟨I1: 1⟩, and ⟨I5: 1⟩, where I2 is linked as a child to the root, I1 is linked to I2, and I5 is linked to I1. The second transaction, T200, contains the items I2 and I4 in *L* order, which would result in a branch where I2 is linked to the root and I4 is linked to I2. However, this branch would share a common **prefix,** I2, with the existing path for T100. Therefore, we instead increment the count of the I2 node by 1, and create a new node, ⟨I4: 1⟩, which is linked as a child to ⟨I2: 2⟩. In general,

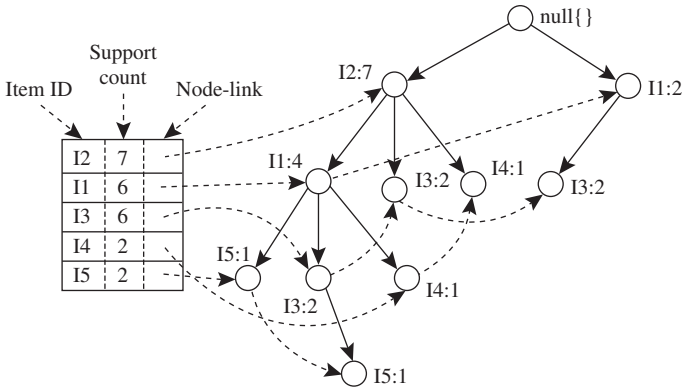


Figure 6.7 An FP-tree registers compressed, frequent pattern information.

when considering the branch to be added for a transaction, the count of each node along a common prefix is incremented by 1, and nodes for the items following the prefix are created and linked accordingly.

To facilitate tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of **node-links**. The tree obtained after scanning all the transactions is shown in Figure 6.7 with the associated node-links. In this way, the problem of mining frequent patterns in databases is transformed into that of mining the FP-tree.

The FP-tree is mined as follows. Start from each frequent length-1 pattern (as an initial **suffix pattern**), construct its **conditional pattern base** (a "sub-database," which consists of the set of *prefix paths* in the FP-tree co-occurring with the suffix pattern), then construct its (*conditional*) FP-tree, and perform mining recursively on the tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

Mining of the FP-tree is summarized in Table 6.2 and detailed as follows. We first consider I5, which is the last item in L, rather than the first. The reason for starting at the end of the list will become apparent as we explain the FP-tree mining process. I5 occurs in two FP-tree branches of Figure 6.7. (The occurrences of I5 can easily be found by following its chain of node-links.) The paths formed by these branches are ⟨I2, I1, I5: 1⟩ and ⟨I2, I1, I3, I5: 1⟩. Therefore, considering I5 as a suffix, its corresponding two prefix paths are ⟨I2, I1: 1⟩ and ⟨I2, I1, I3: 1⟩, which form its conditional pattern base. Using this conditional pattern base as a transaction database, we build an I5-conditional FP-tree, which contains only a single path, ⟨I2: 2, I1: 2⟩; I3 is not included because its support count of 1 is less than the minimum support count. The single path generates all the combinations of frequent patterns: {I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}.

For I4, its two prefix paths form the conditional pattern base, {{I2 I1: 1}, {I2: 1}}, which generates a single-node conditional FP-tree, ⟨I2: 2⟩, and derives one frequent pattern, {I2, I4: 2}.

Table 6.2 Mining the FP-Tree by Creating Conditional (Sub-)Pattern Bases

Item	*Conditional Pattern Base*	*Conditional FP-tree*	*Frequent Patterns Generated*
I5	{{I2, I1: 1}, {I2, I1, I3: 1}}	⟨I2: 2, I1: 2⟩	{I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}
I4	{{I2, I1: 1}, {I2: 1}}	⟨I2: 2⟩	{I2, I4: 2}
I3	{{I2, I1: 2}, {I2: 2}, {I1: 2}}	⟨I2: 4, I1: 2⟩, ⟨I1: 2⟩	{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}
I1	{{I2: 4}}	⟨I2: 4⟩	{I2, I1: 4}

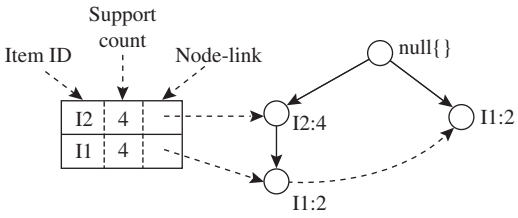


Figure 6.8 The conditional FP-tree associated with the conditional node I3.

Similar to the preceding analysis, I3's conditional pattern base is {{I2, I1: 2}, {I2: 2}, {I1: 2}}. Its conditional FP-tree has two branches, ⟨I2: 4, I1: 2⟩ and ⟨I1: 2⟩, as shown in Figure 6.8, which generates the set of patterns {{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}}. Finally, I1's conditional pattern base is {{I2: 4}}, with an FP-tree that contains only one node, ⟨I2: 4⟩, which generates one frequent pattern, {I2, I1: 4}. This mining process is summarized in Figure 6.9. ■

The FP-growth method transforms the problem of finding long frequent patterns into searching for shorter ones in much smaller conditional databases recursively and then concatenating the suffix. It uses the least frequent items as a suffix, offering good selectivity. The method substantially reduces the search costs.

When the database is large, it is sometimes unrealistic to construct a main memory-based FP-tree. An interesting alternative is to first partition the database into a set of projected databases, and then construct an FP-tree and mine it in each projected database. This process can be recursively applied to any projected database if its FP-tree still cannot fit in main memory.

A study of the FP-growth method performance shows that it is efficient and scalable for mining both long and short frequent patterns, and is about an order of magnitude faster than the Apriori algorithm.

6.2.5 Mining Frequent Itemsets Using the Vertical Data Format

Both the Apriori and FP-growth methods mine frequent patterns from a set of transactions in *TID-itemset* format (i.e., {*TID* : *itemset*}), where *TID* is a transaction ID and *itemset* is the set of items bought in transaction *TID*. This is known as the **horizontal data format**. Alternatively, data can be presented in *item-TID_set* format

Algorithm: **FP_growth.** Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

- D, a transaction database;
- *min_sup*, the minimum support count threshold.

Output: The complete set of frequent patterns.

Method:

1. The FP-tree is constructed in the following steps:
   (a) Scan the transaction database D once. Collect F, the set of frequent items, and their support counts. Sort F in support count descending order as L, the *list* of frequent items.
   (b) Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in D do the following.
   Select and sort the frequent items in *Trans* according to the order of L. Let the sorted frequent item list in *Trans* be $[p|P]$, where p is the first element and P is the remaining list. Call insert_tree($[p|P]$, T), which is performed as follows. If T has a child N such that $N.item\text{-}name = p.item\text{-}name$, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call insert_tree(P, N) recursively.
2. The FP-tree is mined by calling FP_growth(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth(Tree, α)
(1)  if Tree contains a single path P then
(2)     for each combination (denoted as β) of the nodes in the path P
(3)        generate pattern β ∪ α with support_count = minimum support count of nodes in β;
(4)  else for each a_i in the header of Tree {
(5)     generate pattern β = a_i ∪ α with support_count = a_i.support_count;
(6)     construct β's conditional pattern base and then β's conditional FP_tree Tree_β;
(7)     if Tree_β ≠ ∅ then
(8)        call FP_growth(Tree_β, β); }
```

Figure 6.9 FP-growth algorithm for discovering frequent itemsets without candidate generation.

(i.e., {*item* : *TID_set*}), where *item* is an item name, and *TID_set* is the set of transaction identifiers containing the item. This is known as the **vertical data format**.

In this subsection, we look at how frequent itemsets can also be mined efficiently using vertical data format, which is the essence of the **Eclat** (Equivalence Class Transformation) algorithm.

Example 6.6 **Mining frequent itemsets using the vertical data format.** Consider the horizontal data format of the transaction database, D, of Table 6.1 in Example 6.3. This can be transformed into the vertical data format shown in Table 6.3 by scanning the data set once.

Mining can be performed on this data set by intersecting the TID_sets of every pair of frequent single items. The minimum support count is 2. Because every single item is

Table 6.3 The Vertical Data Format of the Transaction Data Set D of Table 6.1

itemset	*TID_set*
I1	{T100, T400, T500, T700, T800, T900}
I2	{T100, T200, T300, T400, T600, T800, T900}
I3	{T300, T500, T600, T700, T800, T900}
I4	{T200, T400}
I5	{T100, T800}

Table 6.4 2-Itemsets in Vertical Data Format

itemset	*TID_set*
{I1, I2}	{T100, T400, T800, T900}
{I1, I3}	{T500, T700, T800, T900}
{I1, I4}	{T400}
{I1, I5}	{T100, T800}
{I2, I3}	{T300, T600, T800, T900}
{I2, I4}	{T200, T400}
{I2, I5}	{T100, T800}
{I3, I5}	{T800}

Table 6.5 3-Itemsets in Vertical Data Format

itemset	*TID_set*
{I1, I2, I3}	{T800, T900}
{I1, I2, I5}	{T100, T800}

frequent in Table 6.3, there are 10 intersections performed in total, which lead to eight nonempty 2-itemsets, as shown in Table 6.4. Notice that because the itemsets {I1, I4} and {I3, I5} each contain only one transaction, they do not belong to the set of frequent 2-itemsets.

Based on the Apriori property, a given 3-itemset is a candidate 3-itemset only if every one of its 2-itemset subsets is frequent. The candidate generation process here will generate only two 3-itemsets: {I1, I2, I3} and {I1, I2, I5}. By intersecting the TID_sets of any two corresponding 2-itemsets of these candidate 3-itemsets, it derives Table 6.5, where there are only two frequent 3-itemsets: {I1, I2, I3: 2} and {I1, I2, I5: 2}. ■

Example 6.6 illustrates the process of mining frequent itemsets by exploring the vertical data format. First, we transform the horizontally formatted data into the vertical format by scanning the data set once. The support count of an itemset is simply the length of the TID_set of the itemset. Starting with $k = 1$, the frequent k-itemsets can be used to construct the candidate $(k + 1)$-itemsets based on the Apriori property.

The computation is done by intersection of the TID_sets of the frequent k-itemsets to compute the TID_sets of the corresponding $(k+1)$-itemsets. This process repeats, with k incremented by 1 each time, until no frequent itemsets or candidate itemsets can be found.

Besides taking advantage of the Apriori property in the generation of candidate $(k+1)$-itemset from frequent k-itemsets, another merit of this method is that there is no need to scan the database to find the support of $(k+1)$-itemsets (for $k \geq 1$). This is because the TID_set of each k-itemset carries the complete information required for counting such support. However, the TID_sets can be quite long, taking substantial memory space as well as computation time for intersecting the long sets.

To further reduce the cost of registering long TID_sets, as well as the subsequent costs of intersections, we can use a technique called *diffset*, which keeps track of only the differences of the TID_sets of a $(k+1)$-itemset and a corresponding k-itemset. For instance, in Example 6.6 we have {I1} = {T100, T400, T500, T700, T800, T900} and {I1, I2} = {T100, T400, T800, T900}. The *diffset* between the two is *diffset*({I1, I2}, {I1}) = {T500, T700}. Thus, rather than recording the four TIDs that make up the intersection of {I1} and {I2}, we can instead use *diffset* to record just two TIDs, indicating the difference between {I1} and {I1, I2}. Experiments show that in certain situations, such as when the data set contains many dense and long patterns, this technique can substantially reduce the total cost of vertical format mining of frequent itemsets.

6.2.6 Mining Closed and Max Patterns

In Section 6.1.2 we saw how frequent itemset mining may generate a huge number of frequent itemsets, especially when the *min_sup* threshold is set low or when there exist long patterns in the data set. Example 6.2 showed that closed frequent itemsets[9] can substantially reduce the number of patterns generated in frequent itemset mining while preserving the complete information regarding the set of frequent itemsets. That is, from the set of closed frequent itemsets, we can easily derive the set of frequent itemsets and their support. Thus, in practice, it is more desirable to mine the set of closed frequent itemsets rather than the set of all frequent itemsets in most cases.

"How can we mine closed frequent itemsets?" A naïve approach would be to first mine the complete set of frequent itemsets and then remove every frequent itemset that is a proper subset of, and carries the same support as, an existing frequent itemset. However, this is quite costly. As shown in Example 6.2, this method would have to first derive $2^{100} - 1$ frequent itemsets to obtain a length-100 frequent itemset, all before it could begin to eliminate redundant itemsets. This is prohibitively expensive. In fact, there exist only a very small number of closed frequent itemsets in Example 6.2's data set.

A recommended methodology is to search for closed frequent itemsets directly during the mining process. This requires us to prune the search space as soon as we

[9] Remember that X is a *closed frequent* itemset in a data set S if there exists no proper super-itemset Y such that Y has the same support count as X in S, and X satisfies minimum support.

can identify the case of closed itemsets during mining. Pruning strategies include the following:

Item merging: *If every transaction containing a frequent itemset X also contains an itemset Y but not any proper superset of Y, then* $X \cup Y$ *forms a frequent closed itemset and there is no need to search for any itemset containing X but no Y.*

For example, in Table 6.2 of Example 6.5, the projected conditional database for prefix itemset {I5:2} is {{I2, I1}, {I2, I1, I3}}, from which we can see that each of its transactions contains itemset {I2, I1} but no proper superset of {I2, I1}. Itemset {I2, I1} can be merged with {I5} to form the closed itemset, {I5, I2, I1: 2}, and we do not need to mine for closed itemsets that contain I5 but not {I2, I1}.

Sub-itemset pruning: *If a frequent itemset* X *is a proper subset of an already found frequent closed itemset* Y *and* support_count(X)=support_count(Y)*, then* X *and all of* X*'s descendants in the set enumeration tree cannot be frequent closed itemsets and thus can be pruned.*

Similar to Example 6.2, suppose a transaction database has only two transactions: $\{\langle a_1, a_2, \ldots, a_{100}\rangle, \langle a_1, a_2, \ldots, a_{50}\rangle\}$, and the minimum support count is $min\_sup = 2$. The projection on the first item, a_1, derives the frequent itemset, $\{a_1, a_2, \ldots, a_{50} : 2\}$, based on the *itemset merging* optimization. Because $support(\{a_2\}) = support(\{a_1, a_2, \ldots, a_{50}\}) = 2$, and $\{a_2\}$ is a proper subset of $\{a_1, a_2, \ldots, a_{50}\}$, there is no need to examine a_2 and its projected database. Similar pruning can be done for $a_3, \ldots, a_{50}$ as well. Thus, the mining of closed frequent itemsets in this data set terminates after mining a_1's projected database.

Item skipping: *In the depth-first mining of closed itemsets, at each level, there will be a prefix itemset* X *associated with a header table and a projected database. If a local frequent item* p *has the same support in several header tables at different levels, we can safely prune* p *from the header tables at higher levels.*

Consider, for example, the previous transaction database having only two transactions: $\{\langle a_1, a_2, \ldots, a_{100}\rangle, \langle a_1, a_2, \ldots, a_{50}\rangle\}$, where $min\_sup = 2$. Because a_2 in a_1's projected database has the same support as a_2 in the global header table, a_2 can be pruned from the global header table. Similar pruning can be done for $a_3, \ldots, a_{50}$. There is no need to mine anything more after mining a_1's projected database.

Besides pruning the search space in the closed itemset mining process, another important optimization is to perform efficient checking of each newly derived frequent itemset to see whether it is closed. This is because the mining process cannot ensure that every generated frequent itemset is closed.

When a new frequent itemset is derived, it is necessary to perform two kinds of closure checking: (1) *superset checking*, which checks if this new frequent itemset is a superset of some already found closed itemsets with the same support, and (2) *subset checking*, which checks whether the newly found itemset is a subset of an already found closed itemset with the same support.

If we adopt the *item merging* pruning method under a divide-and-conquer framework, then the superset checking is actually built-in and there is no need to explicitly

perform superset checking. This is because if a frequent itemset $X \cup Y$ is found later than itemset X, and carries the same support as X, it must be in X's projected database and must have been generated during itemset merging.

To assist in subset checking, a compressed **pattern-tree** can be constructed to maintain the set of closed itemsets mined so far. The pattern-tree is similar in structure to the FP-tree except that all the closed itemsets found are stored explicitly in the corresponding tree branches. For efficient subset checking, we can use the following property: *If the current itemset* S_c *can be subsumed by another already found closed itemset* S_a*, then (1)* S_c *and* S_a *have the same support, (2) the length of* S_c *is smaller than that of* S_a*, and (3) all of the items in* S_c *are contained in* S_a.

Based on this property, a **two-level hash index structure** can be built for fast accessing of the pattern-tree: The first level uses the identifier of the last item in S_c as a hash key (since this identifier must be within the branch of S_c), and the second level uses the support of S_c as a hash key (since S_c and S_a have the same support). This will substantially speed up the subset checking process.

This discussion illustrates methods for efficient mining of closed frequent itemsets. *"Can we extend these methods for efficient mining of maximal frequent itemsets?"* Because maximal frequent itemsets share many similarities with closed frequent itemsets, many of the optimization techniques developed here can be extended to mining maximal frequent itemsets. However, we leave this method as an exercise for interested readers.

6.3 Which Patterns Are Interesting?—Pattern Evaluation Methods

Most association rule mining algorithms employ a support–confidence framework. Although minimum support and confidence thresholds *help* weed out or exclude the exploration of a good number of uninteresting rules, many of the rules generated are still not interesting to the users. Unfortunately, this is especially true *when mining at low support thresholds or mining for long patterns.* This has been a major bottleneck for successful application of association rule mining.

In this section, we first look at how even strong association rules can be uninteresting and misleading (Section 6.3.1). We then discuss how the support–confidence framework can be supplemented with additional interestingness measures based on *correlation analysis* (Section 6.3.2). Section 6.3.3 presents additional pattern evaluation measures. It then provides an overall comparison of all the measures discussed here. By the end, you will learn which pattern evaluation measures are most effective for the discovery of only interesting rules.

6.3.1 Strong Rules Are Not Necessarily Interesting

Whether or not a rule is interesting can be assessed either subjectively or objectively. Ultimately, only the user can judge if a given rule is interesting, and this judgment, being

subjective, may differ from one user to another. However, objective interestingness measures, based on the statistics "behind" the data, can be used as one step toward the goal of weeding out uninteresting rules that would otherwise be presented to the user.

"*How can we tell which strong association rules are really interesting?*" Let's examine the following example.

Example 6.7 **A misleading "strong" association rule.** Suppose we are interested in analyzing transactions at *AllElectronics* with respect to the purchase of computer games and videos. Let *game* refer to the transactions containing computer games, and *video* refer to those containing videos. Of the 10,000 transactions analyzed, the data show that 6000 of the customer transactions included computer games, while 7500 included videos, and 4000 included both computer games and videos. Suppose that a data mining program for discovering association rules is run on the data, using a minimum support of, say, 30% and a minimum confidence of 60%. The following association rule is discovered:

$$buys(X, \text{"}computer\ games\text{"}) \Rightarrow buys(X, \text{"}videos\text{"})$$
$$[support = 40\%, confidence = 66\%]. \tag{6.6}$$

Rule (6.6) is a strong association rule and would therefore be reported, since its support value of $\frac{4000}{10{,}000} = 40\%$ and confidence value of $\frac{4000}{6000} = 66\%$ satisfy the minimum support and minimum confidence thresholds, respectively. However, Rule (6.6) is misleading because the probability of purchasing videos is 75%, which is even larger than 66%. In fact, computer games and videos are negatively associated because the purchase of one of these items actually decreases the likelihood of purchasing the other. Without fully understanding this phenomenon, we could easily make unwise business decisions based on Rule (6.6). ■

Example 6.7 also illustrates that the confidence of a rule $A \Rightarrow B$ can be deceiving. It does not measure the *real strength* (or lack of strength) of the *correlation* and *implication* between A and B. Hence, alternatives to the support–confidence framework can be useful in mining interesting data relationships.

6.3.2 From Association Analysis to Correlation Analysis

As we have seen so far, the support and confidence measures are insufficient at filtering out uninteresting association rules. To tackle this weakness, a correlation measure can be used to augment the support–confidence framework for association rules. This leads to *correlation rules* of the form

$$A \Rightarrow B\ [support, confidence, correlation]. \tag{6.7}$$

That is, a correlation rule is measured not only by its support and confidence but also by the correlation between itemsets A and B. There are many different correlation measures from which to choose. In this subsection, we study several correlation measures to determine which would be good for mining large data sets.

Lift is a simple correlation measure that is given as follows. The occurrence of itemset A is **independent** of the occurrence of itemset B if $P(A \cup B) = P(A)P(B)$; otherwise, itemsets A and B are **dependent** and **correlated** as events. This definition can easily be extended to more than two itemsets. The **lift** between the occurrence of A and B can be measured by computing

$$lift(A, B) = \frac{P(A \cup B)}{P(A)P(B)}. \tag{6.8}$$

If the resulting value of Eq. (6.8) is less than 1, then the occurrence of A is *negatively correlated with* the occurrence of B, meaning that the occurrence of one likely leads to the absence of the other one. If the resulting value is greater than 1, then A and B are *positively correlated*, meaning that the occurrence of one implies the occurrence of the other. If the resulting value is equal to 1, then A and B are *independent* and there is no correlation between them.

Equation (6.8) is equivalent to $P(B|A)/P(B)$, or $conf(A \Rightarrow B)/sup(B)$, which is also referred to as the *lift* of the association (or correlation) rule $A \Rightarrow B$. In other words, it assesses the degree to which the occurrence of one "lifts" the occurrence of the other. For example, if A corresponds to the sale of computer games and B corresponds to the sale of videos, then given the current market conditions, the sale of games is said to increase or "lift" the likelihood of the sale of videos by a factor of the value returned by Eq. (6.8).

Let's go back to the computer game and video data of Example 6.7.

Example 6.8 **Correlation analysis using lift.** To help filter out misleading "strong" associations of the form $A \Rightarrow B$ from the data of Example 6.7, we need to study how the two itemsets, A and B, are correlated. Let $\overline{game}$ refer to the transactions of Example 6.7 that do not contain computer games, and $\overline{video}$ refer to those that do not contain videos. The transactions can be summarized in a *contingency table*, as shown in Table 6.6.

From the table, we can see that the probability of purchasing a computer game is $P(\{game\}) = 0.60$, the probability of purchasing a video is $P(\{video\}) = 0.75$, and the probability of purchasing both is $P(\{game, video\}) = 0.40$. By Eq. (6.8), the lift of Rule (6.6) is $P(\{game, video\})/(P(\{game\}) \times P(\{video\})) = 0.40/(0.60 \times 0.75) = 0.89$. Because this value is less than 1, there is a negative correlation between the occurrence of $\{game\}$ and $\{video\}$. The numerator is the likelihood of a customer purchasing both, while the denominator is what the likelihood would have been if the two purchases were completely independent. Such a negative correlation cannot be identified by a support–confidence framework. ■

The second correlation measure that we study is the χ^2 measure, which was introduced in Chapter 3 (Eq. 3.1). To compute the χ^2 value, we take the squared difference between the observed and expected value for a slot (A and B pair) in the contingency table, divided by the expected value. This amount is summed for all slots of the contingency table. Let's perform a χ^2 analysis of Example 6.8.

Table 6.6 2×2 Contingency Table Summarizing the Transactions with Respect to Game and Video Purchases

	game	$\overline{game}$	Σ_{row}
video	4000	3500	7500
$\overline{video}$	2000	500	2500
Σ_{col}	6000	4000	10,000

Table 6.7 Table 6.6 Contingency Table, Now with the Expected Values

	game	$\overline{game}$	Σ_{row}
video	4000 (4500)	3500 (3000)	7500
$\overline{video}$	2000 (1500)	500 (1000)	2500
Σ_{col}	6000	4000	10,000

Example 6.9 **Correlation analysis using χ^2.** To compute the correlation using χ^2 analysis for nominal data, we need the observed value and expected value (displayed in parenthesis) for each slot of the contingency table, as shown in Table 6.7. From the table, we can compute the χ^2 value as follows:

$$\chi^2 = \Sigma \frac{(observed - expected)^2}{expected} = \frac{(4000-4500)^2}{4500} + \frac{(3500-3000)^2}{3000}$$
$$+ \frac{(2000-1500)^2}{1500} + \frac{(500-1000)^2}{1000} = 555.6.$$

Because the χ^2 value is greater than 1, and the observed value of the slot (*game*, *video*) = 4000, which is less than the expected value of 4500, *buying game* and *buying video* are *negatively correlated*. This is consistent with the conclusion derived from the analysis of the *lift* measure in Example 6.8. ■

6.3.3 A Comparison of Pattern Evaluation Measures

The above discussion shows that instead of using the simple support–confidence framework to evaluate frequent patterns, other measures, such as *lift* and χ^2, often disclose more intrinsic pattern relationships. How effective are these measures? Should we also consider other alternatives?

Researchers have studied many pattern evaluation measures even before the start of in-depth research on scalable methods for mining frequent patterns. Recently, several other pattern evaluation measures have attracted interest. In this subsection, we present

four such measures: *all_confidence*, *max_confidence*, *Kulczynski*, and *cosine*. We'll then compare their effectiveness with respect to one another and with respect to the *lift* and χ^2 measures.

Given two itemsets, A and B, the **all_confidence** measure of A and B is defined as

$$all\_conf(A,B) = \frac{sup(A \cup B)}{max\{sup(A), sup(B)\}} = min\{P(A|B), P(B|A)\}, \tag{6.9}$$

where $max\{sup(A), sup(B)\}$ is the maximum support of the itemsets A and B. Thus, $all\_conf(A,B)$ is also the minimum confidence of the two association rules related to A and B, namely, "$A \Rightarrow B$" and "$B \Rightarrow A$."

Given two itemsets, A and B, the **max_confidence** measure of A and B is defined as

$$max\_conf(A, B) = max\{P(A \mid B), P(B \mid A)\}. \tag{6.10}$$

The *max_conf* measure is the maximum confidence of the two association rules, "$A \Rightarrow B$" and "$B \Rightarrow A$."

Given two itemsets, A and B, the **Kulczynski** measure of A and B (abbreviated as **Kulc**) is defined as

$$Kulc(A, B) = \frac{1}{2}(P(A|B) + P(B|A)). \tag{6.11}$$

It was proposed in 1927 by Polish mathematician S. Kulczynski. It can be viewed as an average of two confidence measures. That is, it is the average of two conditional probabilities: the probability of itemset B given itemset A, and the probability of itemset A given itemset B.

Finally, given two itemsets, A and B, the **cosine** measure of A and B is defined as

$$\begin{aligned} cosine(A, B) &= \frac{P(A \cup B)}{\sqrt{P(A) \times P(B)}} = \frac{sup(A \cup B)}{\sqrt{sup(A) \times sup(B)}} \\ &= \sqrt{P(A|B) \times P(B|A)}. \end{aligned} \tag{6.12}$$

The *cosine* measure can be viewed as a *harmonized lift* measure: The two formulae are similar except that for cosine, the *square root* is taken on the product of the probabilities of A and B. This is an important difference, however, because by taking the square root, the cosine value is only influenced by the supports of A, B, and $A \cup B$, and not by the total number of transactions.

Each of these four measures defined has the following property: Its value is only influenced by the supports of A, B, and $A \cup B$, or more exactly, by the conditional probabilities of $P(A|B)$ and $P(B|A)$, but not by the total number of transactions. Another common property is that each measure ranges from 0 to 1, and the higher the value, the closer the relationship between A and B.

Now, together with *lift* and χ^2, we have introduced in total six pattern evaluation measures. You may wonder, "*Which is the best in assessing the discovered pattern relationships?*" To answer this question, we examine their performance on some typical data sets.

Table 6.8 2×2 Contingency Table for Two Items

	milk	$\overline{milk}$	Σ_{row}
coffee	mc	$\overline{m}c$	c
$\overline{coffee}$	$m\overline{c}$	$\overline{m}\overline{c}$	$\overline{c}$
Σ_{col}	m	$\overline{m}$	Σ

Table 6.9 Comparison of Six Pattern Evaluation Measures Using Contingency Tables for a Variety of Data Sets

Data Set	*mc*	$\overline{m}c$	$m\overline{c}$	$\overline{m}\overline{c}$	χ^2	*lift*	*all_conf.*	*max_conf.*	*Kulc.*	*cosine*
D_1	10,000	1000	1000	100,000	90557	9.26	0.91	0.91	0.91	0.91
D_2	10,000	1000	1000	100	0	1	0.91	0.91	0.91	0.91
D_3	100	1000	1000	100,000	670	8.44	0.09	0.09	0.09	0.09
D_4	1000	1000	1000	100,000	24740	25.75	0.5	0.5	0.5	0.5
D_5	1000	100	10,000	100,000	8173	9.18	0.09	0.91	0.5	0.29
D_6	1000	10	100,000	100,000	965	1.97	0.01	0.99	0.5	0.10

Example 6.10 **Comparison of six pattern evaluation measures on typical data sets.** The relationships between the purchases of two items, *milk* and *coffee*, can be examined by summarizing their purchase history in Table 6.8, a 2×2 contingency table, where an entry such as mc represents the number of transactions containing both milk and coffee.

Table 6.9 shows a set of transactional data sets with their corresponding contingency tables and the associated values for each of the six evaluation measures. Let's first examine the first four data sets, D_1 through D_4. From the table, we see that m and c are positively associated in D_1 and D_2, negatively associated in D_3, and neutral in D_4. For D_1 and D_2, m and c are positively associated because mc (10,000) is considerably greater than $\overline{m}c$ (1000) and $m\overline{c}$ (1000). Intuitively, for people who bought milk ($m = 10,000 + 1000 = 11,000$), it is very likely that they also bought coffee ($mc/m = 10/11 = 91\%$), and vice versa.

The results of the four newly introduced measures show that m and c are strongly positively associated in both data sets by producing a measure value of 0.91. However, *lift* and χ^2 generate dramatically different measure values for D_1 and D_2 due to their sensitivity to $\overline{m}\overline{c}$. In fact, in many real-world scenarios, $\overline{m}\overline{c}$ is usually huge and unstable. For example, in a market basket database, the total number of transactions could fluctuate on a daily basis and overwhelmingly exceed the number of transactions containing any particular itemset. Therefore, a good interestingness measure should not be affected by transactions that do not contain the itemsets of interest; otherwise, it would generate unstable results, as illustrated in D_1 and D_2.

Similarly, in D_3, the four new measures correctly show that m and c are strongly negatively associated because the m to c ratio equals the mc to m ratio, that is, $100/1100 = 9.1\%$. However, *lift* and χ^2 both contradict this in an incorrect way: Their values for D_2 are between those for D_1 and D_3.

For data set D_4, both *lift* and χ^2 indicate a highly positive association between m and c, whereas the others indicate a "neutral" association because the ratio of mc to $\overline{m}c$ equals the ratio of mc to $m\overline{c}$, which is 1. This means that if a customer buys coffee (or milk), the probability that he or she will also purchase milk (or coffee) is exactly 50%. ■

"*Why are* lift *and* χ^2 *so poor at distinguishing pattern association relationships in the previous transactional data sets?*" To answer this, we have to consider the *null-transactions*. A **null-transaction** is a transaction that does not contain any of the itemsets being examined. In our example, $\overline{mc}$ represents the number of null-transactions. *Lift* and χ^2 have difficulty distinguishing interesting pattern association relationships because they are both strongly influenced by $\overline{mc}$. Typically, the number of null-transactions can outweigh the number of individual purchases because, for example, many people may buy neither milk nor coffee. On the other hand, the other four measures are good indicators of interesting pattern associations because their definitions remove the influence of $\overline{mc}$ (i.e., they are not influenced by the number of null-transactions).

This discussion shows that it is highly desirable to have a measure that has a value that is independent of the number of null-transactions. A measure is **null-invariant** if its value is free from the influence of null-transactions. Null-invariance is an important property for measuring association patterns in large transaction databases. Among the six discussed measures in this subsection, only *lift* and χ^2 are not null-invariant measures.

"*Among the* all_confidence, max_confidence, Kulczynski, *and* cosine *measures, which is best at indicating interesting pattern relationships*?"

To answer this question, we introduce the **imbalance ratio** (**IR**), which assesses the imbalance of two itemsets, A and B, in rule implications. It is defined as

$$IR(A,B) = \frac{|sup(A) - sup(B)|}{sup(A) + sup(B) - sup(A \cup B)}, \tag{6.13}$$

where the numerator is the absolute value of the difference between the support of the itemsets A and B, and the denominator is the number of transactions containing A or B. If the two directional implications between A and B are the same, then $IR(A,B)$ will be zero. Otherwise, the larger the difference between the two, the larger the imbalance ratio. This ratio is independent of the number of null-transactions and independent of the total number of transactions.

Let's continue examining the remaining data sets in Example 6.10.

Example 6.11 Comparing null-invariant measures in pattern evaluation. Although the four measures introduced in this section are null-invariant, they may present dramatically

different values on some subtly different data sets. Let's examine data sets D_5 and D_6, shown earlier in Table 6.9, where the two events m and c have unbalanced conditional probabilities. That is, the ratio of mc to c is greater than 0.9. This means that knowing that c occurs should strongly suggest that m occurs also. The ratio of mc to m is less than 0.1, indicating that m implies that c is quite unlikely to occur. The *all_confidence* and *cosine* measures view both cases as negatively associated and the *Kulc* measure views both as neutral. The *max_confidence* measure claims strong positive associations for these cases. The measures give very diverse results!

"*Which measure intuitively reflects the true relationship between the purchase of milk and coffee?*" Due to the "*balanced*" skewness of the data, it is difficult to argue whether the two data sets have positive or negative association. From one point of view, only $mc/(mc + m\overline{c}) = 1000/(1000 + 10,000) = 9.09\%$ of milk-related transactions contain coffee in D_5 and this percentage is $1000/(1000 + 100,000) = 0.99\%$ in D_6, both indicating a negative association. On the other hand, 90.9% of transactions in D_5 (i.e., $mc/(mc + \overline{m}c) = 1000/(1000 + 100)$) and 9% in D_6 (i.e., $1000/(1000 + 10)$) containing coffee contain milk as well, which indicates a positive association between milk and coffee. These draw very different conclusions.

For such "balanced" skewness, it could be fair to treat it as neutral, as *Kulc* does, and in the meantime indicate its skewness using the *imbalance ratio (IR)*. According to Eq. (6.13), for D_4 we have $IR(m, c) = 0$, a perfectly balanced case; for D_5, $IR(m, c) = 0.89$, a rather imbalanced case; whereas for D_6, $IR(m, c) = 0.99$, a very skewed case. Therefore, the two measures, *Kulc* and *IR*, work together, presenting a clear picture for all three data sets, D_4 through D_6. ■

In summary, the use of only support and confidence measures to mine associations may generate a large number of rules, many of which can be uninteresting to users. Instead, we can augment the support–confidence framework with a pattern interestingness measure, which helps focus the mining toward rules with strong pattern relationships. The added measure substantially reduces the number of rules generated and leads to the discovery of more meaningful rules. Besides those introduced in this section, many other interestingness measures have been studied in the literature. Unfortunately, most of them do not have the null-invariance property. Because large data sets typically have many null-transactions, it is important to consider the null-invariance property when selecting appropriate interestingness measures for pattern evaluation. Among the four null-invariant measures studied here, namely *all_confidence*, *max_confidence*, *Kulc*, and *cosine*, we recommend using *Kulc* in conjunction with the imbalance ratio.

6.4 Summary

- The discovery of frequent patterns, associations, and correlation relationships among huge amounts of data is useful in selective marketing, decision analysis, and business management. A popular area of application is **market basket analysis**, which studies

customers' buying habits by searching for itemsets that are frequently purchased together (or in sequence).

- **Association rule mining** consists of first finding **frequent itemsets** (sets of items, such as A and B, satisfying a *minimum support threshold*, or percentage of the task-relevant tuples), from which **strong** association rules in the form of $A \Rightarrow B$ are generated. These rules also satisfy a *minimum confidence threshold* (a prespecified probability of satisfying B under the condition that A is satisfied). Associations can be further analyzed to uncover **correlation rules**, which convey statistical correlations between itemsets A and B.
- Many efficient and scalable algorithms have been developed for **frequent itemset mining**, from which association and correlation rules can be derived. These algorithms can be classified into three categories: (1) *Apriori-like algorithms*, (2) *frequent pattern growth–based algorithms* such as FP-growth, and (3) *algorithms that use the vertical data format*.
- The **Apriori algorithm** is a seminal algorithm for mining frequent itemsets for Boolean association rules. It explores the level-wise mining Apriori property that *all nonempty subsets of a frequent itemset must also be frequent*. At the kth iteration (for $k \geq 2$), it forms frequent k-itemset candidates based on the frequent $(k-1)$-itemsets, and scans the database once to find the *complete* set of frequent k-itemsets, L_k.

  Variations involving hashing and transaction reduction can be used to make the procedure more efficient. Other variations include partitioning the data (mining on each partition and then combining the results) and sampling the data (mining on a data subset). These variations can reduce the number of data scans required to as little as two or even one.
- **Frequent pattern growth** is a method of mining frequent itemsets without candidate generation. It constructs a highly compact data structure (an *FP-tree*) to compress the original transaction database. Rather than employing the generate-and-test strategy of Apriori-like methods, it focuses on frequent pattern (fragment) growth, which avoids costly candidate generation, resulting in greater efficiency.
- **Mining frequent itemsets using the vertical data format (Eclat)** is a method that transforms a given data set of transactions in the horizontal data format of *TID-itemset* into the vertical data format of *item-TID_set*. It mines the transformed data set by *TID_set* intersections based on the Apriori property and additional optimization techniques such as *diffset*.
- Not all strong association rules are interesting. Therefore, the support–confidence framework should be augmented with a pattern evaluation measure, which promotes the mining of *interesting* rules. A measure is **null-invariant** if its value is free from the influence of **null-transactions** (i.e., the *transactions that do not contain any of the itemsets being examined*). Among many pattern evaluation measures, we examined ***lift***, χ^2***, all_confidence, max_confidence, Kulczynski***, and ***cosine***, and showed

that only the latter four are null-invariant. We suggest using the Kulczynski measure, together with the imbalance ratio, to present pattern relationships among itemsets.

6.5 Exercises

6.1 Suppose you have the set $\mathcal{C}$ of all frequent closed itemsets on a data set D, as well as the support count for each frequent closed itemset. Describe an algorithm to determine whether a given itemset X is frequent or not, and the support of X if it is frequent.

6.2 An itemset X is called a *generator* on a data set D if there does not exist a proper sub-itemset $Y \subset X$ such that $support(X) = support(Y)$. A generator X is a *frequent generator* if $support(X)$ passes the minimum support threshold. Let $\mathcal{G}$ be the set of all frequent generators on a data set D.

(a) Can you determine whether an itemset A is frequent and the support of A, if it is frequent, using only $\mathcal{G}$ and the support counts of all frequent generators? If yes, present your algorithm. Otherwise, what other information is needed? Can you give an algorithm assuming the information needed is available?

(b) What is the relationship between closed itemsets and generators?

6.3 The Apriori algorithm makes use of *prior knowledge* of subset support properties.

(a) Prove that all nonempty subsets of a frequent itemset must also be frequent.

(b) Prove that the support of any nonempty subset s' of itemset s must be at least as great as the support of s.

(c) Given frequent itemset l and subset s of l, prove that the confidence of the rule "$s' \Rightarrow (l - s')$" cannot be more than the confidence of "$s \Rightarrow (l - s)$," where s' is a subset of s.

(d) A *partitioning* variation of Apriori subdivides the transactions of a database D into n nonoverlapping partitions. Prove that any itemset that is frequent in D must be frequent in at least one partition of D.

6.4 Let c be a candidate itemset in C_k generated by the Apriori algorithm. How many length-$(k-1)$ subsets do we need to check in the prune step? Per your previous answer, can you give an improved version of procedure has_infrequent_subset in Figure 6.4?

6.5 Section 6.2.2 describes a method for *generating association rules* from frequent itemsets. Propose a more efficient method. Explain why it is more efficient than the one proposed there. (*Hint:* Consider incorporating the properties of Exercises 6.3(b), (c) into your design.)

6.6 A database has five transactions. Let $min\_sup = 60\%$ and $min\_conf = 80\%$.

TID	*items_bought*
T100	{M, O, N, K, E, Y}
T200	{D, O, N, K, E, Y }
T300	{M, A, K, E}
T400	{M, U, C, K, Y}
T500	{C, O, O, K, I, E}

(a) Find all frequent itemsets using Apriori and FP-growth, respectively. Compare the efficiency of the two mining processes.

(b) List all the *strong* association rules (with support s and confidence c) matching the following metarule, where X is a variable representing customers, and $item_i$ denotes variables representing items (e.g., "*A*," "*B*,"):

$$\forall x \in transaction,\ buys(X, item_1) \wedge buys(X, item_2) \Rightarrow buys(X, item_3) \quad [s, c]$$

6.7 (**Implementation project**) Using a programming language that you are familiar with, such as C++ or Java, implement three *frequent itemset mining* algorithms introduced in this chapter: (1) Apriori [AS94b], (2) FP-growth [HPY00], and (3) Eclat [Zak00] (mining using the vertical data format). Compare the performance of each algorithm with various kinds of large data sets. Write a report to analyze the situations (e.g., data size, data distribution, minimal support threshold setting, and pattern density) where one algorithm may perform better than the others, and state why.

6.8 A database has four transactions. Let $min\_sup = 60\%$ and $min\_conf = 80\%$.

cust_ID	*TID*	*items_bought* (in the form of *brand-item_category*)
01	T100	{King's-Crab, Sunset-Milk, Dairyland-Cheese, Best-Bread}
02	T200	{Best-Cheese, Dairyland-Milk, Goldenfarm-Apple, Tasty-Pie, Wonder-Bread}
01	T300	{Westcoast-Apple, Dairyland-Milk, Wonder-Bread, Tasty-Pie}
03	T400	{Wonder-Bread, Sunset-Milk, Dairyland-Cheese}

(a) At the granularity of *item_category* (e.g., $item_i$ could be "*Milk*"), for the rule template,

$$\forall X \in transaction,\ buys(X, item_1) \wedge buys(X, item_2) \Rightarrow buys(X, item_3) \quad [s, c],$$

list the frequent k-itemset for the largest k, and *all* the *strong* association rules (with their support s and confidence c) containing the frequent k-itemset for the largest k.

(b) At the granularity of *brand-item_category* (e.g., $item_i$ could be "*Sunset-Milk*"), for the rule template,

$$\forall X \in customer,\ buys(X, item_1) \wedge buys(X, item_2) \Rightarrow buys(X, item_3),$$

list the frequent k-itemset for the largest k (but do not print any rules).

6.9 Suppose that a large store has a transactional database that is *distributed* among four locations. Transactions in each component database have the same format, namely $T_j : \{i_1, \ldots, i_m\}$, where T_j is a transaction identifier, and i_k $(1 \leq k \leq m)$ is the identifier of an item purchased in the transaction. Propose an efficient algorithm to mine global association rules. You may present your algorithm in the form of an outline. Your algorithm should not require shipping all the data to one site and should not cause excessive network communication overhead.

6.10 Suppose that frequent itemsets are saved for a large transactional database, *DB*. Discuss how to efficiently mine the (global) association rules under the same minimum support threshold, if a set of new transactions, denoted as ΔDB, is *(incrementally) added in*?

6.11 Most frequent pattern mining algorithms consider only distinct items in a transaction. However, multiple occurrences of an item in the same shopping basket, such as four cakes and three jugs of milk, can be important in transactional data analysis. How can one mine frequent itemsets efficiently considering multiple occurrences of items? Propose modifications to the well-known algorithms, such as Apriori and FP-growth, to adapt to such a situation.

6.12 (**Implementation project**) Many techniques have been proposed to further improve the performance of frequent itemset mining algorithms. Taking FP-tree–based frequent pattern growth algorithms (e.g., FP-growth) as an example, implement one of the following optimization techniques. Compare the performance of your new implementation with the unoptimized version.

(a) The frequent pattern mining method of Section 6.2.4 uses an FP-tree to generate conditional pattern bases using a bottom-up projection technique (i.e., project onto the prefix path of an item p). However, one can develop a *top-down projection* technique, that is, project onto the suffix path of an item p in the generation of a conditional pattern base. Design and implement such a top-down FP-tree mining method. Compare its performance with the bottom-up projection method.

(b) Nodes and pointers are used uniformly in an FP-tree in the FP-growth algorithm design. However, such a structure may consume a lot of space when the data are sparse. One possible alternative design is to explore *array- and pointer-based hybrid implementation*, where a node may store multiple items when it contains no splitting point to multiple sub-branches. Develop such an implementation and compare it with the original one.

(c) It is time and space consuming to generate numerous conditional pattern bases during pattern-growth mining. An interesting alternative is to *push right* the branches that have been mined for a particular item p, that is, to push them to the remaining branch(es) of the FP-tree. This is done so that fewer conditional pattern bases have to be generated and additional sharing can be explored when mining the remaining FP-tree branches. Design and implement such a method and conduct a performance study on it.

6.13 Give a short example to show that items in a strong association rule actually may be *negatively correlated.*

6.14 The following contingency table summarizes supermarket transaction data, where *hot dogs* refers to the transactions containing hot dogs, $\overline{hot\ dogs}$ refers to the transactions that do not contain hot dogs, *hamburgers* refers to the transactions containing hamburgers, and $\overline{hamburgers}$ refers to the transactions that do not contain hamburgers.

	hot dogs	$\overline{hot\ dogs}$	Σ_{row}
hamburgers	2000	500	2500
$\overline{hamburgers}$	1000	1500	2500
Σ_{col}	3000	2000	5000

(a) Suppose that the association rule "*hot dogs* $\Rightarrow$ *hamburgers*" is mined. Given a minimum support threshold of 25% and a minimum confidence threshold of 50%, is this association rule strong?

(b) Based on the given data, is the purchase of *hot dogs* independent of the purchase of *hamburgers*? If not, what kind of *correlation* relationship exists between the two?

(c) Compare the use of the *all_confidence*, *max_confidence*, *Kulczynski*, and *cosine* measures with *lift* and *correlation* on the given data.

6.15 (**Implementation project**) The DBLP data set *(www.informatik.uni-trier .de/~ley/db/)* consists of over one million entries of research papers published in computer science conferences and journals. Among these entries, there are a good number of authors that have coauthor relationships.

(a) Propose a method to efficiently mine a set of coauthor relationships that are closely correlated (e.g., often coauthoring papers together).

(b) Based on the mining results and the pattern evaluation measures discussed in this chapter, discuss which measure may convincingly uncover close collaboration patterns better than others.

(c) Based on the study in (a), develop a method that can roughly predict advisor and advisee relationships and the approximate period for such advisory supervision.

6.6 Bibliographic Notes

Association rule mining was first proposed by Agrawal, Imielinski, and Swami [AIS93]. The Apriori algorithm discussed in Section 6.2.1 for frequent itemset mining was presented in Agrawal and Srikant [AS94b]. A variation of the algorithm using a similar pruning heuristic was developed independently by Mannila, Tiovonen, and Verkamo

[MTV94]. A joint publication combining these works later appeared in Agrawal, Mannila, Srikant et al. [AMS+96]. A method for generating association rules from frequent itemsets is described in Agrawal and Srikant [AS94a].

References for the variations of Apriori described in Section 6.2.3 include the following. The use of hash tables to improve association mining efficiency was studied by Park, Chen, and Yu [PCY95a]. The partitioning technique was proposed by Savasere, Omiecinski, and Navathe [SON95]. The sampling approach is discussed in Toivonen [Toi96]. A dynamic itemset counting approach is given in Brin, Motwani, Ullman, and Tsur [BMUT97]. An efficient incremental updating of mined association rules was proposed by Cheung, Han, Ng, and Wong [CHNW96]. Parallel and distributed association data mining under the Apriori framework was studied by Park, Chen, and Yu [PCY95b]; Agrawal and Shafer [AS96]; and Cheung, Han, Ng, et al. [CHN+96]. Another parallel association mining method, which explores itemset clustering using a vertical database layout, was proposed in Zaki, Parthasarathy, Ogihara, and Li [ZPOL97].

Other scalable frequent itemset mining methods have been proposed as alternatives to the Apriori-based approach. FP-growth, a pattern-growth approach for mining frequent itemsets without candidate generation, was proposed by Han, Pei, and Yin [HPY00] (Section 6.2.4). An exploration of hyper structure mining of frequent patterns, called H-Mine, was proposed by Pei, Han, Lu, et al. [PHL+01]. A method that integrates top-down and bottom-up traversal of FP-trees in pattern-growth mining was proposed by Liu, Pan, Wang, and Han [LPWH02]. An array-based implementation of prefix-tree structure for efficient pattern growth mining was proposed by Grahne and Zhu [GZ03b]. Eclat, an approach for mining frequent itemsets by exploring the vertical data format, was proposed by Zaki [Zak00]. A depth-first generation of frequent itemsets by a tree projection technique was proposed by Agarwal, Aggarwal, and Prasad [AAP01]. An integration of association mining with relational database systems was studied by Sarawagi, Thomas, and Agrawal [STA98].

The mining of frequent closed itemsets was proposed in Pasquier, Bastide, Taouil, and Lakhal [PBTL99], where an Apriori-based algorithm called A-Close for such mining was presented. CLOSET, an efficient closed itemset mining algorithm based on the frequent pattern growth method, was proposed by Pei, Han, and Mao [PHM00]. CHARM by Zaki and Hsiao [ZH02] developed a compact vertical TID list structure called *diffset*, which records only the difference in the TID list of a candidate pattern from its prefix pattern. A fast hash-based approach is also used in CHARM to prune nonclosed patterns. CLOSET+ by Wang, Han, and Pei [WHP03] integrates previously proposed effective strategies as well as newly developed techniques such as hybrid tree-projection and item skipping. AFOPT, a method that explores a *right push* operation on FP-trees during the mining process, was proposed by Liu, Lu, Lou, and Yu [LLLY03]. Grahne and Zhu [GZ03b] proposed a prefix-tree–based algorithm integrated with array representation, called FPClose, for mining closed itemsets using a pattern-growth approach.

Pan, Cong, Tung, et al. [PCT+03] proposed CARPENTER, a method for finding closed patterns in long biological data sets, which integrates the advantages of vertical

data formats and pattern growth methods. Mining max-patterns was first studied by Bayardo [Bay98], where MaxMiner, an Apriori-based, level-wise, breadth-first search method, was proposed to find *max-itemset* by performing *superset frequency pruning* and *subset infrequency pruning* for search space reduction. Another efficient method, MAFIA, developed by Burdick, Calimlim, and Gehrke [BCG01], uses vertical bitmaps to compress TID lists, thus improving the counting efficiency. A FIMI (Frequent Itemset Mining Implementation) workshop dedicated to implementation methods for frequent itemset mining was reported by Goethals and Zaki [GZ03a].

The problem of mining interesting rules has been studied by many researchers. The statistical independence of rules in data mining was studied by Piatetski-Shapiro [P-S91]. The interestingness problem of strong association rules is discussed in Chen, Han, and Yu [CHY96]; Brin, Motwani, and Silverstein [BMS97]; and Aggarwal and Yu [AY99], which cover several interestingness measures, including *lift*. An efficient method for generalizing associations to correlations is given in Brin, Motwani, and Silverstein [BMS97]. Other alternatives to the support–confidence framework for assessing the interestingness of association rules are proposed in Brin, Motwani, Ullman, and Tsur [BMUT97] and Ahmed, El-Makky, and Taha [AEMT00].

A method for mining strong gradient relationships among itemsets was proposed by Imielinski, Khachiyan, and Abdulghani [IKA02]. Silverstein, Brin, Motwani, and Ullman [SBMU98] studied the problem of mining causal structures over transaction databases. Some comparative studies of different interestingness measures were done by Hilderman and Hamilton [HH01]. The notion of null transaction invariance was introduced, together with a comparative analysis of interestingness measures, by Tan, Kumar, and Srivastava [TKS02]. The use of *all_confidence* as a correlation measure for generating interesting association rules was studied by Omiecinski [Omi03] and by Lee, Kim, Cai, and Han [LKCH03]. Wu, Chen, and Han [WCH10] introduced the Kulczynski measure for associative patterns and performed a comparative analysis of a set of measures for pattern evaluation.

7 Advanced Pattern Mining

Frequent pattern mining has reached far beyond the basics due to substantial research, numerous extensions of the problem scope, and broad application studies. In this chapter, you will learn methods for advanced pattern mining. We begin by laying out a general road map for pattern mining. We introduce methods for mining various kinds of patterns, and discuss extended applications of pattern mining. We include in-depth coverage of methods for mining many kinds of patterns: multilevel patterns, multidimensional patterns, patterns in continuous data, rare patterns, negative patterns, constrained frequent patterns, frequent patterns in high-dimensional data, colossal patterns, and compressed and approximate patterns. Other pattern mining themes, including mining sequential and structured patterns and mining patterns from spatiotemporal, multimedia, and stream data, are considered more advanced topics and are not covered in this book. Notice that *pattern mining* is a more general term than *frequent pattern mining* since the former covers rare and negative patterns as well. However, when there is no ambiguity, the two terms are used interchangeably.

7.1 Pattern Mining: A Road Map

Chapter 6 introduced the basic concepts, techniques, and applications of frequent pattern mining using market basket analysis as an example. Many other kinds of data, user requests, and applications have led to the development of numerous, diverse methods for mining patterns, associations, and correlation relationships. Given the rich literature in this area, it is important to lay out a clear road map to help us get an organized picture of the field and to select the best methods for pattern mining applications.

Figure 7.1 outlines a general road map on pattern mining research. Most studies mainly address three pattern mining aspects: the kinds of patterns mined, mining methodologies, and applications. Some studies, however, integrate multiple aspects; for example, different applications may need to mine different patterns, which naturally leads to the development of new mining methodologies.

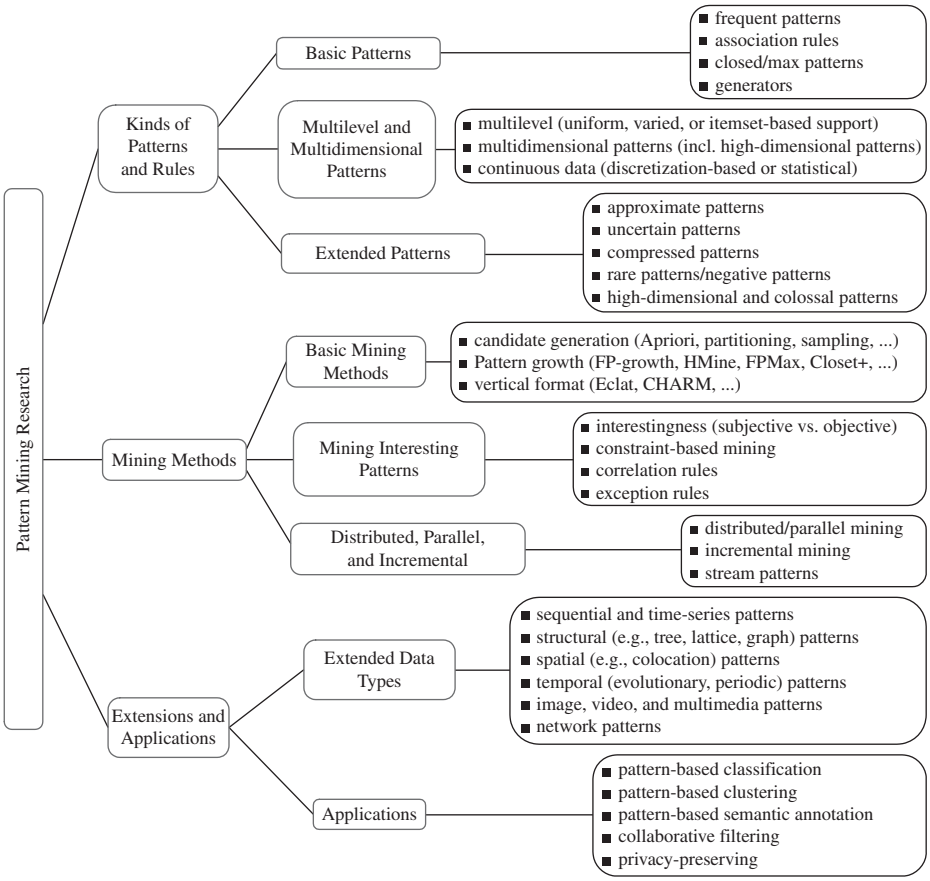


Figure 7.1 A general road map on pattern mining research.

Based on pattern diversity, pattern mining can be classified using the following criteria:

- **Basic patterns:** As discussed in Chapter 6, a frequent pattern may have several alternative forms, including a simple frequent pattern, a closed pattern, or a max-pattern. To review, a **frequent pattern** is a pattern (or itemset) that satisfies a minimum support threshold. A pattern p is a **closed pattern** if there is no superpattern p' with the same support as p. Pattern p is a **max-pattern** if there exists no frequent superpattern of p. Frequent patterns can also be mapped into **association rules**, or other kinds of rules based on interestingness measures. Sometimes we may also be interested in **infrequent** or **rare patterns** (i.e., patterns that occur rarely but are of critical importance, or **negative patterns** (i.e., patterns that reveal a negative correlation between items).

- **Based on the *abstraction* levels involved in a pattern:** Patterns or association rules may have items or concepts residing at high, low, or multiple abstraction levels. For example, suppose that a set of association rules mined includes the following rules where X is a variable representing a customer:

$$buys(X, \text{“}computer\text{”}) \Rightarrow buys(X, \text{“}printer\text{”}) \tag{7.1}$$

$$buys(X, \text{“}laptop\ computer\text{”}) \Rightarrow buys(X, \text{“}color\ laser\ printer\text{”}) \tag{7.2}$$

  In Rules (7.1) and (7.2), the items bought are referenced at different abstraction levels (e.g., “*computer*” is a higher-level abstraction of “*laptop computer*,” and “*color laser printer*” is a lower-level abstraction of “*printer*”). We refer to the rule set mined as consisting of **multilevel association rules**. If, instead, the rules within a given set do not reference items or attributes at different abstraction levels, then the set contains **single-level association rules**.

- **Based on the *number of dimensions* involved in the rule or pattern:** If the items or attributes in an association rule or pattern reference only one dimension, it is a **single-dimensional association rule/pattern**. For example, Rules (7.1) and (7.2) are single-dimensional association rules because they each refer to only one dimension, *buys*.[1]

  If a rule/pattern references two or more dimensions, such as *age, income*, and *buys*, then it is a **multidimensional association rule/pattern**. The following is an example of a multidimensional rule:

$$age(X, \text{“}20\ldots29\text{”}) \wedge income(X, \text{“}52K\ldots58K\text{”}) \Rightarrow buys(X, \text{“}iPad\text{”}). \tag{7.3}$$

- **Based on the *types of values* handled in the rule or pattern:** If a rule involves associations between the presence or absence of items, it is a **Boolean association rule**. For example, Rules (7.1) and (7.2) are Boolean association rules obtained from market basket analysis.

  If a rule describes associations between quantitative items or attributes, then it is a **quantitative association rule**. In these rules, quantitative values for items or attributes are partitioned into intervals. Rule (7.3) can also be considered a quantitative association rule where the quantitative attributes *age* and *income* have been discretized.

- **Based on the *constraints* or *criteria* used to mine *selective patterns*:** The patterns or rules to be discovered can be **constraint-based** (i.e., satisfying a set of user-defined constraints), **approximate**, **compressed**, **near-match** (i.e., those that tally the support count of the near or almost matching itemsets), **top-*k*** (i.e., the k most frequent itemsets for a user-specified value, k), **redundancy-aware top-*k*** (i.e., the top-k patterns with similar or redundant patterns excluded), and so on.

[1] Following the terminology used in multidimensional databases, we refer to each distinct predicate in a rule as a *dimension*.

Alternatively, pattern mining can be classified with respect to the kinds of data and applications involved, using the following criteria:

- **Based on *kinds of data and features* to be mined:** Given relational and data warehouse data, most people are interested in itemsets. Thus, frequent pattern mining in this context is essentially **frequent itemset mining**, that is, to mine frequent *sets of items*. However, in many other applications, patterns may involve sequences and structures. For example, by studying the order in which items are frequently purchased, we may find that customers tend to first buy a PC, followed by a digital camera, and then a memory card. This leads to **sequential patterns**, that is, frequent *subsequences* (which are often separated by some other events) in a *sequence of ordered events*.

  We may also mine **structural patterns**, that is, frequent *substructures*, in a *structured data set*. Note that *structure* is a general concept that covers many different kinds of structural forms such as directed graphs, undirected graphs, lattices, trees, sequences, sets, single items, or combinations of such structures. Single items are the simplest form of structure. Each element of a general pattern may contain a subsequence, a subtree, a subgraph, and so on, and such containment relationships can be defined recursively. Therefore, structural pattern mining can be considered as the most general form of frequent pattern mining.

- **Based on *application domain-specific semantics*:** Both data and applications can be very diverse, and therefore the patterns to be mined can differ largely based on their domain-specific semantics. Various kinds of application data include spatial data, temporal data, spatiotemporal data, multimedia data (e.g., image, audio, and video data), text data, time-series data, DNA and biological sequences, software programs, chemical compound structures, web structures, sensor networks, social and information networks, biological networks, data streams, and so on. This diversity can lead to dramatically different pattern mining methodologies.

- **Based on *data analysis usages*:** Frequent pattern mining often serves as an intermediate step for improved data understanding and more powerful data analysis. For example, it can be used as a feature extraction step for classification, which is often referred to as **pattern-based classification**. Similarly, **pattern-based clustering** has shown its strength at clustering high-dimensional data. For improved data understanding, patterns can be used for semantic annotation or contextual analysis. Pattern analysis can also be used in **recommender systems**, which recommend information items (e.g., books, movies, web pages) that are likely to be of interest to the user based on similar users' patterns. Different analysis tasks may require mining rather different kinds of patterns as well.

The next several sections present advanced methods and extensions of pattern mining, as well as their application. Section 7.2 discusses methods for mining multilevel patterns, multidimensional patterns, patterns and rules with continuous attributes, rare patterns, and negative patterns. Constraint-based pattern mining is studied in

Section 7.3. Section 7.4 explains how to mine high-dimensional and colossal patterns. The mining of compressed and approximate patterns is detailed in Section 7.5. Section 7.6 discusses the exploration and applications of pattern mining. More advanced topics regarding mining sequential and structural patterns, and pattern mining in complex and diverse kinds of data are briefly introduced in Chapter 13.

7.2 Pattern Mining in Multilevel, Multidimensional Space

This section focuses on methods for mining in multilevel, multidimensional space. In particular, you will learn about mining multilevel associations (Section 7.2.1), multidimensional associations (Section 7.2.2), quantitative association rules (Section 7.2.3), and rare patterns and negative patterns (Section 7.2.4). *Multilevel associations* involve concepts at different abstraction levels. *Multidimensional associations* involve more than one dimension or predicate (e.g., rules that relate what a customer *buys* to his or her *age*). *Quantitative association rules* involve numeric attributes that have an implicit ordering among values (e.g., *age*). *Rare patterns* are patterns that suggest interesting although rare item combinations. *Negative patterns* show negative correlations between items.

7.2.1 Mining Multilevel Associations

For many applications, strong associations discovered at high abstraction levels, though with high support, could be commonsense knowledge. We may want to drill down to find novel patterns at more detailed levels. On the other hand, there could be too many scattered patterns at low or primitive abstraction levels, some of which are just trivial specializations of patterns at higher levels. Therefore, it is interesting to examine how to develop effective methods for mining patterns at multiple abstraction levels, with sufficient flexibility for easy traversal among different abstraction spaces.

Example 7.1 **Mining multilevel association rules.** Suppose we are given the task-relevant set of transactional data in Table 7.1 for sales in an *AllElectronics* store, showing the items purchased for each transaction. The concept hierarchy for the items is shown in Figure 7.2. A concept hierarchy defines a sequence of mappings from a set of low-level concepts to a higher-level, more general concept set. Data can be generalized by replacing low-level concepts within the data by their corresponding higher-level concepts, or *ancestors*, from a concept hierarchy.

Figure 7.2's concept hierarchy has five levels, respectively referred to as levels 0 through 4, starting with level 0 at the root node for all (the most general abstraction level). Here, level 1 includes *computer, software, printer and camera,* and *computer accessory*; level 2 includes *laptop computer, desktop computer, office software, antivirus software,* etc.; and level 3 includes *Dell desktop computer, . . . , Microsoft office software,* etc. Level 4 is the most specific abstraction level of this hierarchy. It consists of the raw data values.

Table 7.1 Task-Relevant Data, *D*

TID	*Items Purchased*
T100	Apple 17″ MacBook Pro Notebook, HP Photosmart Pro b9180
T200	Microsoft Office Professional 2010, Microsoft Wireless Optical Mouse 5000
T300	Logitech VX Nano Cordless Laser Mouse, Fellowes GEL Wrist Rest
T400	Dell Studio XPS 16 Notebook, Canon PowerShot SD1400
T500	Lenovo ThinkPad X200 Tablet PC, Symantec Norton Antivirus 2010
…	…

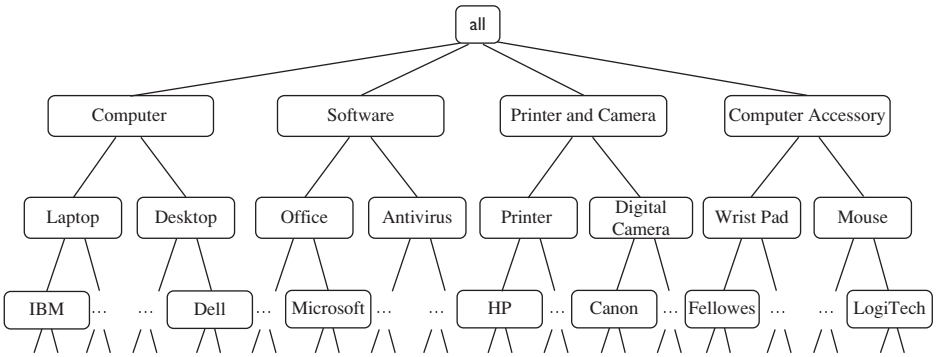


Figure 7.2 Concept hierarchy for *AllElectronics* computer items.

Concept hierarchies for nominal attributes are often implicit within the database schema, in which case they may be automatically generated using methods such as those described in Chapter 3. For our example, the concept hierarchy of Figure 7.2 was generated from data on product specifications. Concept hierarchies for numeric attributes can be generated using discretization techniques, many of which were introduced in Chapter 3. Alternatively, concept hierarchies may be specified by users familiar with the data such as store managers in the case of our example.

The items in Table 7.1 are at the lowest level of Figure 7.2's concept hierarchy. It is difficult to find interesting purchase patterns in such raw or primitive-level data. For instance, if "*Dell Studio XPS 16 Notebook*" or "*Logitech VX Nano Cordless Laser Mouse*" occurs in a very small fraction of the transactions, then it can be difficult to find strong associations involving these specific items. Few people may buy these items together, making it unlikely that the itemset will satisfy minimum support. However, we would expect that it is easier to find strong associations between generalized abstractions of these items, such as between "*Dell Notebook*" and "*Cordless Mouse.*" ■

Association rules generated from mining data at multiple abstraction levels are called **multiple-level** or **multilevel association rules**. Multilevel association rules can be

mined efficiently using concept hierarchies under a support-confidence framework. In general, a top-down strategy is employed, where counts are accumulated for the calculation of frequent itemsets at each concept level, starting at concept level 1 and working downward in the hierarchy toward the more specific concept levels, until no more frequent itemsets can be found. For each level, any algorithm for discovering frequent itemsets may be used, such as Apriori or its variations.

A number of variations to this approach are described next, where each variation involves "playing" with the support threshold in a slightly different way. The variations are illustrated in Figures 7.3 and 7.4, where nodes indicate an item or itemset that has been examined, and nodes with thick borders indicate that an examined item or itemset is frequent.

- **Using uniform minimum support for all levels** (referred to as **uniform support**): The same minimum support threshold is used when mining at each abstraction level. For example, in Figure 7.3, a minimum support threshold of 5% is used throughout (e.g., for mining from "*computer*" downward to "*laptop computer*"). Both "*computer*" and "*laptop computer*" are found to be frequent, whereas "*desktop computer*" is not.

  When a uniform minimum support threshold is used, the search procedure is simplified. The method is also simple in that users are required to specify only

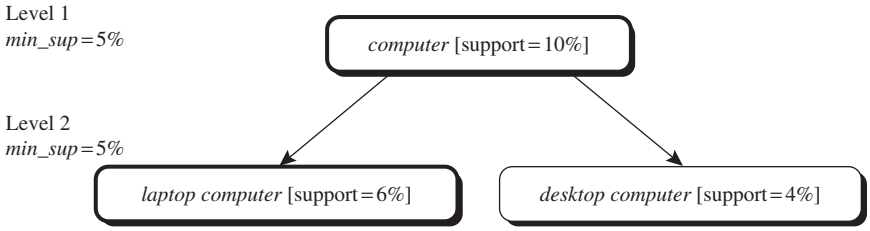


Figure 7.3 Multilevel mining with uniform support.

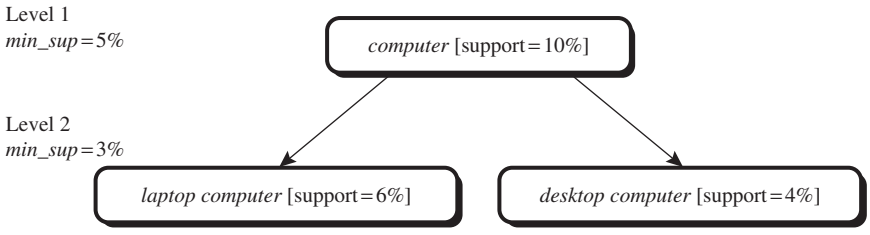


Figure 7.4 Multilevel mining with reduced support.

one minimum support threshold. An Apriori-like optimization technique can be adopted, based on the knowledge that an ancestor is a superset of its descendants: The search avoids examining itemsets containing any item of which the ancestors do not have minimum support.

The uniform support approach, however, has some drawbacks. It is unlikely that items at lower abstraction levels will occur as frequently as those at higher abstraction levels. If the minimum support threshold is set too high, it could miss some meaningful associations occurring at low abstraction levels. If the threshold is set too low, it may generate many uninteresting associations occurring at high abstraction levels. This provides the motivation for the next approach.

- **Using reduced minimum support at lower levels** (referred to as **reduced support**): Each abstraction level has its own minimum support threshold. The deeper the abstraction level, the smaller the corresponding threshold. For example, in Figure 7.4, the minimum support thresholds for levels 1 and 2 are 5% and 3%, respectively. In this way, "*computer,*" "*laptop computer,*" and "*desktop computer*" are all considered frequent.
- **Using item or group-based minimum support (referred to as group-based support)**: Because users or experts often have insight as to which groups are more important than others, it is sometimes more desirable to set up user-specific, item, or group-based minimal support thresholds when mining multilevel rules. For example, a user could set up the minimum support thresholds based on product price or on items of interest, such as by setting particularly low support thresholds for "*camera with price over $1000*" or "*Tablet PC,*" to pay particular attention to the association patterns containing items in these categories.

  For mining patterns with mixed items from groups with different support thresholds, usually the lowest support threshold among all the participating groups is taken as the support threshold in mining. This will avoid filtering out valuable patterns containing items from the group with the lowest support threshold. In the meantime, the minimal support threshold for each individual group should be kept to avoid generating uninteresting itemsets from each group. Other interestingness measures can be used after the itemset mining to extract truly interesting rules.

Notice that the Apriori property may not always hold uniformly across all of the items when mining under reduced support and group-based support. However, efficient methods can be developed based on the extension of the property. The details are left as an exercise for interested readers.

A serious side effect of mining multilevel association rules is its generation of many redundant rules across multiple abstraction levels due to the "ancestor" relationships among items. For example, consider the following rules where "*laptop computer*" is an ancestor of "*Dell laptop computer*" based on the concept hierarchy of Figure 7.2, and

where X is a variable representing customers who purchased items in *AllElectronics* transactions.

$$buys(X, \text{“}laptop\ computer\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”})$$
$$[support = 8\%, confidence = 70\%] \quad (7.4)$$

$$buys(X, \text{“}Dell\ laptop\ computer\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”})$$
$$[support = 2\%, confidence = 72\%] \quad (7.5)$$

“*If Rules (7.4) and (7.5) are both mined, then how useful is Rule (7.5)? Does it really provide any novel information?*” If the latter, less general rule does not provide new information, then it should be removed. Let's look at how this may be determined. A rule $R1$ is an **ancestor** of a rule $R2$, if $R1$ can be obtained by replacing the items in $R2$ by their ancestors in a concept hierarchy. For example, Rule (7.4) is an ancestor of Rule (7.5) because “*laptop computer*” is an ancestor of “*Dell laptop computer.*” Based on this definition, a rule can be considered redundant if its support and confidence are close to their “expected” values, based on an ancestor of the rule.

Example 7.2 **Checking redundancy among multilevel association rules.** Suppose that Rule (7.4) has a 70% confidence and 8% support, and that about one-quarter of all “*laptop computer*” sales are for “*Dell laptop computers.*” We may expect Rule (7.5) to have a confidence of around 70% (since all data samples of “*Dell laptop computer*” are also samples of “*laptop computer*”) and a support of around 2% (i.e., $8\% \times \frac{1}{4}$). If this is indeed the case, then Rule (7.5) is not interesting because it does not offer any additional information and is less general than Rule (7.4). ■

7.2.2 Mining Multidimensional Associations

So far, we have studied association rules that imply a single predicate, that is, the predicate *buys*. For instance, in mining our *AllElectronics* database, we may discover the Boolean association rule

$$buys(X, \text{“}digital\ camera\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”}). \quad (7.6)$$

Following the terminology used in multidimensional databases, we refer to each distinct predicate in a rule as a dimension. Hence, we can refer to Rule (7.6) as a **single-dimensional** or **intradimensional association rule** because it contains a single distinct predicate (e.g., *buys*) with multiple occurrences (i.e., the predicate occurs more than once within the rule). Such rules are commonly mined from transactional data.

Instead of considering transactional data only, sales and related information are often linked with relational data or integrated into a data warehouse. Such data stores are multidimensional in nature. For instance, in addition to keeping track of the items purchased in sales transactions, a relational database may record other attributes associated

with the items and/or transactions such as the item description or the branch location of the sale. Additional relational information regarding the customers who purchased the items (e.g., customer age, occupation, credit rating, income, and address) may also be stored. Considering each database attribute or warehouse dimension as a predicate, we can therefore mine association rules containing *multiple* predicates such as

$$age(X, \text{“20}\ldots\text{29”}) \wedge occupation(X, \text{“}student\text{”}) \Rightarrow buys(X, \text{“}laptop\text{”}). \quad (7.7)$$

Association rules that involve two or more dimensions or predicates can be referred to as **multidimensional association rules**. Rule (7.7) contains three predicates (*age, occupation*, and *buys*), each of which occurs *only once* in the rule. Hence, we say that it has **no repeated predicates**. Multidimensional association rules with no repeated predicates are called **interdimensional association rules**. We can also mine multidimensional association rules with repeated predicates, which contain multiple occurrences of some predicates. These rules are called **hybrid-dimensional association rules**. An example of such a rule is the following, where the predicate *buys* is repeated:

$$age(X, \text{“20}\ldots\text{29”}) \wedge buys(X, \text{“}laptop\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”}). \quad (7.8)$$

Database attributes can be nominal or quantitative. The values of **nominal** (or categorical) attributes are “names of things.” Nominal attributes have a finite number of possible values, with no ordering among the values (e.g., *occupation, brand, color*). **Quantitative** attributes are numeric and have an implicit ordering among values (e.g., *age, income, price*). Techniques for mining multidimensional association rules can be categorized into two basic approaches regarding the treatment of quantitative attributes.

In the first approach, *quantitative attributes are discretized using predefined concept hierarchies.* This discretization occurs before mining. For instance, a concept hierarchy for *income* may be used to replace the original numeric values of this attribute by interval labels such as “0..20K,” “21K..30K,” “31K..40K,” and so on. Here, discretization is *static* and predetermined. Chapter 3 on data preprocessing gave several techniques for discretizing numeric attributes. The discretized numeric attributes, with their interval labels, can then be treated as nominal attributes (where each interval is considered a category). We refer to this as **mining multidimensional association rules using static discretization of quantitative attributes**.

In the second approach, *quantitative attributes are discretized or clustered into “bins” based on the data distribution.* These bins may be further combined during the mining process. The discretization process is *dynamic* and established so as to satisfy some mining criteria such as maximizing the confidence of the rules mined. Because this strategy treats the numeric attribute values as quantities rather than as predefined ranges or categories, association rules mined from this approach are also referred to as **(dynamic) quantitative association rules**.

Let’s study each of these approaches for mining multidimensional association rules. For simplicity, we confine our discussion to interdimensional association rules. Note that rather than searching for frequent itemsets (as is done for single-dimensional association rule mining), in multidimensional association rule mining we search for

frequent *predicate sets*. A ***k*-predicate set** is a set containing k conjunctive predicates. For instance, the set of predicates {*age, occupation, buys*} from Rule (7.7) is a 3-predicate set. Similar to the notation used for itemsets in Chapter 6, we use the notation L_k to refer to the set of frequent k-predicate sets.

7.2.3 Mining Quantitative Association Rules

As discussed earlier, relational and data warehouse data often involve quantitative attributes or measures. We can discretize quantitative attributes into multiple intervals and then treat them as nominal data in association mining. However, such simple discretization may lead to the generation of an enormous number of rules, many of which may not be useful. Here we introduce three methods that can help overcome this difficulty to discover novel association relationships: (1) a data cube method, (2) a clustering-based method, and (3) a statistical analysis method to uncover exceptional behaviors.

Data Cube–Based Mining of Quantitative Associations

In many cases quantitative attributes can be discretized before mining using predefined concept hierarchies or data discretization techniques, where numeric values are replaced by interval labels. Nominal attributes may also be generalized to higher conceptual levels if desired. If the resulting task-relevant data are stored in a relational table, then any of the frequent itemset mining algorithms we have discussed can easily be modified so as to find all frequent predicate sets. In particular, instead of searching on only one attribute like *buys*, we need to search through all of the relevant attributes, treating each attribute–value pair as an itemset.

Alternatively, the transformed multidimensional data may be used to construct a *data cube*. Data cubes are well suited for the mining of multidimensional association rules: They store aggregates (e.g., counts) in multidimensional space, which is essential for computing the support and confidence of multidimensional association rules. An overview of data cube technology was presented in Chapter 4. Detailed algorithms for data cube computation were given in Chapter 5. Figure 7.5 shows the lattice of cuboids defining a data cube for the dimensions *age, income*, and *buys*. The cells of an n-dimensional cuboid can be used to store the support counts of the corresponding n-predicate sets. The base cuboid aggregates the task-relevant data by *age, income*, and *buys*; the 2-D cuboid, (*age, income*), aggregates by *age* and *income*, and so on; the 0-D (apex) cuboid contains the total number of transactions in the task-relevant data.

Due to the ever-increasing use of data warehouse and OLAP technology, it is possible that a data cube containing the dimensions that are of interest to the user may already exist, fully or partially materialized. If this is the case, we can simply fetch the corresponding aggregate values or compute them using lower-level materialized aggregates, and return the rules needed using a rule generation algorithm. Notice that even in this case, the Apriori property can still be used to prune the search space. If a given k-predicate set has support *sup*, which does not satisfy minimum support, then further

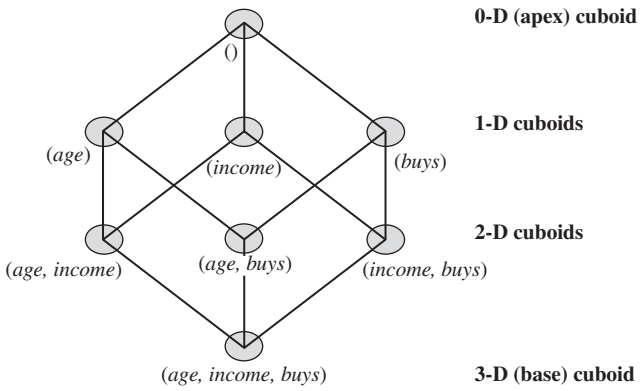


Figure 7.5 Lattice of cuboids, making up a 3-D data cube. Each cuboid represents a different group-by. The base cuboid contains the three predicates *age, income*, and *buys*.

exploration of this set should be terminated. This is because any more-specialized version of the *k*-itemset will have support no greater than *sup* and, therefore, will not satisfy minimum support either. In cases where no relevant data cube exists for the mining task, we must create one on-the-fly. This becomes an iceberg cube computation problem, where the minimum support threshold is taken as the iceberg condition (Chapter 5).

Mining Clustering-Based Quantitative Associations

Besides using discretization-based or data cube–based data sets to generate quantitative association rules, we can also generate *quantitative association rules* by clustering data in the quantitative dimensions. (Recall that objects within a cluster are similar to one another and dissimilar to those in other clusters.) The general assumption is that interesting frequent patterns or association rules are in general found at relatively dense clusters of quantitative attributes. Here, we describe a top-down approach and a bottom-up approach to clustering that finds quantitative associations.

A typical top-down approach for finding clustering-based quantitative frequent patterns is as follows. For each quantitative dimension, a standard clustering algorithm (e.g., *k*-means or a density-based clustering algorithm, as described in Chapter 10) can be applied to find clusters in this dimension that satisfy the minimum support threshold. For each cluster, we then examine the 2-D spaces generated by combining the cluster with a cluster or nominal value of another dimension to see if such a combination passes the minimum support threshold. If it does, we continue to search for clusters in this 2-D region and progress to even higher-dimensional combinations. The Apriori pruning still applies in this process: If, at any point, the support of a combination does not have minimum support, its further partitioning or combination with other dimensions cannot have minimum support either.

A bottom-up approach for finding clustering-based frequent patterns works by first clustering in high-dimensional space to form clusters with support that satisfies the minimum support threshold, and then projecting and merging those clusters in the space containing fewer dimensional combinations. However, for high-dimensional data sets, finding high-dimensional clustering itself is a tough problem. Thus, this approach is less realistic.

Using Statistical Theory to Disclose Exceptional Behavior

It is possible to discover quantitative association rules that disclose exceptional behavior, where "exceptional" is defined based on a statistical theory. For example, the following association rule may indicate exceptional behavior:

$$sex = female \Rightarrow meanwage = \$7.90/hr \ (overall\_mean\_wage = \$9.02/hr). \tag{7.9}$$

This rule states that the average wage for females is only \$7.90/hr. This rule is (subjectively) interesting because it reveals a group of people earning a significantly lower wage than the average wage of \$9.02/hr. (If the average wage was close to \$7.90/hr, then the fact that females also earn \$7.90/hr would be "uninteresting.")

An integral aspect of our definition involves applying statistical tests to confirm the validity of our rules. That is, Rule (7.9) is only accepted if a statistical test (in this case, a Z-test) confirms that with high confidence it can be inferred that the mean wage of the female population is indeed lower than the mean wage of the rest of the population. (The above rule was mined from a real database based on a 1985 U.S. census.)

An association rule under the new definition is a rule of the form:

$$population\_subset \Rightarrow \textbf{mean}\_of\_values\_for\_the\_subset, \tag{7.10}$$

where the mean of the subset is significantly different from the mean of its complement in the database (and this is validated by an appropriate statistical test).

7.2.4 Mining Rare Patterns and Negative Patterns

All the methods presented so far in this chapter have been for mining frequent patterns. Sometimes, however, it is interesting to find patterns that are rare instead of frequent, or patterns that reflect a negative correlation between items. These patterns are respectively referred to as rare patterns and negative patterns. In this subsection, we consider various ways of defining rare patterns and negative patterns, which are also useful to mine.

Example 7.3 **Rare patterns and negative patterns.** In jewelry sales data, sales of diamond watches are rare; however, patterns involving the selling of diamond watches could be interesting. In supermarket data, if we find that customers frequently buy Coca-Cola Classic or Diet Coke but not both, then buying Coca-Cola Classic and buying Diet Coke together

is considered a negative (correlated) pattern. In car sales data, a dealer sells a few fuel-thirsty vehicles (e.g., SUVs) to a given customer, and then later sells hybrid mini-cars to the same customer. Even though buying SUVs and buying hybrid mini-cars may be negatively correlated events, it can be interesting to discover and examine such exceptional cases. ■

An **infrequent** (or **rare**) **pattern** is a pattern with a frequency support that is *below* (or *far below*) a user-specified minimum support threshold. However, since the occurrence frequencies of the majority of itemsets are usually below or even far below the minimum support threshold, it is desirable in practice for users to specify other conditions for rare patterns. For example, if we want to find patterns containing at least one item with a value that is over $500, we should specify such a constraint explicitly. Efficient mining of such itemsets is discussed under mining multidimensional associations (Section 7.2.1), where the strategy is to adopt multiple (e.g., item- or group-based) minimum support thresholds. Other applicable methods are discussed under constraint-based pattern mining (Section 7.3), where user-specified constraints are pushed deep into the iterative mining process.

There are various ways we could define a negative pattern. We will consider three such definitions.

Definition 7.1: If itemsets X and Y are both frequent but rarely occur together (i.e., $sup(X \cup Y) < sup(X) \times sup(Y)$), then itemsets X and Y are **negatively correlated**, and the pattern $X \cup Y$ is a **negatively correlated pattern**. If $sup(X \cup Y) \ll sup(X) \times sup(Y)$, then X and Y are **strongly negatively correlated**, and the pattern $X \cup Y$ is a **strongly negatively correlated pattern**. □

This definition can easily be extended for patterns containing k-itemsets for $k > 2$.

A problem with the definition, however, is that it is not *null-invariant*. That is, its value can be misleadingly influenced by null transactions, where a *null-transaction* is a transaction that does not contain any of the itemsets being examined (Section 6.3.3). This is illustrated in Example 7.4.

Example 7.4 **Null-transaction problem with Definition 7.1.** If there are a lot of null-transactions in the data set, then the number of null-transactions rather than the patterns observed may strongly influence a measure's assessment as to whether a pattern is negatively correlated. For example, suppose a sewing store sells needle packages A and B. The store sold 100 packages each of A and B, but only one transaction contains both A and B. Intuitively, A is negatively correlated with B since the purchase of one does not seem to encourage the purchase of the other.

Let's see how the above Definition 7.1 handles this scenario. If there are 200 transactions, we have $sup(A \cup B) = 1/200 = 0.005$ and $sup(A) \times sup(B) = 100/200 \times 100/200 = 0.25$. Thus, $sup(A \cup B) \ll sup(A) \times sup(B)$, and so Definition 7.1 indicates that A and B are strongly negatively correlated. What if, instead of only 200 transactions in the database, there are 10^6? In this case, there are many null-transactions, that is, many contain neither A nor B. How does the definition hold up? It computes $sup(A \cup B) = 1/10^6$ and $sup(X) \times sup(Y) = 100/10^6 \times 100/10^6 = 1/10^8$.

Thus, $sup(A \cup B) \gg sup(X) \times sup(Y)$, which contradicts the earlier finding even though the number of occurrences of A and B has not changed. The measure in Definition 7.1 is not null-invariant, where *null-invariance* is essential for quality interestingness measures as discussed in Section 6.3.3. ■

Definition 7.2: If X and Y are strongly negatively correlated, then

$$sup(X \cup \overline{Y}) \times sup(\overline{X} \cup Y) \gg sup(X \cup Y) \times sup(\overline{X} \cup \overline{Y}).$$

Is this measure null-invariant? □

Example 7.5 Null-transaction problem with Definition 7.2. Given our needle package example, when there are in total 200 transactions in the database, we have

$$sup(A \cup \overline{B}) \times sup(\overline{A} \cup B) = 99/200 \times 99/200 = 0.245$$
$$\gg sup(A \cup B) \times sup(\overline{A} \cup \overline{B}) = 199/200 \times 1/200 \approx 0.005,$$

which, according to Definition 7.2, indicates that A and B are strongly negatively correlated. What if there are 10^6 transactions in the database? The measure would compute

$$sup(A \cup \overline{B}) \times sup(\overline{A} \cup B) = 99/10^6 \times 99/10^6 = 9.8 \times 10^{-9}$$
$$\ll sup(A \cup B) \times sup(\overline{A} \cup \overline{B}) = 199/10^6 \times (10^6 - 199)/10^6 \approx 1.99 \times 10^{-4}.$$

This time, the measure indicates that A and B are positively correlated, hence, a contradiction. The measure is not null-invariant. ■

As a third alternative, consider Definition 7.3, which is based on the Kulczynski measure (i.e., the average of conditional probabilities). It follows the spirit of interestingness measures introduced in Section 6.3.3.

Definition 7.3: Suppose that itemsets X and Y are both frequent, that is, $sup(X) \geq min\_sup$ and $sup(Y) \geq min\_sup$, where min_sup is the minimum support threshold. If $(P(X|Y) + P(Y|X))/2 < \epsilon$, where ϵ is a negative pattern threshold, then pattern $X \cup Y$ is a **negatively correlated pattern**. □

Example 7.6 Negatively correlated patterns using Definition 7.3, based on the Kulczynski measure. Let's reexamine our needle package example. Let min_sup be 0.01% and $\epsilon = 0.02$. When there are 200 transactions in the database, we have $sup(A) = sup(B) = 100/200 = 0.5 > 0.01\%$ and $(P(B|A) + P(A|B))/2 = (0.01 + 0.01)/2 < 0.02$; thus A and B are negatively correlated. Does this still hold true if we have many more transactions? When there are 10^6 transactions in the database, the measure computes $sup(A) = sup(B) = 100/10^6 = 0.01\% \geq 0.01\%$ and $(P(B|A) + P(A|B))/2 = (0.01 + 0.01)/2 < 0.02$, again indicating that A and B are negatively correlated. This matches our intuition. The measure does not have the null-invariance problem of the first two definitions considered.

Let's examine another case: Suppose that among 100,000 transactions, the store sold 1000 needle packages of A but only 10 packages of B; however, every time package B is

sold, package A is also sold (i.e., they appear in the same transaction). In this case, the measure computes $(P(B|A) + P(A|B))/2 = (0.01 + 1)/2 = 0.505 \gg 0.02$, which indicates that A and B are positively correlated instead of negatively correlated. This also matches our intuition. ■

With this new definition of negative correlation, efficient methods can easily be derived for mining negative patterns in large databases. This is left as an exercise for interested readers.

7.3 Constraint-Based Frequent Pattern Mining

A data mining process may uncover thousands of rules from a given data set, most of which end up being unrelated or uninteresting to users. Often, users have a good sense of which "direction" of mining may lead to interesting patterns and the "form" of the patterns or rules they want to find. They may also have a sense of "conditions" for the rules, which would eliminate the discovery of certain rules that they know would not be of interest. Thus, a good heuristic is to have the users specify such intuition or expectations as *constraints* to confine the search space. This strategy is known as **constraint-based mining**. The constraints can include the following:

- **Knowledge type constraints:** These specify the type of knowledge to be mined, such as association, correlation, classification, or clustering.
- **Data constraints:** These specify the set of task-relevant data.
- **Dimension/level constraints:** These specify the desired dimensions (or attributes) of the data, the abstraction levels, or the level of the concept hierarchies to be used in mining.
- **Interestingness constraints:** These specify thresholds on statistical measures of rule interestingness such as support, confidence, and correlation.
- **Rule constraints:** These specify the form of, or conditions on, the rules to be mined. Such constraints may be expressed as metarules (rule templates), as the maximum or minimum number of predicates that can occur in the rule antecedent or consequent, or as relationships among attributes, attribute values, and/or aggregates.

These constraints can be specified using a high-level declarative data mining query language and user interface.

The first four constraint types have already been addressed in earlier sections of this book and this chapter. In this section, we discuss the use of *rule constraints* to focus the mining task. This form of constraint-based mining allows users to describe the rules that they would like to uncover, thereby making the data mining process more *effective*. In addition, a sophisticated mining query optimizer can be used to exploit the constraints specified by the user, thereby making the mining process more *efficient*.

Constraint-based mining encourages interactive exploratory mining and analysis. In Section 7.3.1, you will study metarule-guided mining, where syntactic rule constraints are specified in the form of rule templates. Section 7.3.2 discusses the use of *pattern space pruning* (which prunes patterns being mined) and *data space pruning* (which prunes pieces of the data space for which further exploration cannot contribute to the discovery of patterns satisfying the constraints).

For pattern space pruning, we introduce three classes of properties that facilitate constraint-based search space pruning: *antimonotonicity, monotonicity*, and *succinctness*. We also discuss a special class of constraints, called *convertible constraints*, where by proper data ordering, the constraints can be pushed deep into the iterative mining process and have the same pruning power as monotonic or antimonotonic constraints. For data space pruning, we introduce two classes of properties—*data succinctness* and *data antimonotonicty*—and study how they can be integrated within a data mining process.

For ease of discussion, we assume that the user is searching for association rules. The procedures presented can be easily extended to the mining of correlation rules by adding a correlation measure of interestingness to the support-confidence framework.

7.3.1 Metarule-Guided Mining of Association Rules

"*How are metarules useful?*" Metarules allow users to specify the syntactic form of rules that they are interested in mining. The rule forms can be used as constraints to help improve the efficiency of the mining process. Metarules may be based on the analyst's experience, expectations, or intuition regarding the data or may be automatically generated based on the database schema.

Example 7.7 **Metarule-guided mining.** Suppose that as a market analyst for *AllElectronics* you have access to the data describing customers (e.g., customer age, address, and credit rating) as well as the list of customer transactions. You are interested in finding associations between customer traits and the items that customers buy. However, rather than finding *all* of the association rules reflecting these relationships, you are interested only in determining which pairs of customer traits promote the sale of office software. A metarule can be used to specify this information describing the form of rules you are interested in finding. An example of such a metarule is

$$P_1(X, Y) \wedge P_2(X, W) \Rightarrow buys(X, \text{"}office\ software\text{"}), \tag{7.11}$$

where P_1 and P_2 are **predicate variables** that are instantiated to attributes from the given database during the mining process, X is a variable representing a customer, and Y and W take on values of the attributes assigned to P_1 and P_2, respectively. Typically, a user will specify a list of attributes to be considered for instantiation with P_1 and P_2. Otherwise, a default set may be used.

In general, a metarule forms a hypothesis regarding the relationships that the user is interested in probing or confirming. The data mining system can then search for

rules that match the given metarule. For instance, Rule (7.12) matches or **complies with** Metarule (7.11):

$$age(X, \text{“30..39”}) \wedge income(X, \text{“41}K\text{..60}K\text{”}) \Rightarrow buys(X, \text{“}office\ software\text{”}). \quad (7.12)$$

■

“*How can metarules be used to guide the mining process?*” Let’s examine this problem closely. Suppose that we wish to mine interdimensional association rules such as in Example 7.7. A metarule is a rule template of the form

$$P_1 \wedge P_2 \wedge \cdots \wedge P_l \Rightarrow Q_1 \wedge Q_2 \wedge \cdots \wedge Q_r, \quad (7.13)$$

where P_i $(i = 1, \ldots, l)$ and Q_j $(j = 1, \ldots, r)$ are either instantiated predicates or predicate variables. Let the number of predicates in the metarule be $p = l + r$. To find interdimensional association rules satisfying the template,

- We need to find all frequent p-predicate sets, L_p.
- We must also have the support or count of the l-predicate subsets of L_p to compute the confidence of rules derived from L_p.

This is a typical case of mining multidimensional association rules. By extending such methods using the constraint-pushing techniques described in the following section, we can derive efficient methods for metarule-guided mining.

7.3.2 Constraint-Based Pattern Generation: Pruning Pattern Space and Pruning Data Space

Rule constraints specify expected set/subset relationships of the variables in the mined rules, constant initiation of variables, and constraints on aggregate functions and other forms of constraints. Users typically employ their knowledge of the application or data to specify rule constraints for the mining task. These rule constraints may be used together with, or as an alternative to, metarule-guided mining. In this section, we examine rule constraints as to how they can be used to make the mining process more efficient. Let’s study an example where rule constraints are used to mine hybrid-dimensional association rules.

Example 7.8 **Constraints for mining association rules.** Suppose that *AllElectronics* has a sales multidimensional database with the following interrelated relations:

- *item(item_ID, item_name, description, category, price)*
- *sales(transaction_ID, day, month, year, store_ID, city)*
- *trans_item(item_ID, transaction_ID)*

Here, the *item* table contains attributes *item_ID, item_name, description, category*, and *price*; the *sales* table contains attributes *transaction_ID day, month, year, store_ID*, and *city*; and the two tables are linked via the foreign key attributes, *item_ID* and *transaction_ID*, in the table *trans_item*.

Suppose our association mining query is "*Find the patterns or rules about the sales of which cheap items (where the sum of the prices is less than $10) may promote (i.e., appear in the same transaction) the sales of which expensive items (where the minimum price is $50), shown in the sales in Chicago in 2010.*"

This query contains the following four constraints: (1) $sum(I.price) < \$10$, where I represents the *item_ID* of a cheap item; (2) $min(J.price) \geq \$50$), where J represents the *item_ID* of an expensive item; (3) $T.city = Chicago$; and (4) $T.year = 2010$, where T represents a *transaction_ID*. For conciseness, we do not show the mining query explicitly here; however, the constraints' context is clear from the mining query semantics. ■

Dimension/level constraints and interestingness constraints can be applied after mining to filter out discovered rules, although it is generally more efficient and less expensive to use them *during* mining to help prune the search space. Dimension/level constraints were discussed in Section 7.2, and interestingness constraints, such as support, confidence, and correlation measures, were discussed in Chapter 6. Let's focus now on rule constraints.

"*How can we use rule constraints to prune the search space? More specifically, what kind of rule constraints can be 'pushed' deep into the mining process and still ensure the completeness of the answer returned for a mining query?*"

In general, an efficient frequent pattern mining processor can prune its search space during mining in two major ways: *pruning pattern search space* and *pruning data search space*. The former checks candidate patterns and decides whether a pattern can be pruned. Applying the Apriori property, it prunes a pattern if no superpattern of it can be generated in the remaining mining process. The latter checks the data set to determine whether the particular data piece will be able to contribute to the subsequent generation of satisfiable patterns (for a particular pattern) in the remaining mining process. If not, the data piece is pruned from further exploration. A constraint that may facilitate pattern space pruning is called a *pattern pruning constraint*, whereas one that can be used for data space pruning is called a *data pruning constraint*.

Pruning Pattern Space with Pattern Pruning Constraints

Based on how a constraint may interact with the pattern mining process, there are five categories of pattern mining constraints: (1) *antimonotonic,* (2) *monotonic,* (3) *succinct,* (4) *convertible,* and (5) *inconvertible*. For each category, we use an example to show its characteristics and explain how such kinds of constraints can be used in the mining process.

The first category of constraints is **antimonotonic**. Consider the rule constraint "$sum(I.price) \leq \$100$" of Example 7.8. Suppose we are using the Apriori framework, which explores itemsets of size k at the kth iteration. If the price summation of the items in a candidate itemset is no less than \$100, this itemset can be pruned from the search space, since adding more items into the set (assuming price is no less than zero) will only make it more expensive and thus will never satisfy the constraint. In other words, if an itemset does not satisfy this rule constraint, none of its supersets can satisfy the constraint. If a rule constraint obeys this property, it is **antimonotonic**. Pruning by antimonotonic constraints can be applied at each iteration of Apriori-style algorithms to help improve the efficiency of the overall mining process while guaranteeing completeness of the data mining task.

The Apriori property, which states that all nonempty subsets of a frequent itemset must also be frequent, is antimonotonic. If a given itemset does not satisfy minimum support, none of its supersets can. This property is used at each iteration of the Apriori algorithm to reduce the number of candidate itemsets examined, thereby reducing the search space for association rules.

Other examples of antimonotonic constraints include "$min(J.price) \geq \$50$," "$count(I) \leq 10$," and so on. Any itemset that violates either of these constraints can be discarded since adding more items to such itemsets can never satisfy the constraints. Note that a constraint such as "$avg(I.price) \leq \$10$" is not antimonotonic. For a given itemset that does not satisfy this constraint, a superset created by adding some (cheap) items may result in satisfying the constraint. Hence, pushing this constraint inside the mining process will not guarantee completeness of the data mining task. A list of SQL primitives–based constraints is given in the first column of Table 7.2. The antimonotonicity of the constraints is indicated in the second column. To simplify our discussion, only existence operators (e.g., $=$, $\in$, but not $\neq$, $\notin$) and comparison (or containment) operators with equality (e.g., $\leq$, $\subseteq$) are given.

The second category of constraints is **monotonic**. If the rule constraint in Example 7.8 were "$sum(I.price) \geq \$100$," the constraint-based processing method would be quite different. If an itemset I satisfies the constraint, that is, the sum of the prices in the set is no less than \$100, further addition of more items to I will increase cost and will always satisfy the constraint. Therefore, further testing of this constraint on itemset I becomes redundant. In other words, if an itemset satisfies this rule constraint, so do all of its supersets. If a rule constraint obeys this property, it is **monotonic**. Similar rule monotonic constraints include "$min(I.price) \leq \$10$," "$count(I) \geq 10$," and so on. The monotonicity of the list of SQL primitives–based constraints is indicated in the third column of Table 7.2.

The third category is **succinct constraints**. For this constraints category, we can *enumerate all and only those sets that are guaranteed to satisfy the constraint.* That is, if a rule constraint is **succinct**, we can directly generate precisely the sets that satisfy it, even before support counting begins. This avoids the substantial overhead of the generate-and-test paradigm. In other words, such constraints are *precounting prunable.* For example, the constraint "$min(J.price) \geq \$50$" in Example 7.8 is succinct because we can explicitly and precisely generate all the itemsets that satisfy the constraint.

Table 7.2 Characterization of Commonly Used SQL-Based Pattern Pruning Constraints

Constraint	*Antimonotonic*	*Monotonic*	*Succinct*
$v \in S$	no	yes	yes
$S \supseteq V$	no	yes	yes
$S \subseteq V$	yes	no	yes
$min(S) \leq v$	no	yes	yes
$min(S) \geq v$	yes	no	yes
$max(S) \leq v$	yes	no	yes
$max(S) \geq v$	no	yes	yes
$count(S) \leq v$	yes	no	weakly
$count(S) \geq v$	no	yes	weakly
$sum(S) \leq v\ (\forall a \in S, a \geq 0)$	yes	no	no
$sum(S) \geq v\ (\forall a \in S, a \geq 0)$	no	yes	no
$range(S) \leq v$	yes	no	no
$range(S) \geq v$	no	yes	no
$avg(S)\ \theta\ v, \theta \in \{\leq, \geq\}$	convertible	convertible	no
$support(S) \geq \xi$	yes	no	no
$support(S) \leq \xi$	no	yes	no
$all\_confidence(S) \geq \xi$	yes	no	no
$all\_confidence(S) \leq \xi$	no	yes	no

Specifically, such a set must consist of a nonempty set of items that have a price no less than \$50. It is of the form S, where $S \neq \emptyset$ is a subset of the set of all items with prices no less than \$50. Because there is a precise "formula" for generating all the sets satisfying a succinct constraint, there is no need to iteratively check the rule constraint during the mining process. The succinctness of the list of SQL primitives–based constraints is indicated in the fourth column of Table 7.2.[2]

The fourth category is **convertible constraints**. Some constraints belong to none of the previous three categories. However, if the items in the itemset are arranged in a particular order, the constraint may become monotonic or antimonotonic with regard to the frequent itemset mining process. For example, the constraint "$avg(I.price) \leq \$10$" is neither antimonotonic nor monotonic. However, if items in a transaction are added to an itemset in price-ascending order, the constraint becomes *antimonotonic*, because if an itemset I violates the constraint (i.e., with an average price greater than \$10), then further addition of more expensive items into the itemset will never make it

[2]For constraint $count(S) \leq v$ (and similarly for $count(S) \geq v$), we can have a member generation function based on a cardinality constraint (i.e., $\{X \mid X \subseteq Itemset \wedge |X| \leq v\}$). Member generation in this manner is of a different flavor and thus is called *weakly succinct*.

satisfy the constraint. Similarly, if items in a transaction are added to an itemset in price-descending order, it becomes *monotonic,* because if the itemset satisfies the constraint (i.e., with an average price no greater than \$10), then adding cheaper items into the current itemset will still make the average price no greater than \$10. Aside from "$avg(S) \leq v$" and "$avg(S) \geq v$," given in Table 7.2, there are many other convertible constraints such as "$variance(S) \geq v$" "$standard\_deviation(S) \geq v$," and so on.

Note that the previous discussion does not imply that every constraint is convertible. For example, "$sum(S)\,\theta\,v$," where $\theta \in \{\leq, \geq\}$ and each element in S could be of any real value, is not convertible. Therefore, there is yet a fifth category of constraints, called **inconvertible constraints**. The good news is that although there still exist some tough constraints that are not convertible, most simple SQL expressions with built-in SQL aggregates belong to one of the first four categories to which efficient constraint mining methods can be applied.

Pruning Data Space with Data Pruning Constraints

The second way of search space pruning in constraint-based frequent pattern mining is *pruning data space*. This strategy prunes pieces of data if they will not contribute to the subsequent generation of satisfiable patterns in the mining process. We consider two properties: *data succinctness* and *data antimonotonicity*.

Constraints are **data-succinct** if they can be used *at the beginning of a pattern mining process* to prune the data subsets that cannot satisfy the constraints. For example, if a mining query requires that the mined pattern must contain *digital camera*, then any transaction that does not contain *digital camera* can be pruned at the beginning of the mining process, which effectively reduces the data set to be examined.

Interestingly, many constraints are **data-antimonotonic** in the sense that *during the mining process*, if a data entry cannot satisfy a data-antimonotonic constraint based on the current pattern, then it can be pruned. We prune it because it will not be able to contribute to the generation of any superpattern of the current pattern in the remaining mining process.

Example 7.9 **Data antimonotonicity.** A mining query requires that $C_1 : sum(I.price) \geq \$100$, that is, the sum of the prices of the items in the mined pattern must be no less than \$100. Suppose that the current frequent itemset, S, does not satisfy constraint C_1 (say, because the sum of the prices of the items in S is \$50). If the remaining frequent items in a transaction T_i are such that, say, $\{i_2.price = \$5, i_5.price = \$10, i_8.price = \$20\}$, then T_i will not be able to make S satisfy the constraint. Thus, T_i cannot contribute to the patterns to be mined from S, and thus can be pruned.

Note that such pruning cannot be done at the beginning of the mining because at that time, we do not know yet if the total sum of the prices of all the items in T_i will be over \$100 (e.g., we may have $i_3.price = \$80$). However, during the iterative mining process, we may find some items (e.g., i_3) that are not frequent with S in the transaction data set, and thus they would be pruned. Therefore, such checking and pruning should be enforced at each iteration to reduce the data search space. ■

Notice that constraint C_1 is a monotonic constraint with respect to pattern space pruning. As we have seen, this constraint has very limited power for reducing the search space in pattern pruning. However, the same constraint can be used for effective reduction of the data search space.

For an antimonotonic constraint, such as $C_2 : sum(I.price) \leq \$100$, we can prune both pattern and data search spaces at the same time. Based on our study of pattern pruning, we already know that the current itemset can be pruned if the sum of the prices in it is over \$100 (since its further expansion can never satisfy C_2). At the same time, we can also prune any remaining items in a transaction T_i that cannot make the constraint C_2 valid. For example, if the sum of the prices of items in the current itemset S is \$90, any patterns over \$10 in the remaining frequent items in T_i can be pruned. If none of the remaining items in T_i can make the constraint valid, the entire transaction T_i should be pruned.

Consider pattern constraints that are neither antimonotonic nor monotonic such as "$C_3 : avg(I.price) \leq 10$." These can be data-antimonotonic because if the remaining items in a transaction T_i cannot make the constraint valid, then T_i can be pruned as well. Therefore, data-antimonotonic constraints can be quite useful for constraint-based data space pruning.

Notice that search space pruning by data antimonotonicity is confined only to a pattern growth–based mining algorithm because the pruning of a data entry is determined based on whether it can contribute to a specific pattern. Data antimonotonicity cannot be used for pruning the data space if the Apriori algorithm is used because the data are associated with all of the currently active patterns. At any iteration, there are usually many active patterns. A data entry that cannot contribute to the formation of the superpatterns of a given pattern may still be able to contribute to the superpattern of other active patterns. Thus, the power of data space pruning can be very limited for nonpattern growth–based algorithms.

7.4 Mining High-Dimensional Data and Colossal Patterns

The frequent pattern mining methods presented so far handle large data sets having a small number of dimensions. However, some applications may need to mine *high-dimensional data* (i.e., data with hundreds or thousands of dimensions). Can we use the methods studied so far to mine high-dimensional data? The answer is unfortunately negative because the search spaces of such typical methods grow exponentially with the number of dimensions.

Researchers have overcome this difficulty in two directions. One direction extends a pattern growth approach by further exploring the vertical data format to handle data sets with a large number of *dimensions* (also called *features* or *items*, e.g., genes) but a *small* number of *rows* (also called *transactions* or *tuples*, e.g., samples). This is useful in applications like the analysis of gene expressions in bioinformatics, for example, where we often need to analyze microarray data that contain a *large* number of genes

(e.g., 10,000 to 100,000) but only a *small* number of samples (e.g., 100 to 1000). The other direction develops a new mining methodology, called *Pattern-Fusion*, which mines *colossal patterns*, that is, patterns of very long length.

Let's first briefly examine the first direction, in particular, a pattern growth–based row enumeration approach. Its general philosophy is to explore the *vertical data format*, as described in Section 6.2.5, which is also known as **row enumeration**. Row enumeration differs from traditional column (i.e., item) enumeration (also known as the *horizontal data format*). In traditional column enumeration, the data set, D, is viewed as a set of rows, where each row consists of an itemset. In row enumeration, the data set is instead viewed as an itemset, each consisting of a set of *row_ID*s indicating where the item appears in the traditional view of D. The original data set, D, can easily be transformed into a transposed data set, T. A data set with a small number of rows but a large number of dimensions is then transformed into a transposed data set with a large number of rows but a small number of dimensions. Efficient pattern growth methods can then be developed on such relatively low-dimensional data sets. The details of such an approach are left as an exercise for interested readers.

The remainder of this section focuses on the second direction. We introduce Pattern-Fusion, a new mining methodology that mines *colossal patterns* (i.e., patterns of very long length). This method takes leaps in the pattern search space, leading to a good approximation of the complete set of colossal frequent patterns.

7.4.1 Mining Colossal Patterns by Pattern-Fusion

Although we have studied methods for mining frequent patterns in various situations, many applications have hidden patterns that are tough to mine, due mainly to their immense length or size. Consider bioinformatics, for example, where a common activity is DNA or microarray data analysis. This involves mapping and analyzing very long DNA and protein sequences. Researchers are more interested in finding large patterns (e.g., long sequences) than finding small ones since larger patterns usually carry more significant meaning. We call these large patterns *colossal* patterns, as distinguished from patterns with large support sets. Finding colossal patterns is challenging because incremental mining tends to get "trapped" by an explosive number of midsize patterns before it can even reach candidate patterns of large size. This is illustrated in Example 7.10.

Example 7.10 The challenge of mining colossal patterns. Consider a 40×40 square table where each row contains the integers 1 through 40 in increasing order. Remove the integers on the diagonal, and this gives a 40×39 table. Add 20 identical rows to the bottom of the table, where each row contains the integers 41 through 79 in increasing order, resulting in a 60×39 table (Figure 7.6). We consider each row as a transaction and set the minimum support threshold at 20. The table has an exponential number (i.e., $\binom{40}{20}$) of midsize closed/maximal frequent patterns of size 20, but only one that is colossal: $\alpha = (41, 42, \ldots, 79)$ of size 39. None of the frequent pattern mining algorithms that we have introduced so far can complete execution in a reasonable amount of time.

row/col	1	2	3	4	…	38	39
1	2	3	4	5	…	39	40
2	1	3	4	5	…	39	40
3	1	2	4	5	…	39	40
4	1	2	3	5	…	39	40
5	1	2	3	4	…	39	40
…	…	…	…	…	…	…	…
39	1	2	3	4	…	38	40
40	1	2	3	4	…	38	39
41	41	42	43	44	…	78	79
42	41	42	43	44	…	78	79
…	…	…	…	…	…	…	…
60	41	42	43	44	…	78	79

Figure 7.6 A simple colossal patterns example: The data set contains an exponential number of midsize patterns of size 20 but only one that is colossal, namely $(41, 42, \ldots, 79)$.

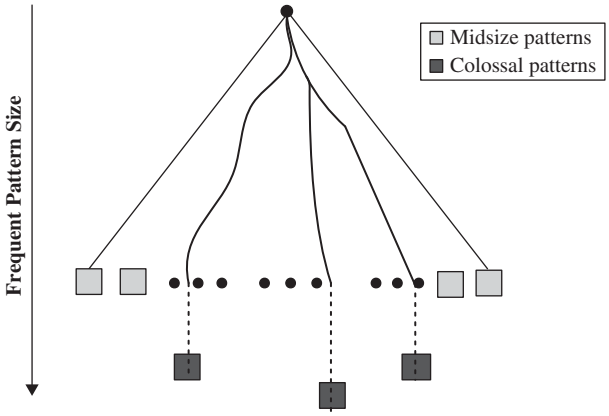


Figure 7.7 Synthetic data that contain some colossal patterns but exponentially many midsize patterns.

The pattern search space is similar to that in Figure 7.7, where midsize patterns largely outnumber colossal patterns. ■

All of the pattern mining strategies we have studied so far, such as Apriori and FP-growth, use an incremental growth strategy by nature, that is, they increase the length of candidate patterns by one at a time. Breadth-first search methods like Apriori cannot bypass the generation of an explosive number of midsize patterns generated,