DATABASE MANAGEMENT SYSTEMS

CSE Semester V (CSC 502) – 2018
IT Semester III (ITC 304) – 2017

About the Author

G K Gupta retired after a thirty-five year distinguished career in computer science education, research and academic management. He is currently Adjunct Professor of Computer Science at the Faculty of Information Technology at Monash University, Clayton.

Professor Gupta was born and raised in India and completed undergraduate studies at the University of Roorkee (now IIT, Roorkee). He did Master's in Computer Science at the University of Waterloo in Canada and then migrated to Australia in 1972 where he joined the Department of Computer Science at Monash University. While working full-time, he obtained a doctorate from the same university.

In 1986, Professor Gupta moved to James Cook University as Foundation Professor and Head to establish a new department of computer science. He also served as Dean of the School of Information Technology at Bond University from1998 to 2002. He came back to Monash University in October 2002 as Deputy Dean. Here he also served as Acting Dean before returning to the position of Professor of Computer Science. Professor Gupta was seconded by the Australian Government to the Asian Institute of Technology in Bangkok for two years 1979–1981, to help set up a new department there. He has also spent a year at the University of Illinois at Urbana-Champaign, another year at Bell Laboratories in New Jersey, eight months at International University in Germany and made two three-month visits to VIT University at Vellore in India.

Professor Gupta is a Fellow of the Association of Computing Machinery (ACM), a Fellow of the Australian Computer Society (ACS) and a Senior Member of the IEEE. He was awarded the inaugural Distinguished Service Award by the Computer Science Research and Education (CORE) society in 2004. Professor Gupta has previously authored the book *Introduction to Data Mining with Case Studies*; published in 2006.

DATABASE MANAGEMENT SYSTEMS

CSE Semester V (CSC 502) – 2018
IT Semester III (ITC 304) – 2017

G K Gupta

Adjunct Professor of Computer Science
Monash University, Clayton
Australia

McGraw Hill Education (India) Private Limited
CHENNAI

McGraw Hill Education Offices

Chennai New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

McGraw Hill Education (India) Private Limited

Published by McGraw Hill Education (India) Private Limited
444/1, Sri Ekambara Naicker Industrial Estate, Alapakkam, Porur, Chennai 600 116

Database Management Systems

1 2 3 4 5 6 7 8 9 D102739 22 21 20 19 18

Printed and bound in India.

Print-Book Edition
ISBN (13): 978-93-5316-139-2
ISBN (10): 93-5316-139-8

E-Book Edition
ISBN (13): 978-93-5316-140-8
ISBN (10): 93-5316-140-1

Director—Science & Engineering Portfolio: *Vibha Mahajan*

Senior Portfolio Manager—Science & Engineering: *Hemant K Jha*
Associate Portfolio Manager—Science & Engineering: *Tushar Mishra*

Production Head: *Satinder S Baveja*
Copy Editor: *Taranpreet Kaur*

General Manager—Production: *Rajender P Ghansela*
Manager—Production: *Reji Kumar*

Typeset at Text-o-Graphics, B-1/56, Aravali Apartment, Sector-34, Noida 201 301, and printed at

Cover Designer: APS Compugraphics
Cover Image Source: Shutterstock

Cover Printer:

Visit us at: www.mheducation.co.in

Write to us at: info.india@mheducation.com

CIN: U22200TN1970PTC111531

Toll Free Number: 1800 103 5875

To

the memory of

Ammaji

who passed away on 15 May 2005
at the age estimated to be 99 years
(Her date of birth was not known)

Contents

Preface

Database technology occupies a very important position in our increasingly computerized society. Almost all computer systems whether they are supermarket checkout systems, air traffic control systems, railways seat reservation systems, planning systems in governments or automatic teller machines (ATMs) with which most of us interact in our daily lives in urban India, have a database system at their core. Therefore, all computer science, information technology, business administration and information science students need to learn about the basic theory and practice of database systems since it is fundamental to much of computer technology.

This book is intended to be a comprehensive text for undergraduate and graduate level database management courses offered in computer science, information systems and business management disciplines. It can be used as a primary read through one-semester by students pursuing BCA, MCA, BSc, MSc and related computer science courses. It can be used as a reference reading by BTech, BE and MTech students and industry professionals.

This book is based on many years of experience of teaching database courses at a number of universities including James Cook University, Monash University and Bond University in Australia, Asian Institute of Technology (AIT) in Thailand, International University in Germany, and VIT University in India. Writing this book has been a wonderful experience for me.

What makes this book unique?

The book differs from other books on database management since it is written especially for students in India and that is why the examples used in the book are primarily those that I hope will interest Indian students. One major example used to illustrate most database concepts that is carried throughout the book is based on cricket One Day International (ODI) matches. Some other real-life examples have also been included. These examples have been used to illustrate several concepts in database design, query processing, distributed databases, object-oriented databases and a number of other topics.

Instead of focusing on how to build the database management system software, this textbook focuses on how to build database applications since most students once they graduate will be developing database applications rather than building database software. Therefore in this book, we concentrate on the concepts underlying the relational data model rather than on any particular commercial DBMS, as these concepts are the foundation of database processing. In a similar way, in the portion of the book which studies transaction processing, we concentrate on the concepts underlying the properties of transactions and the technical issues involved in their implementation, rather than how they are implemented in any particular commercial DBMS.

This text has a logical flow of topics and is supplemented with numerous chapter-end pedagogical features have been provided to enhance practical learning along with several examples and real-life case studies for holistic understanding of the subject. Important definitions and key terms have been emphasized in every chapter for a good grasp of concepts.

Book Overview

The contents of the book are described ahead briefly.

Chapter 1 introduces the subject with a simple example of a cricket database and its features followed by a definition of the term 'database'. A number of popular websites are listed to illustrate the amount of information that many websites now have and is required to be stored in databases. **Chapter 2** discusses the Entity-Relationship model in detail. Representing an Entity-Relationship model by an Entity-Relationship diagram is explained along with generalization, specialization and aggregation concepts. **Chapter 3** describes the relational model and its three major components, viz., data structure, data manipulation and data integrity. Mapping of the E-R model to the relational model is explained. This is supported by a brief description of older database models: hierarchical and network models. **Chapter 4** elucidates data manipulation in relational model including relational algebra and relational calculus. A number of examples of their use are also presented. **Chapter 5** on SQL discusses data retrieval features of SQL using simple examples that involve only one table, followed by examples that involve more than one table using joins and subqueries, as well as built-in functions and GROUPING and HAVING clauses. Use of SQL in enforcing data integrity is also mentioned. **Chapter 6** explains theoretical basis of normalization using the concept of functional dependency followed by a study of single-valued normalization and multivalued normalization.

The structure of information storage on a magnetic disk is described along with the most commonly used data structures, B-tree and hashing in **Chapter 7**. In **Chapter 8**, the concept and properties of a transaction are described along with concepts of a schedule and serializability. Locking and two-phase locking (2PL) are introduced followed by definition of deadlocks, their detection, prevention and resolution.

To the Instructor

This book deals with the most important topics of database management. The book includes many examples which I believe are essential in getting students motivated and in helping them understand the course content. To further this aim, I have also presented varied revision exercises. At the end of each chapter, I have included a list of references in the book that encompasses some of the most highly cited papers in database management.

To the Student

Database management is an important and interesting area and I hope you will enjoy learning about it. The book is written primarily for you. I have provided many examples to help you learn the concepts and the techniques. A large number of references have been provided, highlighting a small number, specially selected for you since they are easy to read. I understand that most students do not have the time to read references other than the prescribed textbooks and lecture notes but these references should be useful if you decide to learn more about database techniques during your studies or afterwards.

Review, short-answer and multiple-choice questions, and exercises have been provided at the end of every chapter. Please try to solve all of these to better understand the material in the book.

Acknowledgements

I would like to thank a number of people who have helped in reading, correcting and giving ideas during the writing of this book. In particular Joe Incigneri read an earlier manuscript in 2009 and helped in improving the readability of the book. Jeanette Niehus has helped in drawing all the diagrams using Microsoft Visio.

I also gratefully acknowledge the suggestions given by the reviewers during the initial process. Their comments have added value to parts of this book.

G K Gupta

Publisher's Note

McGraw Hill Education (India) invites suggestions and comments from you, all of which can be sent to info.india@mheducation.com (Kindly mention the title and author name in the subject line).

Piracy-related issues may also be reported.

Syllabus

Database Management System

CSE Semester V (CSC 502)–2018

Module No.	Unit No.	Topics	Hrs.
1.0		**Introduction Database Concepts:**	4
	1.1	• Introduction, Characteristics of Databases • File System v/s Database System • Users of Databse System	
	1.2	• Data Independence • DBMS System Architecture • Database Administrator	
2.0		**Entity-Relationship Data Model**	
	2.1	• The Entity – Relationship (ER) Model: Entity Types : Weak and Strong Entity Sets, Entity Sets, Types of Attributes, Keys, Relationship Constraints : Cardinality and Participation, Extended Entity-Relationship (EER) Model: Generalization, Specialization and Aggregation	8
3.0		**Relational Model and Relational Algebra**	8
	3.1	• Introduction to the Relational Model, Relational Schema and Concept of Keys • Mapping the ER and EER Model to the Relational Model	
	3.2	• Relational Algebra – Unary and Set Oprations, Relational Algebra Queries	
4.0		**Structured Query Language (SQL)**	12
	4.1	• Overview of SQL • Data Definition Commands, Data Manipulation Commands, Data Control Commands, Transaction Control Commands	
	4.2	• Set and String Operations, Aggregate Function – Group by, Having • Views in SQL, Joins, Nested and Complex Queries, Integrity Constraints: - Key Constraints, Domain Constraints, Referential Integrity, Check Constraints	
	4.3	• Triggers	

Module No.	Unit No.	Topics	Hrs.
5.0		**Relational–Database Design**	**8**
	5.1	• Pitfalls in Relational – Database Designs, Concept of Normalization • Function Dependencies, First Normal Form, 2nd, 3rd, BCNF, Multi Valued Dependencies, 4NF	
6.0		**Transactions Management and Concurrency**	**12**
	6.1	• Transaction Concept, Transaction States, ACID Properties • Concurrent Executions, Serializability – Conflict and View, Concurrency Control: Lock-based, Timestamp-based Protocols	
	6.2	• Recovery System: Failure Classification, Log Based Recovery, ARIES, Checkpoint, Shadow Paging • Deadlock Handling	
		Total	**52**

ROADMAP TO THE SYLLABUS

Database Management Systems
CSE Semester V (CSC 502)–2018

Unit-I: Introduction to Database Concepts

Example of a Database; What is a Database?; Characteristics of Databases; Data Governance and Importance of Databases; History of Database Software; File systems vs Database Systems; What is a DBMS?;Users of a Database System; Advantages of using an Enterprise Database; Concerns when using an Enterprise Database; Three-Level DBMS Architecture and Data Abstraction; Data Independence; DBMS System Architecture; Database Administrator; Designing an Enterprise Database System; Future of Databases—Web Databases; Choosing a DBMS

GO TO Chapter 1: Introduction to Database Management

Unit-II: Entity Relationship Data Model

Benefits of Data Modeling; Types of Models; Phases of Database Modeling; Entity-Relationship Model: Generalization, Specialization and Aggregation; Extended Entity-Relationship Model; Database Design Process; Strengths of the E-R Model; Weaknesses of the E-R Model; A Case Study of Building an E-R Model; Evaluating Data Model Quality

GO TO Chapter 2: Entity–Relationship Data Model

Unit-III: Relational Model and Relational Algebra

Relational Model: Data Structure; Mapping the E-R Model to the Relational Model; Data Manipulation: Data Integrity; Advantages of the Relational Model; Rules for Fully Relational Systems; Earlier Database Models—Hierarchical and Network Models

Relational Algebra: Relational Algebra; Relational Algebra Queries; Relational Calculus; Relational Algebra vs Relational Calculus

GO TO Chapter 3: Relational Model
Chapter 4: Relational Algebra and Relational Calculus

Unit-IV: Structured Query Language

SQL; Data Definition—CREATE, ALTER and DROP Commands; Data Manipulation—SQL Data Retrieval; Views—Using Virtual Tables in SQL; Using SQL in Procedural Programming Languages—Embedded SQL and Dynamic SQL; Data Integrity and Constraints; Triggers and Active Databases; Data Control—Database Security; Further Data Definition Commands; Summary of Some SQL Commands; SQL Standards

GO TO Chapter 5: Structured Query Language (SQL)

Unit-V: Relational Database Design

Normalization: Possible Undesirable Properties of Relations and Schema Refinement; Functional Dependency—Single-Valued Dependencies; Single-valued Normalization; Desirable Properties of Decompositions; Multivalued Dependencies; Denormalization; Inclusion Dependency

Physical Storage and Indexing: Different Types of Computer Memories; Secondary Storage—Hard Disks; Buffer Management; Introduction to File Structures; Unstructured and Unordered Files (Heap Files); Sequential Files (Sorted Files); Index and Index Types; Indexed-Sequential File; B-tree; B+-tree; Advantages and Disadvantages of B-trees and B+-trees;Static Hashing; External Hashing—Files on Disk; Dynamic Hashing for External Files; Inverted Files

GO TO Chapter 6: Normalization
Chapter 7: Physical Storage and Indexing

Unit-VI: Transaction Management and Concurrency

Concept of a Transaction; Examples of Concurrency Anomalies; Schedules and Recoverability; Serializability; Hierarchy of Serializable Schedules; Concurrency Control and Enforcing Serializability; Deadlocks; Lock Granularity; Multiple Granularity and Intention Locking; Nonlocking Technique—Timestamping Control; Nonlocking Technique—Optimistic Control; Transaction Support in SQL; Evaluation of Concurrency Control Mechanisms

GO TO Chapter 8: Transaction Management and Concurrency

Syllabus

Database Management System

IT Semester III (ITC 304)–2017

Sr. No.	Module	Detailed Content	Hours	CO Mapping
0	**Prerequisites**	Basic Knowledge of Operating Systems and File Systems, any Programming Knowledge	02	
I	**Introduction Database Concepts**	Introduction, Characteristics of Databases, File System v/s Database System. Users of a Database System Data Models. Schemas, and Instances, Three-Schema Architecture and Data Independence, Database Administrator (DBA), Role of a DBA	05	CO1
II	**Entity-Relationship Data Model**	Conceptual Modeling of a database, The Entity-Relationship (ER) Model, Entity Types, Entity Sets, Attributes, and Keys, Relationship Types, Relationship Sets, Weak Entity Types Generalization, Specialization and Aggregation, Extended Entity-Relationship (EER) Model	09	CO2
III	**Relational Model and Relational Algebra**	Introduction to Relational Model, Relational Model Constraints and Relational Database Schemas, Concept of Keys: Primary Kay, Secondary Key, Foreign Key, Mapping the ER and EER Model to the Relational Model, Introduction to Relational Algebra, Relational Algebra expressions for • Unary Relational Operations, • Set Theory Operations • Binary Relational operation Relational Algebra Queries	09	CO2
IV	**Structured Query Language (SQL)**	Overview of SQL, Data Definition Commands, Set operations, Aggregate function, Null Values, Data Manipulation commands, Data Control Commands, Views in SQL, Complex Retrieval Queries Using Group By, Recursive Queries, Nested Queries; Referential Integrity in SQL, Event Condition Action (ECA) model (Triggers) in SQL; Database Programming with JDBC, Security and Authorization in SQL, Functions and Procedures in SQL and Cursors	10	CO3, CO4

Sr. No.	Module	Detailed Content	Hours	CO Mapping
V	**Relational-Database Design**	Design guidelines for Relational Schema, Functional Dependencies, Definition of Normal Forms. 1NF, 2NF, 3NF, BCNF, Converting Relational Schema to Higher Normal Forms	08	CO5
VI	**Storage and Indexing**	Operation on Files, Hashing Techniques, Types of Indexes; Single-Level Ordered Indexes; Multilevel Indexes; Overview of B-Trees and B+-Trees; Indexes on Multiple Keys	09	CO6

ROADMAP TO THE SYLLABUS

Database Management Systems
IT Semester III (ITC 304)–2017

Unit-I: Introduction to Database Concepts

Example of a Database; What is a Database?; Characteristics of Databases; Data Governance and Importance of Databases; History of Database Software; File systems vs Database Systems; What is a DBMS?;Users of a Database System; Advantages of using an Enterprise Database; Concerns when using an Enterprise Database; Three-Level DBMS Architecture and Data Abstraction; Data Independence; DBMS System Architecture; Database Administrator; Designing an Enterprise Database System; Future of Databases—Web Databases; Choosing a DBMS

GO TO Chapter 1: Introduction to Database Management

Unit-II: Entity Relationship Data Model

Benefits of Data Modeling; Types of Models; Phases of Database Modeling; Entity-Relationship Model: Generalization, Specialization and Aggregation; Extended Entity-Relationship Model; Database Design Process; Strengths of the E-R Model; Weaknesses of the E-R Model; A Case Study of Building an E-R Model; Evaluating Data Model Quality

GO TO Chapter 2: Entity–Relationship Data Model

Unit-III: Relational Model and Relational Algebra

Relational Model: Data Structure; Mapping the E-R Model to the Relational Model; Data Manipulation: Data Integrity; Advantages of the Relational Model; Rules for Fully Relational Systems; Earlier Database Models—Hierarchical and Network Models

Relational Algebra: Relational Algebra; Relational Algebra Queries; Relational Calculus; Relational Algebra vs Relational Calculus

GO TO Chapter 3: Relational Model
Chapter 4: Relational Algebra and Relational Calculus

Unit-IV: Structured Query Language

SQL; Data Definition—CREATE, ALTER and DROP Commands; Data Manipulation—SQL Data Retrieval; Views—Using Virtual Tables in SQL; Using SQL in Procedural Programming Languages—Embedded SQL and Dynamic SQL; Data Integrity and Constraints; Triggers and Active Databases; Data Control—Database Security; Further Data Definition Commands; Summary of Some SQL Commands; SQL Standards

GO TO Chapter 5: Structured Query Language (SQL)

Unit-V: Relational Database Design

Possible Undesirable Properties of Relations and Schema Refinement; Functional Dependency—Single-Valued Dependencies; Single-valued Normalization; Desirable Properties of Decompositions; Multivalued Dependencies; Denormalization; Inclusion Dependency

GO TO Chapter 6: Normalization

Unit-VI: Storage and Indexing

Different Types of Computer Memories; Secondary Storage—Hard Disks; Buffer Management; Introduction to File Structures; Unstructured and Unordered Files (Heap Files); Sequential Files (Sorted Files); Index and Index Types; Indexed-Sequential File; B-tree; B+-tree; Advantages and Disadvantages of B-trees and B+-trees;Static Hashing; External Hashing—Files on Disk; Dynamic Hashing for External Files; Inverted Files

GO TO Chapter 7: Physical Storage and Indexing

CHAPTER 1

Introduction to Database Management

OBJECTIVES

- Introduce the importance of information for every enterprise.
- Discuss and define the concept of a database.
- Describe data governance and why databases are becoming so important.
- Briefly describe the evolution of database software.
- Describe differences between file systems and database systems.
- Explain database management systems (DBMS). The advantages and disadvantages of using an enterprise database.
- Discuss the three-level database architecture and data abstraction.
- Describe the different types of database users.
- Explain the concept of data independence.
- Present a DBMS system architecture.

KEYWORDS

Database, database management, database management systems, DBMS, data independence, data abstraction, metadata, file systems, advantages and disadvantages of using a database system, three-tier architecture, DBMS architecture and system architecture, end users, catalog, concurrency, transaction management, recovery, applications programmer, web databases, database history, database administrator, database design, requirements, testing, maintaining, importance of databases, characteristics of databases, selecting DBMS software, application programs, systems analysts, application programmers, cricket database, consistency.

The more this power of concentration, the more knowledge is acquired, because this is the one and only method of acquiring knowledge... in doing anything, the stronger the power of concentration, the better will that thing be done. This is the one call, the one knock, which opens the gate of nature, and lets out the flood of light.

***Swami Vivekananda* (1863–1902)**

1.1 INTRODUCTION

Most modern enterprises except some small ones have data that they need to store in ways which will be easy to retrieve later. One of the most common types of database is a list of names and addresses of people who regularly deal with the enterprise. Originally when a business is small such lists may have been written on paper, then in a word processor or spreadsheet, but as the business and lists grow it becomes difficult to deal with the information stored in a word processor or a spreadsheet. Database technology may then be required.

Database management is about managing information resources of an enterprise. The enterprise may be a big company, a small business, a government department or a club; each of them generate a variety of data. The data can be, for example, about customers, credit cards, telephone calls, lottery tickets, orders, library books, students, employees, airline flights, etc. All such data (or information, we won't discuss the difference between the two) for every type of enterprise is a valuable resource and therefore, just like any other valuable resource, it needs to be managed effectively. With the advent of the World Wide Web and its widespread use in business, the amount of data that is being generated is growing exponentially, making data management even more important. Imagine the data being generated and stored by companies like amazon.com, eBay, Google, YouTube and Wikipedia that needs to be managed. On the other hand, consider the amount of data being generated by mobile phones in India. A lot of data must be stored for accounting purposes (information about every call and each sms must be stored for accounting) while other data, like the content of various sms messages, must also be stored for retrieval incase of security checks or more serious criminal proceedings. Thus, the amount of data being generated in the modern world using new technologies is mind boggling.

This chapter is organized as follows. A simple example of a cricket database is presented in the next section and a number of features of this database are discussed. In Section 1.3, the term database is defined and examples related to cricket are presented. An example of a university database is also presented. A number of popular websites are listed to show the amount of information that many websites now carry and which must be stored in databases. Section 1.4 describes a number of characteristics of databases and Section 1.5 discusses data governance and importance of databases for enterprises. Section 1.6 presents a brief history of database software.

Section 1.7 discusses the differences between file systems and database systems. A definition of database management system (DBMS) is presented along with some features of a DBMS in Section 1.8. Variety of users of database systems is described in Section 1.9. The next two sections present a number of advantages and concerns of using a centralized enterprise database system in a large enterprise as compared to the branches maintaining local database systems using conventional file systems. Sections 1.12 and 1.13 are devoted to explaining the three-level DBMS architecture, data abstraction and data independence. A typical DBMS architecture diagram is presented in Section 1.14 along with a description of its different components.

The role of the database administrator is discussed in Section 1.15. Section 1.16 explains the steps involved in designing an enterprise database system. The chapter concludes with a brief introduction to web databases followed by a number of guidelines for selecting a DBMS before making a purchase.

1.2 AN EXAMPLE OF A DATABASE

A simple example of a database is presented below.

Example 1.1

Consider the table presented in Table 1.1 that presents a list of the top ten batsmen in one day international (ODI) cricket in terms of the total number of runs scored. The data was current on 21 August 2010.

Table 1.1 Example of a simple database[1]

Player	*Span*	*Matches*	*Innings*	*Runs*	*Avg.*	*Strike Rate*	*100s*
SR Tendulkar	1989–2010	442	431	17598	45.12	86.26	46
ST Jayasuriya	1989–2010	444	432	13428	32.43	91.22	28
RT Ponting	1995–2010	351	342	13072	42.85	80.84	29
Inzamam-ul-Haq	1991–2007	378	350	11739	39.52	74.24	10
SC Ganguly	1992–2007	311	300	11363	41.02	73.70	22
JH Kallis	1996–2010	303	289	10838	45.72	72.72	17
R Dravid	1996–2009	339	313	10765	39.43	71.17	12
BC Lara	1990–2007	299	289	10405	40.48	79.51	19
Mohammed Yousuf	1998–2010	282	267	9624	42.39	75.15	15
AC Gilchrist	1996–2008	287	279	9619	35.89	96.94	16

It should be noted that in the above database the batting average for a player is not obtained by dividing the total number of runs scored by the total number of innings played. It is obtained by dividing the total runs scored by the total number of innings in which the player got out. The number of times the player was not out is not counted and the number of times a batsman was not out is not included in the table above because of space considerations. For example, Sachin Tendulkar was not out 38 times while Adam Gilchrist was not out only 11 times.

The information in Table 1.1 is quite interesting. There are three players in the top ten ODI batsmen order from India and none from England. Also the number of runs scored and the number of centuries made by Sachin Tendulkar are so high as compared to other players that it is unlikely he would be overtaken by another batsman, for example, Sanath Jayasuriya or Ricky Ponting, in the near future. Also, let us not forget that Tendulkar is an active cricketer who continues to score runs.

1. Full names of all Pakistani players have been used, since they are widely known by their full names.

There are some other things we should note about the database:

1. Each table in the database requires a name so that it can be easily identified by that name. We have named our table the *bestODIbatsmen*[2].
2. The names given to columns in the top row are of special importance in a database. They are the titles of the columns and each column is referred to by that name. If we wish to state the number of ODI matches played by a player, it will be referred to by the name *Matches*. The name *matches* with a lower case *m* will not be acceptable and other names like *Games* will also not work.
3. If some data has been typed incorrectly (and errors in databases are not uncommon), for example, if the name Tendulkar in the table above is mistyped as Tendulker, then there is no simple way to retrieve information about Tendulkar. Although any person interested in cricket would know that Tendulker is really Tendulkar, the computerized database has no intelligence.
4. The second column with the name *Span* gives entries like 1975–85 as the year the player started playing ODI cricket and the last year he played. This data is treated as one value and usually there is no simple way to separate the two dates and query the database on them separately.
5. The database above is ordered according to the total runs scored. Although such ordering is helpful when humans view the database, the ordering of data when stored in a computer is often not known and may change over time. For example, over time Ricky Ponting and Jacques Kallis are likely to make more runs and would be expected to climb up the ladder of best batsmen. Although unlikely, given the lead Sachin Tendulkar has, but Ricky Ponting in theory could take over as the highest scorer of ODI runs. In a computerized database, it is inefficient to be swapping rows of the database to maintain the desired order, when the computer can be used to sort the table, whenever the user needs it sorted on any column. For example, some users may want the table sorted on the batting average while others on the strike rate. As a corollary, there is no concept of the first row or the last row in a computerized database.
6. In the example database, a number of columns are related. For example, batting average is based on the total runs scored and the number of innings played (and the number of innings in which the player was not out), which is itself related to the number of matches played. Such relationships in a database are not desirable, since a user may by mistake increase the number of matches when a player starts a new match, and then forget to increase the number of innings, or the total number of runs scored, or the batting average. The data therefore from time to time could be inaccurate. This does not matter so much for our simple cricket database but it would not be acceptable, say, in a bank accounting database. There are ways of overcoming such problems which are discussed later.
7. A related problem with Table 1.1 is that all the information in it is an aggregation derived from information about each ODI. The example database in Table 1.1 has been used because the information is interesting for people who love cricket but it is not a very good idea to maintain tables of aggregates, since it is difficult to check the accuracy of the information, and it is easy to make a mistake in updating such summary information. If we store information about each ODI in the database then the aggregation can be carried out by the system ensuring there are no mistakes.

---

2. Once a table has a name, it may be written in a compact form by its schema. For example, Table 1.1 may be written as *bestODIbatsmen*(*Player, Span, Matches, Innings, Runs, Avg, Strike Rate, 100s)*. This notation has also been used later.

8. Furthermore, there are some, perhaps not so obvious, problems with our example database. Firstly, there is no convenient way to show that a player is still playing. For example, we have put 1989–2010 in *Span* for Sachin Tendulkar. This does not tell us that he is still playing. Although Adam Gilchrist played during 1996–2008, he has retired and therefore is not playing any more. Furthermore, the database assumes that a player always plays for only one country. We leave it to the reader to discover if there have been any players for which this is not true.

These issues and a variety of other issues related to computerized databases are discussed further in the following chapters.

1.3 WHAT IS A DATABASE?

Whether it is a cricket database or an airline booking database, databases have become essential in all aspects of modern life. It is often said that we live in an information society and that information is a very valuable resource (or, in other words, information is power). In this information society, the term *database* has become a rather common term although its meaning appears to have become somewhat vague, as the importance of database systems has grown. Some people use the term database of an organization to mean all the data in the organization (whether computerized or not). Other people use the term to mean the software that manages the data. We will use it to mean a collection of computerized information such that it is available to many people for various uses. A definition of a database may be given as.

> *A database is a well organized collection of data that are related in a meaningful way which can be accessed in different logical orders but are stored only once. The data in the database is therefore integrated, structured and shared.*

According to this definition, the simple database given in the cricket example in Table 1.1 is not really a database. We may consider it a *toy database* and will continue to keep calling it a database and use it when appropriate. Similarly a *database* that someone might have for personal use on his/her home PC would not be considered a database. Also, a collection of tables that are not related to each other, for example, a table of cricket players and a table of swimmers, unless they can be related through the country they come from or in some other way, cannot be considered a database.

The main features of data in a database are as follows:

- It is well organized.
- It is related.
- It is accessible in different orders without great difficulty.
- It is stored only once.

It is assumed that operations (for example, update, insert and retrieve) on the database can be carried out in a simple and flexible way. Also, since a database tends to be a long term resource of an organization, it is expected that planned as well as unplanned applications can (in general) be carried out without great difficulty.

Example 1.2

We have noted above that a single table like Table 1.1 is not really a database. Now let us add another table. This table is also a summary of performance of a number of players in ODI matches. Consider Table 1.2 given below which gives information about the best bowlers in ODI matches based on the total number of wickets taken before 21 August 2010. Information in this table is also quite interesting. There is still no player from the UK and only one player appears in both Tables 1.1 and 1.2. Only two bowlers from India and three from Sri Lanka appear in Table 1.2.

Table 1.2 The best ODI bowlers (Table Name: *bestbowlers*)

Player	*Span*	*Matches*	*Balls*	*Runs*	*Wickets*	*Ave*	*Econ*	*ST*
M Muralitharan	1993–2010	337	18169	11885	515	23.07	3.92	35.2
Wasim Akram	1984–2003	356	18186	11812	502	23.52	3.89	36.2
Waqar Younis	1989–2003	262	12698	9919	416	23.84	4.68	30.5
WPUJC Vaas	1994–2008	322	15775	11014	400	27.53	39.4	39.2
SM Pollock	1996–2008	303	15712	9631	393	24.50	3.67	39.9
GD McGrath	1993–2007	250	12970	8391	381	22.02	3.88	34.0
A Kumble	1990–2007	271	14496	10412	337	30.89	4.30	43.0
J Srinath	1991–2003	229	11935	8847	315	28.08	4.44	37.8
B Lee	2000–2009	185	9442	7428	323	22.99	4.72	29.2
ST Jayasuriya	1989–2009	444	14838	11825	322	36.72	4.78	46.0

Now let us ask the question: Do the two tables, Table 1.1 and Table 1.2, together form a database? Although the information is well organized, the information in the two tables is difficult to relate. We do have some common data, for example, the names of those players that appear in both the tables (ST Jayasuriya is the only one) but the relationship is not strong. So it may be concluded that these two tables also do not form a database.

Example 1.3

Let us now consider a third table. Table 1.3, called *Players*, includes information about all the players given in Table 1.1 and 1.2. As we have noted earlier, there is no concept of sorting of rows in a real database. Although the information in Table 1.3 has been sorted on the column Country but it could easily be sorted according to any other column. Also, there are no duplicates in the table.

Now let us again ask the same question as we did earlier. Do the three tables, Tables 1.1, 1.2 and 1.3 together form a database? By including this table, the relationship between the three tables becomes stronger. We can now find out which players are best ODI batsmen and come from India, which could not be done before. It is also now possible, for example, to prepare a list of all Indian players who are either best ODI batsmen or best ODI bowlers. We can therefore conclude that the three tables together could be called a database since they have the properties of a database and assuming the tables are being shared.

Table 1.3 List of some players that have played in ODIs (Table Name: *Players*)

Name	*Country*	*Place of birth*	*Year of birth*
RT Ponting	Australia	Launceston	1974
AC Gilchrist	Australia	Bellingen	1971
GD McGrath	Australia	Dubbo	1970
B Lee	Australia	Wollongong	1976
SR Tendulkar	India	Mumbai	1973
SC Ganguly	India	Calcutta	1972
R Dravid	India	Indore	1973
A Kumble	India	Bangalore	1970
J Srinath	India	Mysore	1969
Inzamam-ul-Haq	Pakistan	Multan	1970
Wasim Akram	Pakistan	Lahore	1966
Waqar Younis	Pakistan	Vehari	1969
Mohammed Yousuf	Pakistan	Lahore	1974
JH Kallis	South Africa	Cape Town	1975
SM Pollock	South Africa	Port Elizabeth	1973
ST Jayasuriya	Sri Lanka	Matara	1969
WPUJC Vaas	Sri Lanka	Muttumagala	1974
M Muralitharan	Sri Lanka	Kandy	1972
BC Lara	West Indies	Santa Cruz	1969

We have included limited information in the example tables because of space considerations. For example, we could include the number of times a batsman was not out, each batsman's highest score, each bowler's best bowling performance, each player's date of first ODI played and so on.

Just like in cricket, in every modern enterprise, a large amount of data is generated about its operations. This data is sometimes called *operational data*. The operational data includes the data an organization must necessarily maintain about its operation but does not include temporary results or any transient information.

Since data is a valuable resource for every enterprise, often a great deal of money is spent collecting, storing, maintaining and using it. The running of a modern enterprise depends on proper maintenance of its operational data.

A Case Study

The case study deals with the operational data of an enterprise; the enterprise being a university. A university's operational data includes information about courses, students, academic staff, other staff, and enrolments. There is much more information of course, for example, departments, library data, financial data as well as data about research and about university facilities.

The information may include the following:

1. Student personal data (studentID, student's name, gender, current address, home address, date of birth, nationality)
2. Student academic data (studentID, student's school results, courses completed and grades, current enrolment)
3. Academic staff data (staff ID, staff member's name, gender, current address, date of birth, nationality, academic qualifications, appointment history, current appointment, salary history, current salary, sabbatical leave information, recreational leave, sick leave)
4. Research data (projectID, researcherID, funding, publications, research students)
5. Non-academic staff data (staffID, staff member's name, gender, current address, date of birth, nationality, qualifications, experience, appointment history, current appointment, salary history, current salary, recreational leave, sick leave)
6. Courses (or units) offered data (courseID, course name, department, syllabus, lecturer, quota if any)
7. Financial data (budget information, receipts, expenditure)

The above data is only a very small sample of data that a university generates (for example, data generated by the library is not included and data about facilities, for example, buildings and equipment is not included) and we leave it to the reader to think of other information that should be included for each of the items listed above. In fact, in the early 1990s, the Australian universities initiated a project to develop a unified database model for all student information that could be used by all Australian universities. After months of effort by a team of experts and spending millions of dollars, the project was cancelled in spite of the team having developed the specifications of the information which was more than 1000-pages long. The difficulty was that not all the universities agreed with the collection of information that was developed.

The above data would have a number of classes of users, for example:

1. Instructors who need information about enrolments in the courses that they teach and heads of departments about finances of their departments. These are users that use the information in the database but do not develop their own software. They are often called *end users*.
2. Programmers in the IT services unit who develop programs (called *application programs*) to produce reports that are needed regularly by the government, university administration and the departments. These programmers are called *applications programmers*. They are also responsible for developing programs that assist end users in retrieving the information that they need in their work.
3. The Registrar or some other person is in-charge of the database and makes decisions about the kind information that is to be stored in the database, and also decides who can modify and access which parts of the database.

For a large modern enterprise, it is difficult to overestimate the value of its databases. From time to time, we read stories in the press describing enterprises that were not able to handle their data effectively. In fact, in 2002, there was a case of an Australian university using the wrong database software to process its data which resulted in a loss of millions of dollars. Some interesting stories of database and software development failures are given in Glass (1997, 2001).

To conclude, we note that data

- is a valuable resource and an investment and
- should be carefully managed like other resources (for example, manpower).

Example 1.4
Very Large Databases

The example given below illustrates the amount of information currently being circulated on the Web and its fast paced growth.

Computing has changed enormously since the early days of very large and very expensive computers. This is partly because of technological developments in computer hardware and software and partly because of the development of the World Wide Web. Some of these trends can be illustrated by the popularity of the following websites:

1. Google, Yahoo and MSN—Every day, millions of people use these three websites that enable people to search the Web, use chatrooms, free email and a number of other services.
2. Ebay—A site that enables people to sell and buy new and used items by way of a person-to-person auction.
3. YouTube—Every day, millions of people watch this website that enables them to put and share videos on the Web.
4. MySpace—Every day a large number of people visit this website that facilitates social networking.
5. Facebook— It is another social networking website that connects people through social networks and encourages online interaction through groups and communities.
6. Wikipedia—A free online collaborative encyclopaedia that includes a huge amount of information on almost every topic.
7. Skype—This is a nearly free person-to-person telecommunications application that uses the Internet for making phone calls.
8. Flickr—A large number of people use this website to share photos.
9. Blogger—Free personal journal publishing tool that allows people to publish their own blogs and share them with others.
10. Fotolog—A website that enables people to share photos and blogs.

The reader is encouraged to consider the size of these sites and find out how much information each of these sites carry. Most of these sites are growing rapidly and have millions of users. For example, Facebook is reported to have more than 500 million users in September 2010[3] (less than six years after it was started) and there are 30 billion pieces of content shared each month. MySpace, also of a similar size, had 100 billion rows of data, 14 billion comments on the site, 20 billion mails, 50 million mails per day (more than Yahoo, Hotmail or Google), 10 billion friend relationships, 1.5 billion images, and 8 million images being uploaded per day as per data published in 2008.

3. Refer to the statistics posted at
   http://www.facebook.com/press/info.php?statistic
   Also look at http://blog.compete.com/2009/02/09/facebook-myspace-twitter-social-network

How would one manage such a huge amount of information?

In addition to these trends, there are a number of other new trends in computing. For example, the popularity of devices like iPod and iPhone, the digitization of millions of old books, the development of Internet TV and the availability of wireless broadband. All these developments deal with processing, storing and communicating information. Although the computer was developed to solve numerical problems, it has increasingly become an information-handling device. However, the conceptual basis of the computer architecture has not changed so significantly.

1.4 CHARACTERISTICS OF DATABASES

Various characteristics distinguish the database approach from conventional file processing systems. Some of these differences are discussed in Section 1.7. In this section a list of a various characteristics of database systems is given.

1.4.1 Persistent Data

A database only stores persistent data and once data has been stored in a database system it always stays there unless deleted by an explicit request.

1.4.2 Metadata and Self-describing Nature

A database system contains not only the database itself but also the descriptions of the data structures and its constraints. This is called *metadata* which may be defined as *data about data*. In the context of a database, metadata includes information about how the data is stored in the computer storage. Metadata, discussed in more detail later, is an integral part of the database and stored in the database itself. It is used not only by database managers but also by database end-users. This separation makes database system very different from traditional file-based systems in which data definition was a part of the application programs.

1.4.3 Data Independence

Perhaps the most important characteristic of a database system is data independence which may be defined as the ability to separate the database definition from the physical storage organization of the database. Therefore, information about the physical structure of the database is stored in the system catalog (the database metadata) and not in the application programs. Therefore any changes in the physical structure of the database do not impact the application programs. Data independence is discussed in more detail in Section 1.13.

1.4.4 Access Flexibility and Security

In modern database systems no restrictions are placed on the content the users can access unless the user is not authorized to access some data. Security is enhanced by using the concept of database views which

are subsets of the database. Different views may be defined for different classes of users thereby controlling access to the database system. Each view might contain only the data relevant for a user or a group of users.

Most modern database systems, other than those used for personal use, are multiuser systems that allow multiple users to access the system at the same time. Providing this facility to the users requires concurrency control to ensure that several users accessing the same data item at the same time do so without violating the database consistency.

1.5 DATA GOVERNANCE AND IMPORTANCE OF DATABASES

An enterprise has many assets (for example, people, property, intellectual property) but an important asset whose value is often not recognized is the enterprise data. This is however changing in today's competitive world. Every enterprise needs a 'single version of the truth' but it is not always possible to find that version. Many years ago, the company, Coca Cola, could not even be sure of the number of bottles of soft drinks they were producing each day, in all their plants. Therefore, many enterprises are now recognizing the importance of *data governance*. Data governance essentially refers to decision-making regarding effective management of an enterprise's data including data quality, metadata, and data access.

Since the 1960s there has been a continual and rapid growth in the use of database technology which can deal with some aspects of data governance. Databases have become an important part of every company with data that needs to be managed. Not only has there been a dramatic growth in the number of such systems but also enormous growth in the amount of information stored in them. The complexity of applications of database systems has also being growing simultaneously. A report produced by Lyman and Varian at the University of California at Berkeley is a fascinating study on the amount of information in the world and its pace of growth.

Databases contain an enterprise's data or what we call information. As noted earlier, we do not wish to discuss the meanings of data, information and knowledge since there is no need to establish either a hierarchy or a temporal sequence in discussing data and information. It appears information is a term that is more in vogue these days than data. No one talks about the discipline of data processing any more. Instead we are more likely to use the term 'information technology'.

It is said that information is power and we live in an information society. In some sense, every society, whether developing or developed, is an information society although the more complex a society, the more central information is to its activities. Therefore as a country develops, it uses more information. We can easily understand that information has great value. Let us give some examples to illustrate that value:

- The website Google has nothing but information, and its market capitalization is well over US$100 billion.
- All the websites listed above in Example 1.4 are totally driven by information and they have significant impact on human societies all over the world.
- Warren Buffett, an American investor, has become one of the richest men in the world by just trading information.

There are many other signs of importance and growth of information. The sale of hard disks has been growing exponentially and the price per MByte has been coming down exponentially so much so that a TByte disk in India was selling for below 4,500 INR in January 2011.

Information has therefore become important and an element of commerce. Earlier, success was based on criteria as finance, manufacturing, land, food and so on, but today successful businesses are those that can manage their information. Managing information involves effective collection, organisation, storage, processing and presentation of information. For this reason effective database management and data governance in an enterprise are of great important.

1.6 HISTORY OF DATABASE SOFTWARE

In the early days of computing, computers were mainly used for solving mathematical problems. Even in solving mathematical problems, it was observed that some tasks were common to many problems that were being solved. It therefore was considered desirable to build special software modules that performed frequently occurring computing tasks such as computing Sin(x). This led to the development of mathematical software libraries that are now an integral part of any computer system and most users do not even know that such libraries exist.

By the late 1950s, storage, maintenance and retrieval of non-numeric data had become important. Again, it was observed that some data processing tasks (for example, sorting) occurred quite frequently. This led to the development of *generalized* software modules for some of the most common and frequently carried out tasks.

A general module by its nature tends to be somewhat less efficient than a module designed for a specific problem. In the late fifties and early sixties, when hardware costs were high, use of general modules was not popular since it became a matter of trade-off between hardware and software costs. In the last 30–40 years, hardware costs have been going down dramatically while the costs of building software have not. It is therefore no more a trade-off between hardware and software costs since hardware is relatively cheap and building reliable software is expensive. It therefore makes sense to build generalized software that many enterprises can use.

The earliest known use of the term *database* was in 1963 and the first database management systems were developed in the 1960s. Database management systems evolved from generalized modules for file processing (some of them were called SORT/MERGE packages) as the users demanded more extensive and more flexible facilities for managing increasing amounts of data. This development of database management systems allowed effective storage and retrieval from large databases which was not possible before. Data storage media (disks) were still quite expensive as the amount of data which businesses and governments wanted to process continued to grow. During the next 10–20 years a number of DBMS software packages became available. Most of the early DBMS software were vendor specific, for example, IBM developed a system called IMS (Information Management System) in the mid 1960s which is still being used to process data in some legacy systems. Another early DBMS was called IDMS (Integrated Data Management Store). The data models used by these were the network model based on Bachman's ideas, and the hierarchical model used in the IMS database system. Both IMS and IDMS were non-relational systems. More details about the database models used in these systems is given at the end of Chapter 3. There were other database systems that were

released at the end of 1960s. For example, a DBMS called ADABAS was released by a German company, Software AG in 1970. It was initially released for IBM mainframes. It used an inverted list database.

Database technology has undergone major changes since the early database systems of the 1960s and 1970s which were based on the hierarchical and network models that were developed in the 1960s. With the proposal of the relational database model around 1970, research was started for building relational DBMS at IBM and University of California, Berkeley. Most modern systems are now based on the relational database model which is discussed in Chapter 3. An interesting early history of database systems is presented by Fry and Sibley (1976).

1.7 FILE SYSTEMS VS DATABASE SYSTEMS

There are a number of major differences between a file processing system and a database system. A file processing system often uses a relatively simple data structure to store information about one particular thing. For example, a file may be about players consisting of player records including names, date of birth and nationality while another file may be about matches consisting of match records including date, ground and the teams competing. A file may be using file management software, for example, like the widely used Indexed Sequential Access Method (ISAM) but most file processing systems are implemented for a single application. The same data may be repeated in more than one file and the physical storage structure of the same data may be different for different files if the application programs were written by different programmers.

Therefore in most file management systems, every application carries out its own file processing even if two application programs are carrying out very similar tasks. The programs may involve each user defining and using one or more files that he/she needs. For example, in a university environment, the admissions department may have defined its own files and its application programs may be making use of those files while academic departments may have their own file systems with neither of the departments knowing anything about the other departments' files although the files of both the departments are dealing with students. Such independent maintenance and processing of data not only leads to wastage of computer storage, it also leads to redundancy or replication of data which can further lead to data inconsistencies. Furthermore, the files of the two departments cannot have any inter-relationship among the data stored in their files even if the data is related.

In a database system, no application program accesses the files directly since there is a single repository of data that is defined and maintained by the database system and then accessed by all users. The access to the database is handled by the database management system. When an application program needs to access some data then it must call upon the database system to provide it the data that is required. The database system not only manages all the data, but as already noted, it also contains complete definition or description of the database structure and constraints called metadata.

A database system also allows multiple users to access the same database at the same time. Such use requires concurrency control so that several users trying to update the same data at the same time are controlled. Concurrency control is discussed in Chapter 9. This is just not possible in a conventional file system. Moreover to enforce security, virtual tables called views may be created in database systems and users are restricted to only one or more views of the data that they are allowed to see. This is discussed in Chapter 5.

In a traditional file system, if some changes are made in the structure of one or more files, the changes will affect all the application programs that use them. On the other hand, in case of a database system, the structure of the database is stored separately and therefore changes to data structure do not normally lead to changes in the application programs. This property is known as *data independence* and is discussed later in this chapter in Section 1.13.

Another major difference between file systems and database systems is that a database can represent complex relationships among the information that is stored in the database and can therefore retrieve and update related data easily which is not possible in file processing systems. Finally, there are a number of additional characteristics of a database that are not possible in a file system. These include an ability to define and enforce integrity constraints for the data that is stored in the database. A database also usually provides facilities for backup and recovery. These are discussed in Chapter 11.

In a summary, file processing systems may not be very stable over time because file processing tasks may change over time and these changes may in turn require expensive changes to be made to the files. Furthermore, in file systems, there is a possibility of information being duplicated and this can lead to redundancy and wastage of computer storage which in turn may also result in data inconsistencies. A database system on the other hand usually does not suffer from such problems since the database system does not need to be modified if new data processing is needed. A database may be shared by many users and many applications leading to higher productivity. In addition, it provides facilities for concurrency control, security, integrity and recovery.

1.8 WHAT IS A DBMS?

As discussed earlier, a database is a well organized collection of data. Before the database can be used, the database needs to be stored in the computer in some way so that data from the database may be retrieved when required and modified when necessary. Simple tables like the cricket database given above may even be stored in an Excel spreadsheet file and retrieved, viewed and printed when the need arises. However, when a database is large, for example, an airline's booking database; storing the database in a spreadsheet is not practical since the database is too large and needs to allow many users from different parts of the world to access it at the same time. The users should be able to carry out operations like insertion, deletion and retrieval (and others, for example, sorting). To provide such facilities, any large database needs to be managed by a substantial package of software. This software is commonly called a *Database Management System (DBMS)*. The primary purpose of a DBMS is to allow a user to store, update and retrieve data in abstract terms and thus make it easy to maintain and retrieve information from a database. A DBMS relieves the user from having to know about exact physical representations of data and having to specify detailed algorithms for storing, updating and retrieving data. Without a DBMS, a database would be of little use. With a DBMS, a database becomes a powerful tool to support the operations of an enterprise and to understand and analyse the enterprise.

A database management system may be defined as follows:

> *A Database Management System (DBMS) is a large software package that controls the specification, organization, storage, retrieval and update of data in a database.*

A DBMS carries out many different tasks including the provision of various facilities to enable a user to specify and build a database, and to access and modify data once the database has been built. The DBMS is an intermediate link between the physical database, the computer and the operating system, and on the other hand, the users. To provide different facilities to different types of users, a DBMS normally provides one or more specialized programming languages often called *database languages*. Different DBMS may provide different database languages, but in recent times all DBMS are based on the same abstract model of a database, the relational data model. Given that the underlying model of almost all modern database systems is the same, a language called Structured Query Language or SQL has been declared as a de facto standard. The relational model and SQL is discussed in Chapter 3 and 5 respectively.

Let us consider a simple example of SQL, given in Fig. 1.1, for retrieving data from the database of *bestODIbatsmen*.

```
SELECT Player
FROM bestODIbatsmen
WHERE matches > 400
```

Figure 1.1 Structure of a simple SQL query

This query would retrieve the names of the batsmen that are in the table *bestODIbatsmen* (Table 1.1) and have played more than 400 ODI matches (that is, SR Tendulkar and ST Jayasuriya).

SELECT, FROM and WHERE are reserve words in SQL. Some people do not like the keyword SELECT because what is really meant is either find the names or print the names or retrieve the names but it is the keyword used in SQL. The word FROM must always be followed by the name of the table where the information is to be found (there can be more than one table containing the same information, as we will see later) and WHERE is followed by one or more conditions that the information must satisfy.

One simple way of looking at the way the DBMS processes this query is to think of the software looking at every row in the table in turn and selecting only those rows for which the WHERE condition is true. The DBMS then provides only the values from the columns whose names follow the keyword SELECT for the rows that have been selected.

It should be noted that retrieving data from a structured database like that given in Tables 1.1, 1.2 and 1.3 is quite different than retrieving data from the Web using a search engine, since the Web stores data which is not as well structured as a table.

Since a DBMS is a generalized (large) piece of software, every time a new database is set up using the software, the steps given below must be followed:

1. The database must be defined using the DBMS. Although the simple cricket database in Table 1.1 has only one table, any significant database would have much more information and many tables (more than 100 tables are not uncommon). The structure of the information (for example, number of tables and their columns and each column's type) needs to be specified to the DBMS. This logical structure of the database is often called a *schema* of the database that is to be set up. Obviously some language is required to be able to communicate the structure to the DBMS. A language that allows the database to be described to the DBMS as well as provide facilities for changing the database (for example, adding a column to the table *bestbasmen*; Table 1.1 above may need a column specifying the number of times the player was not out) and for defining and changing physical data structures is called the *data definition language (or DDL)*.
2. The database must be loaded. Put the initial data that is available into the computer using the DDL. Obviously, a database is dynamic and information is added, modified and deleted from it all the time.

3. The database may now be queried. New data can be inserted, deleted and updated, once the data has been stored in the computer. To manipulate, modify and retrieve information, a language for manipulating and retrieving data stored in the DBMS is needed. Such a language is called a *data manipulation language (DML)*.

Each DBMS needs a DDL to define the database and a DML to retrieve and modify information in the database. The two languages, DDL and DML, may be parts of a unified database language as, for example, in SQL which is discussed in Chapter 5. We also note that SQL provides facilities for use in two different ways:

1. *Query Language*—SQL provides powerful facilities to interact with the database. A query language is designed to be simple so that it can be used by nonprogrammers.
2. *Extended Host Language*—Although the SQL query language provides powerful interactive facilities to database users, it is often necessary to retrieve information from a database and process it in a way that cannot be done by the query language. It then becomes necessary for a database system to provide facilities in a standard programming language like Java or C++ to enable the user to interact with the database through software that has been developed in one of these standard programming languages. The programming language that is extended is usually called a *host language*.

To summarize, a database system consists of the items listed in Fig. 1.2.

Some years ago, as noted earlier, DBMS packages marketed by computer manufacturers (for example, IBM) were designed to run only on that manufacturer's machines. The market has changed dramatically since the 1980s and now independent software houses are designing and selling DBMS software (for example, ORACLE or Microsoft SQL Server) that can run on many different types of machines.

- The database (data)
- A DBMS (software)
- A DDL and a DML (part of the DBMS)
- Application programs

Figure 1.2 Components of a database system

1.9 USERS OF A DATABASE SYSTEM

There are several ways of classifying database users. They can be classified by the way the users interact with the system or by their roles in a database system. We will present a top-down approach, starting from the database administrators.

1.9.1 Database Administrators

We do not discuss the role of database administrators here since their role is discussed later in the chapter in detail in Section 1.15.

1.9.2 Database Designers

In large database design projects there are usually two types of designers, viz., logical database designers and physical database designers. The logical designers help in identifying data and relationships between them (as we will see in the next chapter) without taking into account the software that will be used. Once the data and relationships have been identified and modeled, the physical designers look at the ways the database

model can be best mapped to physical storage (for example, hard disks) to deliver the best performance. Issues of physical storage are discussed in Chapter 7 and a number of possible data structures and their characteristics are described. The database designers may need to discuss with the database users which aspects of performance are important to them.

1.9.3 System Analysts and Application Programmers

These highly technical users determine the requirements of end users. Systems analysts develop specifications for application programs that are needed to meet these requirements. Application programmers then implement these specifications as application programs that are appropriately tested, documented and maintained. Such programs need to provide the functionality that is required by end users for whom the programs are being designed. The programmers may also be involved in designing and implementing some DBMS modules, for example, for implementing catalog, interface processors, data access, and interfaces to the software packages.

1.9.4 End Users

End users are a great variety of users that use a database in their work. They use the database for querying, updating, and generating reports. Not all end users are using a database every day, for example, I don't use the library database every day but I do use it regularly. Users like me may be called *casual end users* of the library database. In some cases, a user may be using a database even less frequently than I use the library catalogue. For example, I may check my airline bookings when I have booked a flight but this might happen only 2–3 times a year. Such users may be called *occasional end users*.

The end users normally learn only a few facilities that they use repeatedly. These facilities may include ways to use a database query language to specify their query requests. For example, while accessing the university library catalogue I need to learn the method to search it at an advanced level, reserve a book, renew a loan and browse through some electronic books and journals available in the Monash library.

1.9.5 Naive and Sophisticated Users

Some end users have jobs that require constant querying and updating a database, using standard types of queries and updates (sometime called *canned* transactions) that have been carefully programmed and tested. They do not need to understand the DBMS or the underlying database. Such users may be called *naïve users* and include, for example, a bank teller, a cashier, a policeman, or a nurse. On the other hand, some users are quite sophisticated and are familiar with the DBMS that they are using, including the underlying structure of the database. These users may be called *sophisticated users* and they are likely to be using SQL interactively or in an embedded form to perform their queries. These users include engineers, computer scientists and business analysts who are thoroughly familiar with the facilities of the DBMS and the underlying database and are able to implement its applications to meet their advanced requirements.

1.9.6 Workers Behind the Scene

They are not classified as database users since they typically do not use the database as a part of their job. They include tool developers that develop tools for database design, performance monitoring, graphical interfaces, and test data generation. Another group of users behind the scenes is operators and maintenance personnel as

well as system administration personnel who are responsible for the actual running and maintenance of the hardware and software environment for the database system.

1.10 ADVANTAGES OF USING AN ENTERPRISE DATABASE

A question that often arises in large organizations is why local branches can't keep local information on a local computer using their own favourite software? Some small branches may feel that they do not need to use a DBMS since they have always managed their data using their favourite file processing system software (for example, Filemaker Pro[4]). There are many reasons why using such file processing software and maintaining information locally is often not a good idea.

In a central enterprise database system, the data is managed by the enterprise DBMS and all access to the data is through the DBMS providing a key to effective data processing. This contrasts with local data being stored and processed locally perhaps using different DBMS and without any central control. The local database may be based on considerable local knowledge of data structures being used. Although distributed database technology (discussed in Chapter 13) allows an enterprise to develop a conceptual view of the database even if the database is physically distributed, such distributed databases require considerable central monitoring and control.

In a database system, whether it is distributed or not, the enterprise DBMS provides the interface between the application programs and the data. When changes are made to data representation, the metadata[5] maintained by the DBMS is changed but the DBMS continues to provide data to application programs in the same way as before. The DBMS handles the task of transformation of data whenever necessary.

This independence between the programs and the data, as noted earlier, is data independence. Data independence is important because every time some change needs to be made to the database structure, the application programs that were being used before the change should normally continue to work. To provide

4. FileMaker Pro is a cross-platform file processing software that now allows some database functionality including SQL and web publishing support as well as support for forms and reports.
5. In the context of database, metadata is data about the database, for example, names of tables, number of columns in each of them, their types, number of rows in each table, etc. The term 'meta' comes from a Greek word that means 'alongside, with, after, next'. In addition to database systems, metadata applies to many situations. For example, when an MS Word document is created, it is automatically saved into a file with additional document metadata information that is hidden from the user. This metadata information can hint at authorship, times and dates of revisions and other tidbits. Some of this metadata can be viewed by looking at the properties of the document. Third-party programs are available that will crack open even more metadata hidden in MS Word documents. Another example of metadata is the catalogue in a library. The catalogue is metadata since it is about resources available in the library. In the context of a database, the metadata is stored in the *data dictionary*, which is a term for data describing the database. The database software uses this metadata to interpret and execute SQL queries (as discussed in Chapter 7). Metadata can also be used as a window into the database and to learn about objects and data within the database.

   Web pages also have metadata about them. Web document metadata may include the title, author, and modification date of a web document, copyright and licensing information about the document, its content rating, or the availability schedule for some shared resource. Given the importance of web metadata, a Resource Description Framework (RDF) has been designed to represent it. This metadata can itself be presented in a database.

a high degree of data independence, a DBMS must include a well-designed metadata management system.

When an enterprise uses a DBMS, all enterprise data may be integrated into one system, thus reducing redundancies (redundancy occurs when exactly the same data is stored in more than one place) and making data management more efficient. In addition, the integrated DBMS provides centralized control of the operational data. Some advantages of data independence, integration and centralized control are listed in Fig. 1.3.

- Redundancies and inconsistencies can be reduced
- Better service can be provided to users
- The cost of developing and maintaining systems is low
- Standards can be enforced
- Security can be improved
- Integrity can be improved
- A data model must be developed

Figure 1.3 Advantages of using a database system

These advantages can be elaborated as given below.

- *Redundancies and inconsistencies can be reduced*—Data inconsistencies are often encountered in everyday life. For example, I have come across situations when a new address is communicated to an organization that we deal with (for example, a bank, telephone company, or an electricity company) which results in some communications from that organization being sent to the new address while others continue to be mailed to the old address. Such situations are now less common since combining all the data in a single database results in each piece of information being stored only once. This reduces redundancy. It also reduces inconsistencies, since the enterprise does not have two different addresses anymore. And it reduces the cost of collection, storage and updating of data.
- *Better service can be provided to the users*—A DBMS is often used to support enterprise operations and to provide better service to the enterprise users of the system and to its customers. Once an enterprise establishes a centralized database, the availability of information and the possibility of its revision is likely to improve and the users can obtain new and combined information as soon as it is available. Also, use of a DBMS allows users that do not know programming to interact with the data more easily. The ability to quickly obtain new and combined information is becoming increasingly important for improving services to demanding customers in a competitive environment. Furthermore, various levels of government bodies are now requiring organizations to provide more information about their activities. A centralized DBMS makes it easy to respond to such unforeseen information requests.
- *The cost of developing and maintaining systems is lower*—Since the data and application programs are independent of each other, each of them can usually be changed without affecting the other. This can be particularly important for a website that is supported by a web database. In spite of the large initial cost of setting up a database, normally the overall cost of setting up a database and developing and maintaining application programs, for example, for increasingly important web applications is lower than using earlier technologies.
- *Standards can be enforced*—Since all access to the centralized database has to go through the DBMS, enterprise standards are easier to enforce. Standards may include the naming of the data, the format of the data, and the structure of the data. Furthermore, since all the enterprise data is stored together, data backup may be carried out more regularly and more efficiently.
- *Security can be improved*—Setting up a centralized database makes it easier to enforce security restrictions. Since the data is centralized, it is easier to control who has access to what parts of the database. However, setting up a database can also make it easier for a determined person to breach security. This is discussed in the next section.

- *Integrity can be improved*—Since the data of the organization using a database is often centralized and would be used by many users at the same time, it is essential to enforce appropriate integrity controls. Integrity may be compromised in many ways. For example, someone may make a mistake in data input and the salary of a full-time employee may be entered as Rs. 4,000 rather than Rs. 40,000. A college student may be shown to have borrowed books from the college library but he/she has no enrolment in the college. Salary of a staff member working in one department may be coming out of the budget of another department.

  If a number of users are allowed to update the same data item at the same time, there is a possibility that the result of a number of updates will not be the same as intended. For example, in an airline DBMS we could have a situation where two travel agents sell the same airline seat at the same time and the total number of bookings made is larger than the capacity of the aircraft. In Chapter 9, various ways of avoiding such situations are discussed. However, since all data is stored only once, it is often easier to maintain integrity in modern database systems than in older systems.
- *A data model must be developed*—Perhaps the most important advantage of setting up an enterprise database system is the requirement that an overall enterprise data model be built. If an enterprise were to set up several database systems, it is likely that the overall enterprise view would not be considered. Building an overall view of the enterprise data, although often an expensive and onerous exercise, is usually cost-effective in the long term. It also usually provides flexibility as changes are often necessary as the enterprise changes in response to changing demands of the clients and/or customers.

1.11 CONCERNS WHEN USING AN ENTERPRISE DATABASE

As noted earlier, a centralized enterprise DBMS provides online access to the database for many users concurrently. Because of the large number of users, (assuming that the enterprise is large), who will be accessing the enterprise database, the enterprise may incur additional risks as compared to a more restrictive older style file processing system in the areas listed in Fig. 1.4.

- Enterprise vulnerability may be higher
- Confidentiality, privacy and security may be compromised
- Data quality and integrity may be lower
- Data integrity may be compromised
- The cost of building and using a DBMS can be high
- It may difficult to change the database when necessary

Figure 1.4 Disadvantages of using a DBMS

These issues can be elaborated as given below.

- *Enterprise vulnerability*—Centralizing all the data of an enterprise in one database often means that the database becomes an indispensable resource. The survival of a large enterprise may then depend on reliable information being made available from its database on a 24/7 basis. If the database system goes down for an hour or two, the enterprise may be paralysed while the system is down. At the time of writing this book, the university computer system has been going down frequently due to installation of some new network hardware. When the network has been down, most staff members have not been able to do much work and many have just gone home. In fact an enterprise is not only dependent on its databases, it can even be dependent on information in a search engine like Google. For example,

when there were problems with Google in January 2009, many staff members found it difficult to keep working effectively. The enterprise can also become vulnerable to destruction of the enterprise database or to unauthorized modification of the database.

- *Confidentiality, privacy and security*—When information is centralized and is made available to users from remote locations, it is possible to apply stringent controls on who is able to see what information and who is able to modify what information. However the possibilities of abuse can also increase because some hackers may be able to penetrate the security controls. To reduce the chances of unauthorized users accessing enterprise information, it is necessary to take technical, administrative and possibly legal measures, but an intruder with sufficient resources may still be able to breach database security. A recent University of Cambridge report[6], issued in March 2009, describes how agents of the Chinese government were able to penetrate computers in the office of the Dalai Lama as well as many other computer systems in 103 countries.[7]
- *Data quality*—Since a database provides its users the convenience to access information remotely, adequate controls are needed to control users updating data and thus control data quality and maintain integrity. With an increased number of users accessing data directly, there is an increased risk that data quality and/or integrity will be compromised partly because of input errors. For example, at the time of writing this book, it has been reported that there was an input error in the university salary database which resulted in a person who retired continuing to be paid after the retirement date.
- *The cost of building and using a DBMS*—The database approach provides a flexible approach to data management where new applications can be developed relatively inexpensively. The flexibility is not without its costs, the major cost being that of modeling and building the database, and training or employing personnel. As noted earlier, building a database for a large and complex organization can be hugely expensive but such an organization cannot operate without an effective database system. For example, building a database for an international airline like Air India or a large bank like ICICI that will continue to meet the enterprise needs in the future is a very challenging task.
- *Inability to change the database when necessary*—In a competitive global environment every business has to be able to change quickly to meet new threats from competitors. An airline may merge with another or develop a partnership with a regional airline. A bank may start an insurance business or develop a stock broking business. The database should be such that it can be modified to deal with such major changes to the business. Not all database systems are easily adaptable to new business demands, in particular when a business undergoes a merger or acquisition. If the database system is inflexible, the enterprise may lose its competitive advantage after a dramatic change like a merger or acquisition.

1.12 THREE-LEVEL DBMS ARCHITECTURE AND DATA ABSTRACTION

We now discuss a conceptual framework for a database management system. Several different frameworks have been suggested since the1970s. For example, a framework may be based on the functions that the various

6. S. Nagaraja and R. Anderson, *The Snooping Dragon: Social-Malaware Surveillance of the Tibetan Movement*, Technical Report Number 746, Computer Laboratory, University of Cambridge, UK, March 2009.
7. J. Markoff, *Vast Spy System Loots Computers in 103 Countries*, New York Times, March 28, 2009.

components of a DBMS provides to its users. It may also be based on different users' views of data that are possible within a DBMS. We consider the latter approach.

A commonly used view of data is the three-level architecture suggested by ANSI/SPARC (American National Standards Institute/Standards Planning and Requirements Committee). ANSI/SPARC produced an interim report in 1972 followed by a final report in 1977. These reports proposed an architectural framework for databases. Under this approach, a database is considered as containing data about an *enterprise*. The three-level architecture presented in Fig. 1.5 provides three different views of the data. These views are discussed in some detail below.

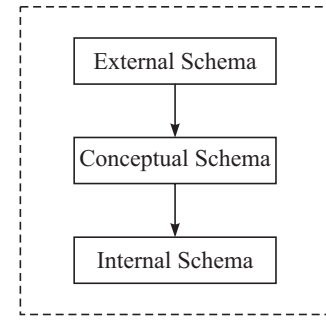

Figure 1.5 ANSI/SPARC DBMS architecture

1.12.1 External View

Each individual user has his/her own view of his/her enterprise database. This view may be a restricted view of the database and the same database is likely to provide a number of different views for different classes of users. In general, the end users and even the application programmers are only interested in a subset of the database if the database is large and complex. For example, a department head in a university may only be interested in the departmental finances and student enrolments in his/her department but not in the buildings and grounds information. The librarian would not be expected to have any interest in the information about departmental finances. The staff payroll office may have no particular interest in student enrolments in computer science unless those numbers have an impact on staff salaries.

1.12.2 Conceptual View

The conceptual view is the information model of the enterprise and contains the view of the whole enterprise without any concern for its physical implementation. This view is usually more stable than the other two views. In a database, it may be desirable to change the way some of the data is stored (internal view) to improve performance while there is no change in the conceptual view of the database. Also, from time to time, a user's view (external view) of the database will change if the user's responsibilities or needs change. The conceptual view is the overall enterprise view of the database and it includes all the information that is of interest and is to be represented in the database. The conceptual view is represented by the conceptual schema which includes details of each of the various types of data.

1.12.3 Internal View

This is the physical or storage view of the database which mostly does not need to be understood by the users. The person managing the database needs to understand this view since the efficiency of the database depends on the way it is stored. The following aspects are considered at this level:

1. What data structure will be used?
2. What access paths do users require to retrieve data, for example, specification of primary and secondary keys, indexes, and sequencing?

3. Miscellaneous, for example, data compression and encryption techniques, and optimization of the internal structures.

Efficiency considerations are the most important at the internal level and the data structures used to represent the database are chosen to provide an efficient database. The internal view does not deal with the physical devices directly. It is only necessary for the internal view to look at a physical device as a collection of physical pages and allocate storage in terms of logical pages.

The separation of the conceptual view from the internal view enables a logical description of the database without the need to specify physical structures. This is usually called *physical data independence*. When we separate the external views from the conceptual view we are able to change the conceptual view without changing the external views. This separation is sometimes called *logical data independence*. Data independence is discussed in more detail in Section 1.13.

Example 1.5

Assuming the three-level view of the database, a number of mappings are needed to enable the users working with one of the external views. For example, the staff payroll office in a university may have an external view of the database that consists of the following information only:

1. Each staff member's employee number, full name and home address.
2. Each staff member's tax information, for example, number of dependants and other relevant information.
3. Each staff member's bank information including the account number where salary is deposited.
4. Each staff member's employment status, salary level and leave information.

We have only listed a sample of information that the payroll office requires. Normally there would be other information including information that deals with payments, provident fund, retirement or pension scheme that an employee participates in.

The conceptual view of the company's database may contain detailed information about the managerial staff, full-time staff, and casual staff as well as the company's financial information and so on. A mapping will have to be created where all the staff members in the different categories are combined into one category for the payroll office. This will need to be mapped to the external view for the payroll office. Also, if there are some changes in the conceptual view, the external view will remain the same even if the mapping is changed.

1.12.4 Data Abstraction

The three-level database architecture allows a clear separation of the overall database view (conceptual view) from the external users' understanding of the database and from the physical data structure layout. A database system that is able to separate the three different views of data is likely to be flexible and adaptable. This flexibility and adaptability is called *data abstraction*. It is a powerful concept because it provides a number of advantages as mentioned below:

1. *Portability*—Portability is important since it is often desirable to be independent of a specific database vendor. The application programs need to be DBMS vendor specific, thus making it easier to replace DBMS software from one vendor to another.

2. *Flexibility and Adaptability*—Flexibility is important since enterprise requirements change according to business conditions and it is important that a database system be able to meet the requirements of new services.
3. *Application Decoupling and Data Independence*—These are important, as discussed earlier, as it is desirable to separate the underlying physical data and the applications. The applications can therefore make use of the data without specific knowledge of the database. This is called *data independence* and is discussed in more detail in the next section.

1.13 DATA INDEPENDENCE

As discussed earlier, data independence deals with independence between the way the data is structured and the programs that manipulate it. Application programs in a database system can continue to work in a certain sense independent of the storage structure of the database. Given data independence, a DBMS may change the structure of the data without having to change the application programs. This is not possible in conventional file processing systems which have a very low level of data independence since the file structure and the application programs are tightly coupled together. Also, no changes can be made to the file structure without making a corresponding change in the application programs. The programs use detailed knowledge of the file structure in manipulating data.

It is possible to have a high level of data independence in database systems only because the application programs do not deal with data in the database directly. Instead the database management software serve data to the application programs in a pre-defined and agreed manner that is independent of the way the data is stored by the system.

The three-level DBMS architecture discussed in the last section illustrates the two types of data independence possible. The first is called *logical data independence*. The second is called *physical data independence*. Logical data independence is the idea of changing the conceptual view, for example, adding a new column in a table, without affecting the external views. Thus it allows logical changes to be made to a database without worrying about the application programs. Physical data independence, on the other hand, is the idea of changing the internal view; for example, changing the data structure used for storing the database, without affecting the conceptual or the external views.

Data independence implies that the application programs should not need to know any of the following:

- The ordering of the data fields in a record
- The ordering of the records in a file
- The size of each record
- The size of each field
- The format and type of each data item
- Whether a file sorted or not
- The type of data structure being used to store some of the data

Data independence does not mean that the user does not need to know:

1. The names of files in which the relevant data is to be found
2. The names of the data fields that the user wishes to access

As noted earlier, the level of data independence of a system depends on the sophistication of metadata management in the system. A system with a sophisticated metadata management will involve changing the metadata when the physical data structure of some data is changed. Once the change is made, all the database application programs should still be able to run perfectly and find the data they use.

1.14 DBMS SYSTEM ARCHITECTURE

As noted earlier, a database management system is a complex piece of software that consists of a number of modules. The DBMS may be considered as an agent that allows communication between different types of users with the physical database and the operating system without the users being aware of every detail of how it is done.

In addition to providing facilities for defining the database and retrieving information, a DBMS provides facilities for the following:

- User communication with the system
- Authentication and authorization
- Query processing including a syntax checker and translator for the DDL and DML
- Access path selection
- Transaction management including concurrency control, logging manager, locking manager and recovery manager
- Database management including a file manager and the physical database
- Catalog manager

To provide all these facilities and some others, a DBMS has system architecture like the simple architecture shown in Fig. 1.6.

The various components of the database system architecture are discussed below.

1. **User Communication, Authentication and Authorization**
   This component deals with the user when he/she communicates with the DBMS. The communication depends on the kind of database system. It may be a two-tier system or a three-tier system. In each case, the system would include a mechanism for the user to log in and be authenticated. Once the user is allowed to access the database, the system will check the user's credentials and allow the user to access the parts of the database that he/she is authorized to.
2. **Query Processor**
   This component of a DBMS has several subcomponents including a query parser, a query rewriter, a query optimizer and a query executor. Query processor involves the DBMS interpreting the DDL and DML statements from the users and checking them for correct syntax and, if found to be correct, translate them into query execution plans. While developing query execution plans, the query optimizer attempts to efficiently find the best plan. These plans are then passed on to the database manager which then executes them. This is perhaps the most important component of a DBMS since each user is querying the database and is hoping to get results quickly. The queries are mostly submitted in a declarative language SQL and therefore its translation and optimization is very different than that of

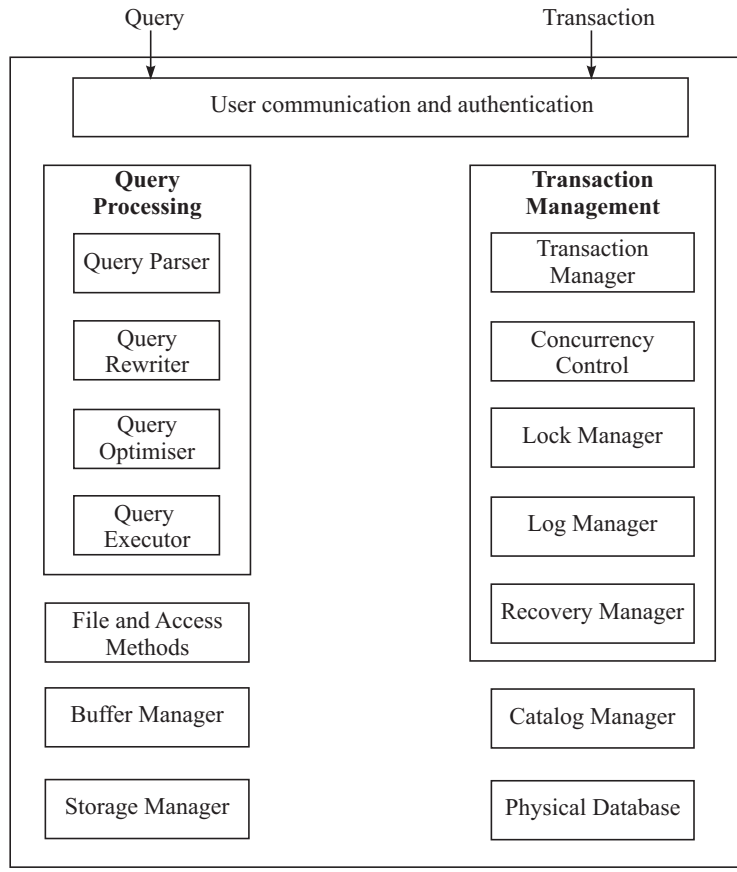

Figure 1.6 System architecture of a DBMS

procedural language programs. Query processing including details about the various components of a query processor are discussed in Chapter 7.

In a nonprocedural or a declarative language like SQL, no sequence of operations is explicitly given and therefore a query may be processed in a number of different ways, often called *plans*, where each plan might have a different sequence of operations. Optimization in declarative languages therefore is a more complex task, since it not only involves code optimization (ways to carry out operations that need to be carried out) but also selecting the best plan including selecting the best access paths. In many situations, especially if the database is large and the query is complex, a very large number of alternative plans are often possible and it is then not practical to enumerate them all and select the best. Then, it is necessary to consider a small number of possible plans and select the best option from those. Since the savings from query optimization can be substantial, it is acceptable that a DBMS spend some time in finding the best plan to process the query. Again, of course, we assume that we are dealing with queries that are expected to consume a significant amount of computing resources. There is no point in spending time optimizing queries that are so simple that almost any approach could be used to execute them in a small amount of time.

Further details of query processing are provided in Chapter 8.

3. **Transaction Management and Concurrency Control**

Modern database systems are multi-user systems. Some large systems may support thousands of users. Multiple users are able to use a single system simultaneously because the processor (or processors) in the system is shared amongst the users trying to access computer resources (including databases) simultaneously. The concurrent execution of programs is therefore interleaved, with each program being allowed access to the CPU at regular intervals. The concept of a transaction is very important since many users are using a multi-user system concurrently and the transactions must be managed so that they do not interfere with each other.

Control is therefore needed to coordinate concurrent accesses to a DBMS so that the overall correctness of the database is maintained. Efficiency is also important since the response time for interactive users ought to be short.

Our discussion of concurrency in Chapter 9 will be transaction based. A transaction is assumed to be a logical unit of work, and a unit of consistency in that preserves database consistency. Transaction processing database systems are now ubiquitous. For example, they include e-commerce, billing systems including telecommunications accounting, payroll, airline and hotel reservations, banking and electronic stock market transactions, to name a few.

Transaction management consists of a number of components including the following:

- Concurrency control
- A lock manager
- A log manager
- Recovery manager

Transaction management and concurrency control, backup and recovery, security and integrity are discussed in Chapters 9, 10, 11 and 12 respectively.

4. **File and Access Methods**

This component of the DBMS interacts with the physical storage that stores the database. The file manager is responsible for storing the database on the disk and for providing the DBA a number of data structure options for storing the database. This module also assists in maintaining the metadata of the database as well as indexes for the various files. Further details are provided in Chapter 7.

5. **Buffer Manager**

A database is normally resident on a disk although data can only be manipulated in the main memory. A part of the database therefore must be loaded in the main memory for retrieval and updates, and it must be written back to the disk once it is modified. The database therefore has a buffer manager that is responsible for handling the interfacing between the main memory and disk. The buffer manager maintains a number of pages of the same size and a large buffer size is important for the performance of a large and active database system. Further details are provided in Section 7.4.

6. **Storage Manager**

The storage manager is a DBMS subsystem that provides support for database management including management of access paths and indexes.

7. **Catalog Manager**

We have discussed earlier the role of metadata to describe the data structures (tables and their properties, users, indexes, constraints and so on) in a database. This data is maintained in the catalog and stored

as relational tables. This allows the catalog to be queried in the same way as the rest of the document is queried. Further details about metadata are provided in Section 8.4.

8. **Physical Database**
   This is the physical storage of the database, normally on a disk. The database tables of the physical database are stored using a number of different file structures and indexes as discussed in Chapter 7. The performance of a large database depends critically on how efficiently the data has been structured on the disk.

The article by Hellerstein et al., provides a detailed discussion of database system architecture.

1.15 DATABASE ADMINISTRATOR

The database can meet the demands of a variety of users in the organization effectively only if it is maintained and managed properly. Usually a person (or a group of persons) centrally located, with an overall view of the database, is needed to keep the database running smoothly. Such a person is often called the *Database Administrator* (*DBA*). In a large enterprise, a DBA should be assisted by an advisory committee that includes members from relevant units of the company.

A DBA would normally understand the following:

- What data is in the database and where did it come from?
- What does the data in the database mean?
- How reliable, accurate, and authoritative is the data?
- If all enterprise applications that need the data are able to use it?
- Who ultimately governs the definition, lifecycle, and use of data in the database?

Given this understanding of the data, the DBA normally has a large number of tasks related to maintaining and managing the database. These tasks would generally include the tasks listed in Fig. 1.7.

These tasks can be elaborated as follows:

1. *Developing the conceptual schema*—The DBA in consultation with senior management of the enterprise (perhaps assisted by external consultants) may be responsible for defining the conceptual schema of the database. In some cases this task may be the responsibility of the Chief Information Officer (CIO) of the enterprise, if the enterprise has one. It may be necessary to make changes to the

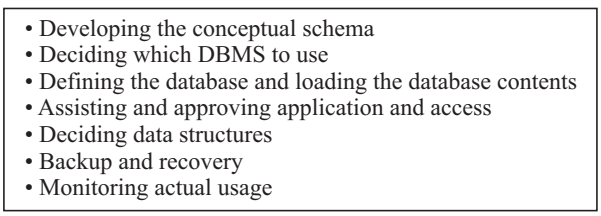

Figure 1.7 List of tasks handled by a DBA

conceptual schema of the database periodically. The person who had responsibility for developing the initial conceptual schema should normally also be involved in making changes to it.

2. *Deciding which DBMS to use*—Once the conceptual schema has been decided, a decision regarding the DBMS to be used needs to be made. In some cases, the enterprise would have made this decision already, perhaps based on previous experience of using a DBMS within the enterprise. In some cases it may be appropriate to employ a database consultant to help in selecting a DBMS.
3. *Defining the database and loading the database contents*—Once the DBMS has been acquired and installed, the DBA would be responsible for implementing the conceptual schema by defining the conceptual schema using the DDL, and then loading the data the enterprise initially wishes to load in the database. Loading the database often is an incremental process.
4. *Assisting and approving applications and access*—The DBA would normally provide assistance to end users as well as programmers interested in writing application programs to access the database. The DBA would also approve or revoke access to the various parts of the database by different users.
5. *Deciding data structures*—Once the database contents have been decided, the DBA would normally take decisions regarding the manner in which data is to be stored and the indexes that need to be maintained. In addition, a DBA normally monitors the performance of the DBMS and makes changes to data structures if the performance justifies them. In some cases, radical changes to the data structures may be called for. Physical data structures are discussed in Chapter 6.
6. *Backup and recovery*—Since the database is such a valuable asset, the DBA must make all possible efforts to ensure that the asset is not damaged or lost. This normally requires a DBA to ensure that regular backups of a database are carried out and stored off-site and that in case of failure (or some other disaster like fire or flood) suitable recovery procedures can be used to reassemble the database with as little down time as possible. Recovery and backing techniques are discussed in Chapter 9.
7. *Monitoring actual usage*—The DBA monitors actual usage to ensure that policies laid down regarding use of the database are being followed. The usage information is also used for performance tuning.

There are many other tasks that a DBA often has to handle. These tasks may involve diagnosing, troubleshooting and resolving any database related problems. The DBA also deals with database security in addition to managing user identification, authentication and authorization.

1.16 DESIGNING AN ENTERPRISE DATABASE SYSTEM

As noted earlier, an enterprise database system is a complex and large software project. In Section 1.8 we noted ways in which a DBMS can be populated. Building a new database is just as complex as any large software project. Building a new database system usually goes through the phases shown in Fig. 1.8, just like the software development life cycle (SDLC).

These phases are discussed briefly. All design processes are usually iterative processes. For example, the second phase may discover some inconsistencies of the first phase which may need to be fixed before proceeding further.

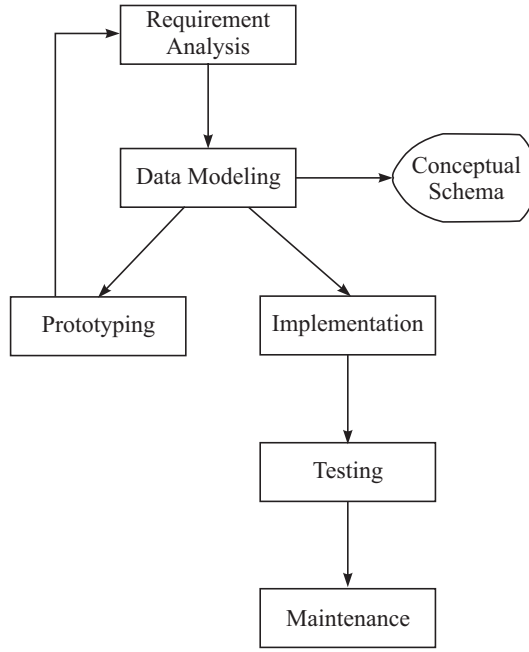

Figure 1.8 Database Design Lifecycle

1.16.1 Requirements Analysis

Each large software project involves requirement analysis to ensure that the software to be developed meets the requirements of the enterprise (or the client). For a database system, requirement analysis involves fully understanding the requirements of the proposed system and describing them precisely, including the queries that are likely to be posed. These requirements must be explored in-depth ensuring that a clear and concise description of the requirements is prepared. The description should include all the properties of the objects of interest, ways they are related to other objects and frequency of access. Nature and volume of the transactions should also be explored and documented. Furthermore, the possibility of growth of the data in future should also be considered. This can be a very challenging task.

A requirements specification document should be prepared based on all this information. This document must be accepted and signed by both parties, viz., the database designer and the client. Once signed, the document is then used in all design phases that follow.

1.16.2 Data Modeling

Once the first phase has been completed, the information content of the proposed database system is known and the requirements specification document is available, the information must be modeled using a data modeling technique. This is the most important phase of the database design because the quality of the database system and its adaptability is often significantly influenced by the quality of the data model. This

is elaborated in the next chapter using a relatively simple modeling approach called the Entity-Relationship model. The primary aim of data modeling is to develop an enterprise view of the proposed database system with the ultimate aim of achieving an efficient database that is suitable for the workload and meets all the requirements. To some extent data modeling and requirements analysis go hand-in-hand because data modeling can also assist in communications between the designer and the client because of simplicity of its representation.

1.16.3 Prototyping

Prototyping can be a very helpful phase of the design life cycle because it helps enormously in improving communication between the designer and the client. A prototype should be a quick and dirty implementation of a simplified proposed database in order to confirm the specifications developed in the requirements phase.

1.16.4 Implementation

Given the data model developed in the data modeling phase which we assume is independent of the DBMS to be used, the model needs to be transformed using the DBMS that the enterprise has selected for the database. If the data modeling exercise of the last phase was successful then the implementation should also be good. We assume that the system is based on a relational database model that is discussed in Chapter 3. A suitable set of software tools should be available for users to enable them to use the system effectively.

1.16.5 Testing

Just like for any software system, a database system must also be tested to ensure that the system is reliable, efficient and correct, and meets the demands specified in the requirements document. The testing phase should discover most of the errors that may have arisen during the modeling and implementation phases. Testing may also be used to show to the client how the system is likely to appear and perform.

1.16.6 Maintenance

In spite of thorough testing, any new database system is likely to have errors and omissions and these need to be fixed during the maintenance phase. In addition, on using the system the client will almost always come up with new demands that require modifications in the design and implementation of the system. Even if no new demands are made, the client might find that some database queries or transactions are not performing well and need performance improvement. The database administrator is of course in-charge of resolving all such client requests. Maintenance of a system can become very expensive unless the system is well designed.

1.17 FUTURE OF DATABASES—WEB DATABASES

Web databases are the future of database technology. These are by far the most useful and versatile database applications currently available and most likely to be more important in the future. Most web applications

involve the client requesting a web server for some information, which the web server usually finds by searching one of its databases and then serves it to the client. A variety of technologies are being used for these database interactions including CGI programs and scripts. Each copy of a CGI program connects to a web database server, which controls all requests to the databases, then returns any data retrieved to the CGI program which in turn is returned to the client.

In a typical client-server application, the user's workstation contains the software that provides the graphical user interface (for example, the browser). The server includes the database and a program to manage read and write access to it. Each module of a client-server may be developed concurrently by different teams of programmers and the database on the server can be easily moved to another server if this becomes necessary.

There are other types of databases on the Web. For example, Google has a very large database called the Bigtable, which is a distributed database system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of servers. Many projects at Google store data in a file system called Bigtable, including web indexing, Google Earth, and Google Finance. These different applications place different demands on Bigtable. Despite these different demands, Bigtable is being used successfully to provide a flexible, high-performance solution for all of the Google products.

1.18 CHOOSING A DBMS

Choosing any major piece of software is a difficult task. Certainly choosing a DBMS for a large enterprise is a challenging task since the software could even cost one crore rupees or more. Some basic information about various DBMS available in the market at the time of writing this book has been included. Oracle is the largest vendor of DBMS software. Other big vendors are Microsoft and IBM. Some DBMS packages from them as well as other vendors in the market are listed in Fig. 1.9.

- Oracle
- Microsoft Office Access
- Microsoft SQL Server 2008
- Ingres
- IBM DB2
- Sybase
- MySQL

Figure 1.9 List of popular DBMS software

Microsoft Access is part of the Microsoft Office suite of software. It is a nice implementation of relational database. For simple queries one may use QBE, Query by Example, a graphical query language which is very different from SQL. It is an easy-to-learn query language. Microsoft SQL Server 2008 is a major Microsoft database product. SQL Server 2008 has been reported to have many enhancements in comparison with its previous version. MySQL is an open source database management system software.

In selecting DBMS software, it is important to consider whether the system is to be used for business or personal use. If it is to be used for business, the size of the business, the size of the database, the number of users and response time required are the kind of issues that may need to be considered. Also, one may need to know the budget the enterprise is willing to spend and the amount of software support that will be required?

While selecting the software, in addition to the actual data management capabilities of the DBMS, it is important to consider the extent of the development environment supported by the DBMS, since the development environment supported will have a significant effect on the ability to develop new applications and to port and maintain existing applications.

Support for the standard query language SQL is now common for all DBMS although some DBMSs have added functionality not supported by the standard SQL. The added functionality can be useful but the database staff will need training in the additional features. Furthermore, using vendor-dependent features can lock the enterprise in continuing to use the same vendor which may not be desirable.

If the enterprise is large, it might be possible to undertake evaluation based on clearly identified requirements. In some cases using a database consultant might make sense. But if the organization is of small or medium size then it is crucial that the vendor selected shows a great deal of empathy and support. Another issue that also must be considered seriously is the ease of application development (or ease of finding third party applications developers). Yet another issue that may be important in some situations is how fast can the applications that are urgently needed be developed? How easy is it going to be for the employees to use? These are often difficult nontechnical issues.

For personal use, efficiency issues may not be so important since the database is likely to be small. Something like Microsoft Access might be suitable.

Some rules of thumb in selecting DBMS software for a large enterprise are as follows:

1. *Focus on effectiveness not efficiency*—Efficiency is always important, especially if the database is being used in a real-time environment where the response time is crucial. Often though one wants a system that is effective, i.e., it gets the job done satisfactorily.
2. *Focus on good end user tools*—Since a database is primarily used by the enterprise users, the software must provide tools that can provide a good user interface.
3. *Focus on good application development tools*—Most database systems require at least some application programs that need to be developed in-house. Therefore, it is important that tools be available for developing such programs.
4. *Focus on availability of third party developers*.
5. *Focus on availability of a user group support for the DBMS*.

Most DBMS products provide a 4GL (fourth-generation language) to assist in application development but the quality of the 4GL varies and may need to be evaluated.

SUMMARY

In this introductory chapter, we have covered the following topics:

- The amount of data various enterprises are collecting is growing because of increasing use of OLTP systems as well the World Wide Web.
- Database management is about management of information of an enterprise.
- A database is well organized collection of information.
- The information in a database is related, stored only once and is shared.
- Very large databases are being maintained by a number of enterprises, for example, by search engines and social networking sites.
- Database approach is different than file processing approach because file processing provides little data independence.

- A DBMS is a large software package designed to manage information.
- A DBMS has many modules including query processor, concurrency control and file manager.
- A database system has many different types of users, for example, end users, naive users and sophisticated users.
- There are a number of advantages and concerns when an enterprise database system is used, for example, reduction of redundancy and database security.
- A database system may be looked as three-level architecture; the three levels are external, conceptual and internal.
- Importance of concepts of data abstraction and data independence has been explained.
- A simple architecture of DBMS has been presented. It has many modules including query processor, concurrency control and recovery manager.
- The database administrator is responsible for managing the database system.
- The responsibilities of a DBA are described including database design, data loading and database monitoring.
- Developing a database system is similar to developing any major software since it includes requirements analysis, design and testing steps.
- Web databases are becoming increasingly important because of widespread use of the World Wide Web.
- It is important to remember a number of factors when buying a DBMS including the importance of DBMS effectiveness and database tools.

REVIEW QUESTIONS

1. Give an example of a simple database related to information that you use regularly. (Section 1.2)
2. List names of some large Indian websites that collect huge amount of information. (Section 1.3)
3. What is a database? Discuss its main features and explain the importance of each feature. (Section 1.3)
4. List the characteristics of databases. (Section 1.4)
5. What is data governance and why do enterprises require it? (Section 1.5)
6. When did database management become important in computing? When was the relational database system invented? (Section 1.6)
7. Describe the disadvantages of using a file processing system compared to a database system? (Section 1.7)
8. What is a DBMS? What tasks does a DBMS carry out? (Section 1.8)
9. What classes of users use database systems? (Section 1.9)
10. Discuss the advantages of using the DBMS approach as compared to using a conventional file system. (Section 1.10)
11. Discuss the concerns in using the DBMS approach as compared to using a conventional file system. (Section 1.11)
12. List the three views for a database system in the three-level architecture. Describe the role of these views. (Section 1.12)

13. Explain data abstraction and data independence. (Section 1.12 and 1.13)
14. List the major components of a DBMS architecture and describe their functions. (Section 1.14)
15. Describe the role of a database administrator. (Section 1.15)
16. What are the phases in developing an enterprise database system? (Section 1.16)
17. Briefly describe the role of database technology for the Web. (Section 1.17)
18. List three major considerations when buying a DBMS for an enterprise. (Section 1.18)

SHORT ANSWER QUESTIONS

The following questions are based on the assumption that a database is being shared amongst several users.

1. Name one large enterprise in your city which could not operate effectively without a database system.
2. Name one large website that you use regularly. Estimate its database in terms of number of people that use it and information about each of these users?
3. Why database systems are becoming important?
4. When were the first database management systems developed?
5. Name three popularly used DBMS.
6. List three major differences between file systems and database systems.
7. Which is the largest database company in the world?
8. How many mobile phone calls are made and SMSs are sent in India every day? Make an estimate based on some simple assumptions about the number of mobile phones in India.
9. List the major classes of nontechnical users that use a database system.
10. Who are the workers behind the scene in a database system?
11. List three benefits using an enterprise database system, compared to a file processing system.
12. List three areas of concern in using an enterprise database system.
13. What are the three levels in a three-level database architecture.
14. What is data abstraction?
15. What are the benefits of data independence?
16. List three components of query processing.
17. List three components of transaction management.
18. What does the buffer manager do?
19. What is metadata and catalog?
20. Why does a DBMS require a concurrency control module?
21. List three duties of a DBA.
22. What is the first step in designing an enterprise database system?

MULTIPLE CHOICE QUESTIONS

1. Which one of the following should be the main focus of an enterprise database?
   (a) Enterprise computers
   (b) Enterprise information
   (c) Enterprise computer network
   (d) Enterprise management
2. Which one of the following is a problem with the single table database presented at the beginning of this chapter?
   (a) It is difficult to find if a player is still playing.
   (b) A number of columns are related.
   (c) It assumes that each player plays only for one country during his lifetime.
   (d) All of the above
3. Which one of the following is the most accurate definition of a database?
   (a) A computerized record-keeping system.
   (b) A centralized collection of enterprise data.
   (c) Software designed to access and manipulate data.
   (d) None of the above
4. Due to which one of the following reason have databases become so important for all large enterprises?
   (a) Hard disks are becoming cheap.
   (b) Database technology is a mature technology.
   (c) Oracle and IBM have good database software.
   (d) The quantity of information and its importance is growing.
5. A DBMS provides a variety of facilities, some of which are listed below. Which one should not be on this list?
   (a) Data definition and manipulation
   (b) Data recovery and concurrency control
   (c) Data security and recovery
   (d) Data modeling
6. A database has a number of features. Which of the following are features of a database?
   (a) It is centralized.
   (b) It is well organized.
   (c) It is related.
   (d) It is accessible in different orders without great difficulty.
   (e) All of the above
7. Consider the following schemas of the tables:
   ODIbestbat (Player, Span, Runs, Average, Strike Rate)
   Testbestbat (Player, Span, Runs, Average, Centuries)
   ODIbestbowl (Player, Span, Runs, Wickets, Average, Economy Rate)
   Testbestbowl (Player, Span, Runs, Average, Economy rate)
   Player (Name, Country, Place of Birth, Date of Birth)
   ODImatch (Date, Host, Visitor, Result)

Testmatch (Date, Host, Visitor, Result)
Testranking (Country, Points)
ODIranking (Country, Points)

Which one of the following together satisfy the properties that a database should possess?

(a) ODIbestbat and ODIranking
(b) ODIbestbat and ODImatch
(c) ODIbestbat and Testbestbat
(d) Player, ODIranking
(e) None of the above

8. There are many advantages of using a database approach in contrast to using older data processing techniques. Which one of the following is **not** true if the database approach is used?
   (a) Data redundancy can be reduced.
   (b) Data inconsistency can be avoided to some extent.
   (c) Computing resources needed for data processing can be reduced.
   (d) Data processing standards can be enforced.
9. Which one of the following is **not** necessarily a characteristic of a database system?
   (a) Self-describing nature
   (b) Data independence
   (c) Data accuracy
   (d) Support for multiple views of the data
10. Which of the following should a good DBMS used for a large multi-user enterprise provide?
   (a) Control redundancy
   (b) Restrict unauthorized access
   (c) Enforce integrity constraints
   (d) Provide backup and recovery
   (e) All of the above
11. It is claimed that data independence is a major objective of database systems. Which one of the following statements about data independence is **not** true?
   (a) Data independence is the immunity of applications to change in physical storage structure and access strategy.
   (b) Data independence leads to difficulties in integrating data and eliminating redundancies.
   (c) Data independence makes it possible for different applications to have different views of the same data.
   (d) Data independence is necessary to enable the database to grow without affecting existing applications.
12. The ANSI/SPARC model of DBMS architecture comprise three levels, viz., internal (or physical), external and conceptual. Which of the following is **not** correct?
   (a) The internal view of a database deals with physical records and device specific considerations such as track and sector sizes.
   (b) The conceptual view of a database is the community view of the database, i.e., a view of the database *as it really is*.
   (c) The external view of a database is the individual user view, i.e., often this is a restricted view of the conceptual view.
   (d) There can be only one external view, only one conceptual view and only one internal view.

13. As a result of data independence, which one of the following is **not** true?
    (a) The user does not need to know the size of each file record.
    (b) The user does not need to know if the file is sorted.
    (c) The user does not need to know the name of the file that has the relevant information.
    (d) The user does not need to know the ordering of the fields in the records.
14. Which one of the following is a major difference between a file processing system and a database system?
    (a) File processing system files may be main memory resident while database is disk resident.
    (b) There is no data independence in file-processing systems.
    (c) File systems are able to represent relationships between various data while databases cannot.
    (d) File processing system can be used by many users.
15. Which of the following are important components of the DBMS architecture?
    (a) Query optimizer
    (b) Database manager
    (c) File manager
    (d) Physical database
    (e) All of the above are important
16. When choosing a DBMS which of the following are important?
    (a) Effectiveness
    (b) Efficiency
    (c) The name of the company
    (d) Good end user tools
    (e) All of the above are important

EXERCISES

1. Discuss the importance of information for modern enterprises. Give some examples to illustrate your answer.
2. Explain the relationship between an enterprise, its data processing needs and its database.
3. Why should you choose a database system instead of simply storing data in a spreadsheet? When would it make sense not to use a database system?
4. What is logical data independence and why is it important? Give a practical example when logical data independence is important.
5. What is meant by logical and physical structure of data? Explain the difference between logical and physical data independence.
6. Explain the difference between external view, internal view, and conceptual view in three-tier database architecture. How are these different schema layers related to concepts of logical and physical data independence?
7. Which of the following plays an important role in representing information about the real world in a database? Explain briefly.
    - Data definition language
    - Data manipulation language
8. Would a DBMS designed for use on a PC by a single user also have the same components as shown in Fig. 1.6? Explain your answer.

9. Discuss the responsibilities of a DBA. Explain what would happen if a DBA was not responsible for an enterprise database system? Why does the DBA need to understand query optimization? Explain the role of a DBA in a simple database like the ODI database if it is being used by many people.
10. Could a modern university be supported if a file processing system was used to manage enterprise information? List three major difficulties that are likely to arise if no database system was available to the university. Discuss the difficulties.
11. Study the role of data governance by looking at some recent literature on data governance. What benefits can be obtained by an enterprise using data governance which they cannot get with current database systems? Discuss.
12. Select an enterprise that you are familiar with (for example, a school, college, company, small business, club or association). List all the information that this enterprise uses.
13. Give examples to illustrate three major advantages and three areas of concern when using an enterprise database system. Discuss their importance.
14. List five queries you might wish to pose to the ODI database given in Tables 1.1, 1.2 and 1.3.
15. Why is data independence important? Discuss its importance by giving an example.
16. Present a conceptual view of a student database. Present some external views given by different types of end users and show the mappings of these views to the conceptual view.

PROJECTS

1. If you have access to the Indian Railways Timetable, describe how you would computerize it and how much memory space the computerized timetable would take? What other information would be useful in the timetable?
2. Select a local company large enough to have a lot of information about its operations. Arrange to meet someone in that company who can tell you about the information they have and the ways they manage and use it. Write a brief report on the same.

LAB EXERCISES

1. Northern India University (NIU) is a large institution with several campuses scattered across north India. Academically, the university is divided into a number of faculties, such as Faculty of Arts and Faculty of Science. Some faculties operate on a number of campuses. Faculties, in turn, are divided into schools. For example, the School of Physics, the School of Information Technology, etc. The schools are not divided further into departments.

   Each student in the University is enrolled in a single course of study which involves a fixed core of subjects specific to that course as well as a number of electives taken from other courses. Each course is offered by one particular school. Each school has a number of lecturing staff. Students are awarded a grade for each subject they chose.

   List all the information that you need to store in a database for NIU.

2. List all the information that you might require if you were in charge of a second, small airport in Mumbai that mainly handles private planes. The database is to be designed to keep track of information about airplanes, their owners, and pilots. Each plane is stored in a particular hangar which has a number and some other information about it. You will need to carefully think about the information required. A person who knows more about small private planes might be able to assist you.

BIBLIOGRAPHY

There are many good books and references on database management. The article by Fry and Sibley presents an interesting historical account of early DBMS developments. The other two articles, by Jardine and by Tsichritzis and Klug, deal with the ANSI/SPARC model. We have listed a collection of 16 database books.

For Students

The books by Churcher, Date, Stephens and Takahashi et al., are good introductory books for database management. Each of them has a different style. The book by Date has been around for a long time and is now in its eighth edition. It is simple and easy to read. The books by Pascal, Chao and Churcher are also easy to read. The book by Takahashi et al., is a translation of a popular Japanese book which includes many illustrations. The two books by Robert Glass describe some actual software disasters including some that involved database systems.

For Instructors

A comprehensive introduction to database management is provided in the book written by Elmasri and Navathe, which deals with all the topics covered in this book. The book by Ramakrishnan and Gehrke is a slightly more advanced introduction. The book by Silberchatz, Korth and Sudarshan is a comprehensive text. The books by Garcia-Molina et al., and Ullman and Widom are comprehensive introductions to database management. The article by Hellerstein, Stonebraker and Hamilton is a 120-page long article on architecture of database systems for those who want to learn more about this topic.

Chao, L., *Database Development and Management (Foundations of Database Design)*, Auerbach, 2006.

Churcher, C., *Beginning Database Design: From Novice to Professional*, APress, 2007.

Date, C. J., *An Introduction to Database Systems*, Eighth Edition, Addison-Wesley, 2003.

Elmasri, R., and S. B. Navathe, *Fundamentals of Database Systems*, Fifth Edition, Addison-Wesley, 2006.

Fry, J. P., and E. H. Sibley, "Evolution of Database Management Systems", *ACM Computing Surveys*, Vol. 8, pp 7- 42, March 1976.

Glass, R. L., Software *Runaways: Monumental Software Disasters*, Prentice Hall, 1997.

Glass, R. L., *Computingfailure.com: War Stories from the Electronic Revolution*, Prentice Hall, 2001.

Garcia-Molina, H., J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, Second Edition, Prentice Hall, 2008.

Garcia-Molina, H., J. D. Ullman, and J. Widom, *Database System Implementation*, Prentice Hall, 2000.

Hellerstein, J. H., M. Stonebraker, and J. Hamilton, "Architecture of a Database System", *Foundations and Trends in Databases*, Vol. 1, No. 2, 2007, pp 141-259. (May be downloaded from http://www.db.cs.berkeley.edu/papers/fntdb07-architecture.pdf (120 pgs))

Jardine, D. A. (Ed.), *The ANSI/SPARC DBMS Model*, North-Holland, 1977.

Kifer, M., A. Bernstein, and P. M. Lewis, *Database Systems: An Application-Oriented Approach*, Second Edition, Addison-Wesley, 2006.

Lyman, P., and H. R. Varian, "How Much Information*?"*, *School of Information Management and Systems*, *University of California*, *Berkeley*, 2003. http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/

Pascal, F., *Practical Issues in Database Management: A Reference for the Thinking Practitioner*, Addison-Wesley, 2000.

Ramakrishnan R., and J. Gehrke, *Database Management Systems*, Third Edition, McGraw-Hill, 2002.

Silberchatz, A., H. Korth, and S. Sudarshan, *Database Systems Concepts*, Fifth Edition. McGraw-Hill, 2005.

Stephens, R., *Beginning Database Design Solutions*, Wiley, 2008.

Takahashi, M., S. Azuma, and Trend-pro Ltd., *The Manga Guide to Databases*, No Starch Press, 2009.

Tsichritzis D., and A. Klug (Eds.), "The ANSI/X3/SPARC Framework", *Information Systems*, Vol. 3, No. 3, 1978.

Ullman, J. D., and J. Widom, *First Course in Database Systems*, Third Edition, Prentice Hall, 2007.

CHAPTER 2

Entity–Relationship Data Model

OBJECTIVES

- Discuss the importance of data modeling.
- Describe the different phases of database modeling.
- Explain the Entity-Relationship model for modeling a database.
- Define entity, entity sets, relationship, relationship sets, different types of relationships, and attributes.
- Describe representing entities, attributes and relationships in entity-relationship diagrams.
- Explain binary and ternary relationships.
- Discuss the concepts of generalization, specialization and aggregation.
- Describe the extended Entity-Relationship model.
- Present guidelines for building an Entity-Relationship model.
- Analyze the strengths and limitations of the Entity-Relationship model.
- Present a case study of developing an Entity-Relationship model.
- Explain evaluation criteria for different Entity-Relationship database models.

KEYWORDS

Modeling, database design, Entity-Relationship model, E-R model, entity, entity set, domain, attributes, diagrams, single-valued and composite attributes, relationship, one-to-one, one-to-many, many-to-many, binary relationship, weak entities, generalization, aggregation, specialization, E-R diagrams, extended Entity-Relationship model, EER, cardinality ratio, superkey, supertype, subtype, attribute inheritance, model evaluation.

Aag Rameshwari Aanee Bamb Someshwari.
Translation: The problem is one thing and the solution is given for something else.

Marathi proverb

2.1 INTRODUCTION

As noted in the last chapter, building a database system is a complex process that normally requires analyzing the user's requirements followed by a design process and finally concluding with an implementation. During analysis and design, the database designer normally needs to build a model of the proposed database system. Database modeling is essential to adequately manage the database development process.

Models have not been invented recently. Humans have used models for a long time for a variety of purposes and perhaps that is why modeling is interpreted differently by different people. Consider some examples of model building:

1. Architects build models of buildings to show what a building or a house would look like. Landscape architects build models of landscaping to show what a landscape or garden will look like.
2. Aeronautical engineers build models of planes to show what a plane would look like. Scaled models may also be used in testing to evaluate a plane's performance.
3. Computer architects build models of new computers or new circuit boards that are implemented in hardware.
4. Traffic engineers build models of traffic for a city or part of a city to show why there are traffic problems or how they could be solved.

The reader may think of other applications. Building models is common in science, engineering and computing. A model is an abstract (and often incomplete) representation of the design or definition of a complex product, process or situation (for example, information requirements of an enterprise). A model need not be related to one particular situation; it may model a class of situations. For example, a database model may model all kinds of libraries but often each new situation requires a customized model. A model may describe, at different levels of abstraction, an organization of the elements of a product, process or situation.

The role of data modeling in building database systems is similar to the role of an architect in building a substantial house or a building. In building a house, one does not just collect all the family members and draw some plans on a piece of paper and give it to a builder expecting it to lead to a well designed house that will meet the current and future needs of the family. Similarly, in building an enterprise database system one does not just collect all the users of the database and draw some plans on a piece of paper and give them to a database builder expecting it to lead to a well designed database system that will meet the current and future needs of the enterprise. Data modeling is therefore similar to the work of an architect whose job is to design a model that transforms the client's requirements into specifications which are then given to the database builder. Just like the architect does not disappear when the builder arrives, a database modeler also does not disappear when the database builder takes over. Both the architect and the database modeler have a role in supervising the building activity.

This chapter is organized as follows. The benefits of data modeling are discussed along with a description of various types of models in Section 2.3 and phases of database modeling in Section 2.4. In Section 2.5 the Entity-Relationship model is discussed in detail, including the definition of an entity, a relationship, different types of relationships, attributes and how to represent an Entity-Relationship model by an Entity-Relationship diagram. Generalization, specialization and aggregation are discussed in Section 2.6. The extended E-R Model is presented in Section 2.7 followed by discussion of database design in Section 2.8. Section 2.9 describes the strengths of the E-R model while Section 2.10 describes its weaknesses. A detailed case study of building an Entity-Relationship model is then presented in Section 2.11. Section 2.12 deals with evaluation of data model quality.

2.2 BENEFITS OF DATA MODELING

As noted earlier, models are built for a variety of purposes. Database modeling is essential in building any nontrivial database system for assisting communication between the client and the database designer. Modeling has the following benefits:

1. *Focusing on essentials*—Modeling often leads to developing an abstract and simple view of the product or process that is the subject of modeling. It is thus possible to ignore the details that tend to distract from the essential features. For example, when modeling a library we need to focus on the essential features, e.g., borrowers, books, serials, and at least in the beginning it may be possible to ignore that there are a variety of books (for example, some may have several editions) and serials (for example, some are published monthly while others quarterly and so on).
2. *Ease of communication and understanding*—Model building formalizes the process of reliable communication and assists in reaching an agreement. Model building therefore helps all parties involved in building a model to reach a common understanding of the subject of modeling, given that modeling formalizes communication. Communication during modeling also involves developing documentation describing what is being modeled. A model may be expressed in text or in a graphical language or it may even be more concrete like a prototype. A suitable expression medium should be chosen to assist communication. Designing a good library system would require suitable communication between the database designer, the librarians and the users (for example, academics and students, for a university library)
3. *Product or process improvement*—Since modeling involves communication between the various stakeholders, modeling can lead to improvement in the process or product that is being modeled as it provides a forum for consideration of alternative ways of doing things. For example, a process being modeled may involve enterprise customers supplying some information by completing one or more forms. Discussion during modeling may lead to these forms being improved, some forms being eliminated while others are put on the Web. In another modeling example, the design process may lead to the enterprise website being used more effectively thus improving customer service.
4. *Exploring alternatives*—In some modeling exercises, modeling can assist in answering 'what if' questions before building the database system. A user may want to explore whether it makes sense to include an aspect of the enterprise in the proposed database system. Sometimes it may be necessary to build a prototype to evaluate alternatives. For example, in a university it may be worth exploring whether the system should allow the students to enrol online if the university does not already do so.

2.3 TYPES OF MODELS

There are many different types of models. Models may be classified in the following classes according to the aim of modeling:

1. *Descriptive*—The primary purpose of such models is to describe or understand phenomena or complex machinery. For example, meteorologists build models to understand and predict weather; investment companies build models of share markets to understand the share market and to predict share prices or share index movements; and a high school teacher may build simple models to describe or explain some scientific concept to students. The aim for such models is to describe or understand what is being modeled.
2. *Prescriptive*—The primary purpose of prescriptive models is to clearly specify what a piece of machinery or software is supposed to do. In some cases prototype models may be built to specify how something is supposed to behave or look. The aim for such models is to specify what is being modeled. As an example, it has been shown that prescriptive models of human problem solving are able to explain how humans solve problems.
3. *Representative*—The primary purpose of such models is to simulate the behaviour of some phenomena or machinery. For example, computer games that simulate racing cars may be considered representative models. Representative models have been built in many other fields, for example, models of face recognition techniques. The aim for such models is to simulate what is being modeled.

A model does not have to belong to one class; it can belong to more than one class. Database models are normally prescriptive since they prescribe or specify what the database system is supposed to do although they also serve a descriptive role in helping communication between the designer and the customer by describing what the database would include. Therefore before the data available in an enterprise can be put in a DBMS, a database model or an overall abstract view of the enterprise data must be developed. The view can then be translated into a form that is acceptable by the DBMS. Although at first sight a modeling task may appear trivial, it is a very complex process for large database systems since the process often involves making compromises between what the users want and what can be realistically implemented. For example, if a user wants one single model to model all the cricket statistics that are available at the *cricinfo* website (http://stats.cricinfo.com/ci/engine/current/records/index.html), the data model would be very complex.

2.4 PHASES OF DATABASE MODELING

In the last chapter we presented the different phases of database design life cycle. One of the phases of the design life cycle was database modeling. This phase is discussed in more detail in this section.

The complexity of mapping the enterprise data to a database management system can be reduced by dividing the process into two phases. The first phase as noted above involves building an overall view (or model) of the *real world* which is the enterprise. This process is often called *conceptual modeling*. The objective of the model is to represent, as accurately as possible, the information structures of the enterprise that are of interest and independent of all physical considerations. This model is sometimes called an *enterprise conceptual*

schema (a schema essentially shows the structure of the data in a database). The process of developing the model may be considered as the requirements analysis of the database development life cycle. As noted earlier, this phase assists in communication between the users and designers of the database system. This model is then mapped in the second phase, and the logical design phase, to the database schema appropriate for the database model to be used. The logical design process is an abstraction process which captures the meaning of enterprise data without getting involved in the details of individual data values or details of the DBMS to be used for implementation. Figure 2.1 shows this two-step process.

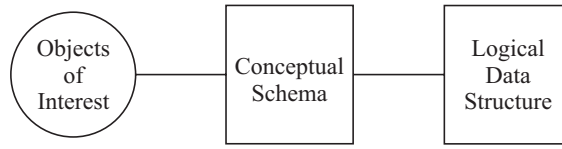

Figure 2.1 Two phases of database modeling

The process is somewhat similar to modeling in the physical sciences. In conceptual database design, a model is built to enhance understanding and to abstract details. A model should provide a reasonable interpretation of the real-life situation. A database designer therefore must explicitly consider what is to be included in the model and what is to be left out. For example, a model of employee data in an enterprise may not be able to capture the knowledge that employees Radha Rani and Krishan Kumar are married to each other. We accept such imperfections of the model since we want to keep the model as simple as possible. It is clearly impossible (and possibly undesirable or even illegal, for example, because of privacy concerns) to record every piece of information that could be available in an enterprise. We only desire that the meaning captured by a data model should be adequate for the purpose that the data is to be used for.

Data modeling often requires careful analysis of basic assumptions underlying the users' view of their world. For example, in creating a cricket database, we may need to consider the following:

- Who exactly is a batsman? Are all players in the team batsmen? Are all players in the team bowlers?
- How do we wish to interrogate the database? Would a player's full name be used or the first and the last name or just the last name or any of them? Names from different regions, for example, from north India and from south India, are quite different and a model should be able to deal with all names in a sensible way.
- Would the database allow a situation in which a player has represented more than one country during his career? If yes, how would such situations be modeled?
- Is it possible that a player could change his name during his career? For example, a player may convert from Christianity to Islam or from Hindusim to Christianity. If the database is to allow such rare incidents, how would such situations be modeled?

The person organizing the data to set up the database not only has to model the enterprise but also has to consider the efficiency and constraints of the DBMS. Implementation issues are considered in a third phase after the logical model has been completed.

As noted earlier, when the database is complex, the database design phase is likely to be challenging. A number of techniques are available for database modeling but we discuss only one such technique that is popular, and we believe, is easy to understand. The technique uses a convenient representation that enables

the database designer to view the data from the point of view of the whole enterprise. This well known technique is called the Entity-Relationship model.

2.5 THE ENTITY-RELATIONSHIP MODEL

The Entity-Relationship model (E-R model) is a widely used database modeling technique because it is simple and relatively easy to understand. The method facilitates communication between the database designer and the end user. It enables an abstract global view (that is, the *enterprise conceptual schema*) of the enterprise application to be developed without any consideration of efficiency of the ultimate database. The Entity-Relationship model was originally proposed in 1976 by Peter Chen although many variants of the technique have subsequently been devised.

E-R modeling is based on the concept that organizations consist of people, facilities and objects. These components of an enterprise interact with each other to achieve certain goals of the enterprise. As an example, when dealing with an enterprise that deals with cricket, the batsmen interact with bowlers, fielders and umpires to achieve the aim of playing a match. Another example is to look at a book publishing business which involves books, authors, printers, distributors, editors, sales data and so on.

All such things that are of interest to the enterprise are called *entities* and their interactions are called *relationships*. To develop a database model based on the E-R model involves identifying the entities and relationships of interest. These are often displayed in an E-R diagram. Before discussing these, we need to describe an entity and a relationship.

2.5.1 Entities and Entity Sets

An entity is a real or abstract object. It may be a person, a place, a building, or an event like a contract or a concert or an order that can be distinctly identified and is of interest. An entity exists independent of what does or does not exist in an enterprise. A specific cricket player, for example, Sachin Tendulkar with player ID 89001 or a venue Eden Gardens with a venue ID 65478 or a country India are examples of entities assuming that the database application requires information about them. Although entities are often concrete like a company, an entity may be abstract like an idea, concept or an event, for example, an ODI match 2688 between India and Australia or a Delhi to Mumbai flight number IA123. Whether an entity is real or abstract, the model makes no distinction between them but they must be something about which information is important.

Definition—Entity
An entity is a real or abstract object that can be distinctly identified and is of interest. Whether an entity is real or abstract, it must be something about which information is important.

The kind of entities needed by a database depends on the requirements of its applications. It is usually not possible to decide which entities are of interest to an enterprise based on direct observations of things in an enterprise. That is why a model must be developed to identify all entities that are of interest to an enterprise.

An entity must of course be identifiable so it must have a name or an identifier. In addition an entity must have some information beyond its identifier for it to be an entity.

A model is only going to reflect the perspective of a certain application. Often it will not reflect the whole enterprise. The part of an enterprise that is of interest is called the *Universe of Discourse* (or U of D).

Entities are classified into different *entity sets* (or *entity types*). An entity set is a set of objects called *entity instances* or *entity occurrences*. An entity instance is an individual object of a given entity type. For example, all cricket players may constitute an entity set *Player* with a particular player (say, Ricky Ponting) being an instance of *Player* and all matches may belong to an entity set *Match* with a particular match (say, the 2003 match between Australia and India played at the MCG) being an instance of the entity *Match*. An entity may belong to one or more entity sets. For example, a cricket player would belong to *Player* and can also belong to another entity set *Captain*. Entity sets therefore are not always disjointed.

Figure 2.2 shows two instances of entity *Bestbatsmen*. Repeating what has been noted above, an entity set or entity type is a collection of (usually more than one) entity instances of the same type. A clear distinction therefore needs to be made between the terms *entity set* or *entity type* and *entity instance*.

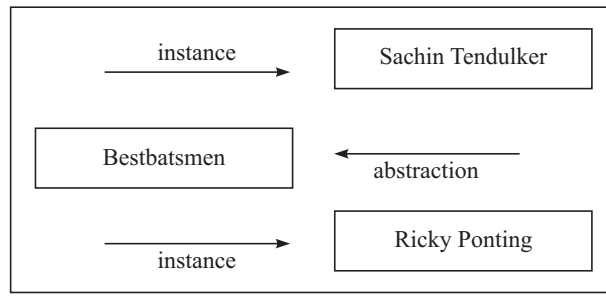

Figure 2.2 Entity instances of entity *Bestbatsmen*

As we have noted earlier, all entities have information about them that is of interest and each entity is described by the information about its properties. Instances of the same entity set are described by the same set of properties. It is, for example, not possible to have two members of an entity set *Bestbatsmen* where one player has one set of properties and the other has a different set of properties or one player is identified by his name while another by his player ID.

Real world enterprises consist of entities as well as interactions between them. Therefore a database is a collection of interrelated information about an enterprise. A database consisting of a number of entity sets also includes information about relationships between them.

2.5.2 Relationships, Roles and Relationship Sets

An enterprise database may consist of many entities, perhaps hundreds or even thousands of them in a large enterprise. All of these entities will have associations between them although some of the associations may be of no interest to the enterprise. In spite of this no entity even in a large enterprise can be on its own with no interaction with one or more other entities. As noted above, all interactions need not be represented in the database. For example, two academics in a university may have research interests in the same area but for

some applications this may be of no interest to the university. Usually an association between any two or more entities will be defined for a given instant of time since associations can change with time.

This relationship can be defined as follows:

Definition—Relationship
A relationship is an association among entities.

Relationships exist between entities and may be looked upon as mappings between two or more entity sets. A relation therefore is a subset of all possible combinations of entity instances from each entity type participating in the relationship. The set of all possible combinations can be large, for example, there can be 10,000 possible relationship instances if each of the two entities has 100 entity instances. These combinations are called the cross product of the entities involved in the relationship. Let us consider an example involving an entity set *Player* and another *Match*. All possible combinations of the entity instances would be a large number of relationship instances meaning that each player had played in every match, which obviously does not make sense.

The entities associated in a relationship may be of the same type or of different types. For example, batting is a relationship between the entity sets *Player* and *Match*. Captain is a relationship between the entity set *Player* with itself.

Each entity set in a relationship has a role or a function that the entity plays in the relationship. For example, a *Player* entity in a relationship *Batting* has a role of a batsman while the role of the entity set *Match* can only be best described as match.

Often relationship names are verbs and entity names are nouns which enables one to read the entity and relationship, for example, as 'player batted in match'.

Although we allow relationships among any number of entity sets, the most common relationships are binary relationships between two entity sets. The *degree* of a relationship is the number of entity sets associated in the relationship. Unary, binary and ternary relationships therefore have degrees 1, 2 and 3 respectively.

We consider a binary relationship R between two entity types E_1 and E_2. The relationship may be viewed as two mappings $E_1 \rightarrow E_2$ and $E_2 \rightarrow E_1$. It is important to consider the constraints on these two relationships (or mappings). It may be that one object from E_1 may be associated with exactly one object in E_2 or any number of objects in E_1 may be associated with any number of objects in E_2. Often the relationships are classified as one-to-one (1-1), one-to-many (1-*n*) or many-to-many (*m*-*n*). For example, marriage is a one-to-one relationship (in most countries) between an entity set *Person* to itself. If an entity in E_1 can be associated with any number of entities in E_2, but an entity in E_2 can only be associated with at most one entity in E_1, the relationship is called one-to-many. In a many-to-many relationship, an entity instance from one entity set can be related to any number of entity instances in the other entity set.

We consider an example of entity sets *Employee* and *Office*[1]. Let us look at what these two entity sets might look like if all the information about them was put in two separate Tables 2.1 and 2.2.

1. Offices can be an entity only if we have some information, in addition to its identifier, about each office. The information in Table 2.2 includes the office number, building name, capacity (number of people), and area in square metres. More information like number of telephones, etc., could also be of interest.

Table 2.1 Information about *Employee*

EMPID	*Name*	*Position*	*Address*
863456	J. White	Engineer	123 Main Street
921234	M. Lal	Designer	45 Kingsway
909876	J. Blue	Architect	654 Gandhi Road
952468	S. Kale	Manager	987 GT Road
799751	G. Black	Intern	32 New Road
857623	C. Bright	Programmer	234 TN Road
897689	L. Grey	Intern	32 New Road
993579	L. Green	Director	35 Delhi Road

Table 2.2 Information about *Office*

Office Number	*Building*	*Capacity*	*Area (sqm0)*
CS101	Computing	2	25
CS103	Computing	3	35
MA201	Mathematics	2	32
MA100	Mathematics	5	50
BU212	Business	3	42
BU301	Business	1	20
BU201	Business	4	40
CS108	Computing	4	45
CS110	Computing	3	30

We have put some information in the two tables although in a real practical application there is likely to be much more information than we have displayed in the tables. Also, we have assumed that the column *Building* has no information about buildings other than their names. If there was other information of interest about the buildings, we might need another table for that information. Such issues are discussed later in this chapter.

To relate the information in the two tables we need to have a relationship between them. Let us call this relationship *occupies* (since an employee occupies an office) and consider the following possibilities:

(a) If for each employee there is at most one office and for each office there is at most one employee, the relationship is one-to-one.

(b) If an office can accommodate more than one employee but an employee has at most one office, the relationship between *Office* and *Employee* is one-to-many.

(c) If an office can accommodate more than one employee and an employee can be assigned more than one office, the relationship between them is many-to-many. For example, an engineer may have one office in the workshop and another in the design office. Also each design office may accommodate more than one staff member. The relationship is therefore many-to-many.

These three relationships are shown in Figs 2.3(a), 2.3(b) and 2.3(c).

We now define the concept of cardinality ratio of a relationship.

Definition—Cardinality Ratio of a Relationship
Cardinality ratio of a binary relationship is the maximum number of relationship instances that an entity can participate in.

The cardinality ratio may be one-to-one, one-to-many, many-to-one, or many-to-many. Examples of different types of relationships are given below.

One-to-one Relationship

Figure 2.3(a) shows a one-to-one relationship between *Employee* and *Office*. The relationship shows that each instance of *Employee* in the relationship is related to at most one instance of *Office* and every instance of *Office* is related to at most one instance of *Employee.*

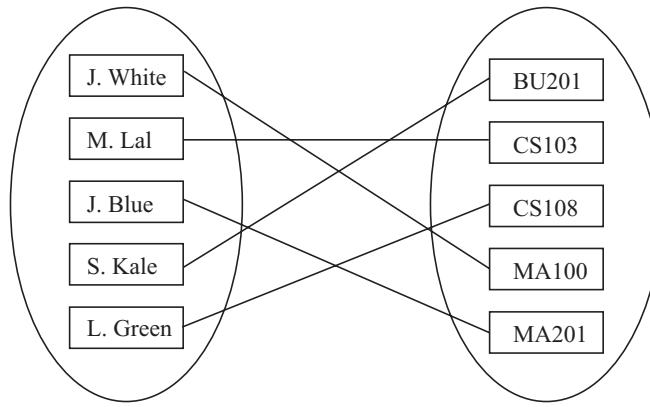

Figure 2.3(a) One-to-one relationship between *Employee* and *Office*

Note that the relationship shown in Fig. 2.3(a) does not show all the instances of entities given in Tables 2.1 and 2.2 because some *Employees* may not yet have been allocated *Offices* and some *Offices* may not yet have any *Employees* allocated to them.

One-to-many Relationship

Figure 2.3(b) shows a many-to-one relationship between *Employee* and *Office*. In this case a number of employees may occupy the same office but an employee can have only one office. Note that each instance of *Employee* is only related to one instance of *Office* but each instance of *Office* may be related to one or more instances of *Employee*. This relationship is not that unusual. For example, I am writing this while visiting VIT University in Vellore where academics share offices and a number of students (up to six, I am told) live in the same hostel room which may result in similar relationships between entities *students* and *rooms*.

Many-to-many Relationship

Figure 2.3(c) shows a many-to-many relationship between *Employee* and *Office*. In this case a number of employees may occupy the same office and an employee may have more than one office. Therefore a number

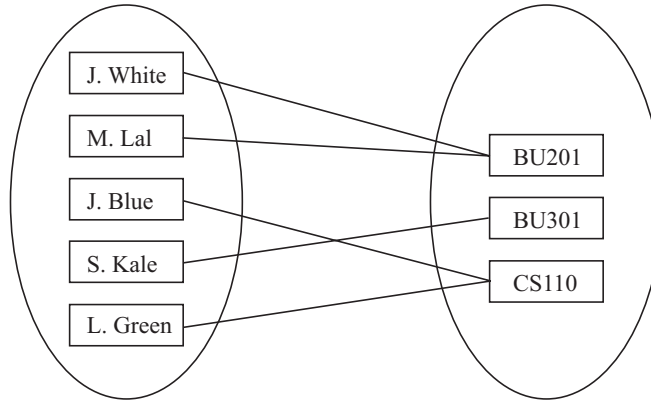

Figure 2.3(b) One-to-many relationship between *Employee* and *Office*

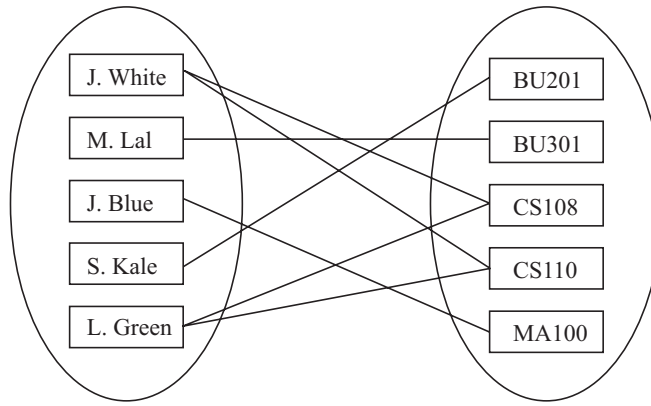

Figure 2.3(c) Many-to-many relationship between *Employee* and *Office*

of instances of *Employee* are related to one or more instances of *Office*. For example, J. White and L. Green share the office CS108 while L. Green has another office CS110 that is also shared with J. White.

The *Employee-Office* relationship is similar to many other relationships in life, for example, the relationship between lecturers and courses.

An easy way to determine the relationship type between two entities is to ask questions like the following:

- How many offices can an employee have?
- How many employees can an office hold?

As noted earlier, relationships are often binary but more than two entity types may participate in a relationship type.

Also there may be more than one type of relationship between the same two entity types. For example, there can be a number of relationship types between an entity set *employee* and another entity set *project* if one

of the relationships identifies employees working on projects while the other identifies employees that are project managers for projects. Another example is two relationships, *batting* and *bowling*, between the entity types *player* and *match*.

Ternary relationships, those between three entity types, are generally more complex than binary relationships. It is however in some cases possible to replace a ternary relationship by two binary relationships between the three entity types that participated in the ternary relationship. We will discuss this further later.

Ternary relationships are more complex and may be 1:1:1 or 1:1:*m* or 1:*m*:*n* or *p*:*m*:*n*. We leave it to the reader to think of examples for each type of ternary relationship.

2.5.3 Attributes, Values and Value Set

As we have noted earlier, an E-R database model consists of entities and relationships. Since these are the objects of interest, there is always information about them that is useful to the enterprise. This information in fact describes each entity instance in an entity set and the information of interest is called entity *attributes.* For example:

- Entity's name or identifier
- Entity's location
- Entity's other properties

Entity name or identifier enables us to identify each entity instance uniquely. If an entity does not have a unique name or identifier then each entity may be identified by values of a collection of attributes. In addition, each entity instance is described by properties or a set of attributes and their values at some given time. Often we are only interested in the current values of the properties and it is then convenient to drop time from the information being collected.

In some applications, however, time is very important. For example, a personnel department of a company would wish to have access to salary history and status history of all employees of the enterprise. A number of ways are available for storing such temporal data. Further details about temporal databases are beyond the scope of this book.

Relationships may also have relevant information about them. A cricket player in an entity set *player* is likely to be described by its attributes like *player id*, *name*, *country*, etc. A relationship like *batting* between entities *player* and *match* might have attributes like the *number of runs*, *number of fours*, *number of sixes*, etc., as attributes.

Types of Attributes

Attributes can be of several different types. As discussed earlier, they can be single-valued or multivalued, simple or composite or be stored or derived. We discuss these different types briefly.

- *Simple or Composite Attributes*—Most attributes are simple (also called atomic) and can not be divided into smaller components. Such attributes include country, place of birth, player ID, and number of runs and so on. Attributes that are composite can be divided into smaller components. Typical examples of composite attributes are name (consisting of at least the first name and the last name), date of birth (day, month and year), and address (street number, street name, city and PIN).

- *Single-valued or Multivalued*—Many attributes have a single value, for example, the age of a player, his last name, the player ID and so on. Some attributes on the other hand are multivalued. For example, names of umpires in a match is multivalued since there are two umpires on the ground, there is also a TV umpire, a reserve umpire and a match referee. Other typical examples of multivalued attributes are the courses a student is enrolled in or the programming languages an employee knows.
- *Stored or Derived*—Although it is not recommended, a database may have some attribute values that are related. A simple example is the total number of ODI runs scored in a player's career or the batting average, or the number of centuries scored. These are all derived attributes based on stored attributes like the number of runs scored in each match. The age of a player may also be a derived attribute if the database has a stored attribute that gives the date of birth.
- *Required or Optional*—An attribute may be required or it may be optional. Required attributes include player ID, match ID, and player's last name and first name. Optional attribute may be the year of the first test in an ODI database since some ODI players may not have played in a test match.

For each attribute there are a number of legal or permitted values. These sets of legal values are sometimes called *value sets* or *domain* of the attribute. For example, a player id may have legal values only between 10000 to 99999 while the year value in a match date may have admissible values only between say 1930 and the current year. A number of attributes may share a common domain of values. For example, a batsman's run score and a bowler's runs conceded may have the same legal values.

An attribute may be considered as a function that maps from an entity set or a relationship set into a value set. For example, the attribute *age* may map into an integer value set with values between 15 and 45 if the attribute age belongs to an entity player. The set of (attribute, data value) pairs, one pair for each attribute, partially describe each entity instance. For example, an entity instance of entity set player may be described by

- (player id, 101122)
- (player name, Ricky Ponting)

It should be noted that a data model has no way of knowing what an attribute or property means. The data models do not have any semantic knowledge of the model. The name of the person could be *Eman* but that has no impact on the model as long as the user understands that *Eman* is the name of the person.

As an example, consider a database that includes Tables 2.1 and 2.2 as entity sets *Employee* and *Office* and two additional entity sets *Department* and *Project* which are not shown. We therefore now have the following four entity sets.

- *Employee*—the set of all employees of a company. The attributes of the entity set *Employee* are EmpID, Name, Position and Address.
- *Office*—the set of all offices in a company. The attributes of the entity set *Office* are Office Number, Building, Capacity and Area.
- *Department*—the set of all departments of the company. The attributes of the entity set *Department* are Department Number, Department Name, SupervisorID.
- *Project*—the set of all projects at the company. The attributes of the entity set *Project* are Project Number, Project Name, ManagerID.

Once again, a real-world example would include a lot more information in the database.

Note that no relationships have been included in the list above. We propose three relationships although more may be required.

- *Occupies*—this is a relationship between entity sets *Employee* and *Office*. The attributes of the relationship set *occupies* are EmpID, Office Number, Date Allocated.
- *Works_for*—this is a relationship between entity sets *Employee* and *Department*. The attributes of the relationship set *works_for* are EmpID, Department Number, Date Joined, Qualifications, and Position.
- *Works_on*—this is a relationship between entity sets *Employee* and *Project*. The attributes of the relationship set *works_on* are EmpID, Project Number, Date Joined, Status, Position.

A careful reader would have noted that all the attributes for relationships shown above need not be included in the relationships. For example, should Qualifications and Position really belong in *Works_for*? What is the meaning of Position in *Works_on*? We will not resolve such issues here but will discuss them later in this chapter and also in Chapter 5 on normalization.

It should be noted that many further attributes would be included in the entity sets and relationship sets in a real-world application. The additional entity sets and relationship sets included in the database may also be represented as tables, similar to Tables 2.1 and 2.2. We give an example of a table representation for relationship set *Occupies* in Fig. 2.3(c). It is shown in Table 2.3 although mapping of entities and relationships to tables will be discussed in detail in Chapter 3.

Table 2.3 Relationship *occupies* shown in Fig. 2.3(c) represented as a table

EmpID	*Office Number*	*Date Allocated*
863456	CS108	01/03/2005
863456	CS110	02/04/2008
921234	BU301	03/07/2010
909876	MA100	31/12/1999
952468	BU201	06/06/2006
993579	CS108	01/01/2000
993579	CS110	07/07/2007

Note that the office CS108 is occupied by two employees and so is CS110.

2.5.4 Representing Entities and Relationships

Since each entity set normally consists of many entity instances, it is essential to be able to uniquely identify each entity instance in every entity set. Similarly it is essential to be able to uniquely identify each relationship instance in every relationship set. Since entities are represented by the values of their attribute set, it is necessary that the set of attribute values not be identical for any two entity instances. If two entity instances from an entity set have exactly the same attribute values and the entity instances are different (although it may be difficult to imagine when such a situation might arise) then an artificial attribute may need to be included in the attribute set to assist entity identification. An artificial attribute may also be included to simplify entity identification. For example, although each instance of the entity set *Employee* in a company database perhaps can be identified by the values of its attributes *name* and *address*, it is convenient to include an artificial attribute *EmpID* to make identification of each Employee entity instance easier.

Entity Key, Candidate Key and Super Key

A group of attributes (possibly one) used for identifying each entity in an entity set is called an *entity key*. For example, for the entity set *Employee, EmpID* is an entity key and so is (*Name, Address*). Of course, if *k* is a key then so is each super set of *k*. A *superkey* is any set of attributes such that no two tuples of a relation can have the same values for *k* for every possible pair of distinct tuples.

To reduce the number of possible keys, it is usually required that a key be *minimal* in that no subset of the key should be a key. For example, *name* by itself could not be considered an entity key since two employees could have the same name. When several minimal keys exist (such keys are often called *candidate keys*), any semantically meaningful key may be chosen as the *entity primary key*.

Definition—Primary Key
The primary key of an entity is an attribute or a set of attributes that uniquely identifies a specific instance of an entity. Every entity in the Entity-Relationship model must have a primary key.

Definition—Candidate Key
When an entity has more than one attribute (or a set of attributes) that can serve as a primary key, each of such keys is called a candidate key. One and only one of the candidate keys is chosen as the primary key for the entity.

It should be noted that a superkey is unique whereas, as discussed in the next chapter, a candidate key is both unique and irreducible.

Similarly, each relationship instance in a relationship set needs to be identified. This identification is always based on the primary keys of the entity sets involved in the relationship set. The primary key of a relationship set is the combination of the attributes that form the primary keys of the entity sets involved. In addition to the entity identifiers, the relationship key also (perhaps implicitly) identifies the role that each entity plays in the relationship.

If *EmpID is* the primary key of the entity set *Employee* and *ProjNum* is the primary key of *Project*, the primary key of the relationship set *Works_on* is (*EmpID, ProjNum*). Similarly the primary key for the relationship set *Works_ for* is (*EmpID, DeptNum*) since *EmpID* is the primary key of the entity set *Employee* and *DeptNum* is the primary key of *Department.* The role of the two entities is implied by the order in which the two primary keys are specified.

Weak Entity Types

In certain cases, the entities in an entity set cannot be uniquely identified by the values of their own attributes. For example, children of employees in a company may have names that are not unique. One solution to this problem is to assign unique numbers to all children of employees in the company. Another, more natural, solution is to identify each child by his/her name and the primary key of the parent who is an employee of the company. The names of the children of each employee are expected to be unique. Such an attribute(s) that discriminates between all the entities that are dependent on the same parent entity is sometimes called a *discriminator*. It cannot be called a key since it does not uniquely identify an entity without the use of the

relationship with the employee. Similarly, history of employment of an employee if included in the database would not have a primary key without the support of the employee's primary key. Such entities that require a relationship to be used in identifying them are called *weak entities*. Entities that have primary keys are called *strong* or *regular entities*. Similarly, a relationship may be weak or strong (or regular). A *strong relationship* is between entities each of which is strong; otherwise the relationship is a *weak relationship*. For example, any relationship between the children of the employees and the schools they attend would be a weak relationship. A relationship between the *Employee* entity set and the *Project* entity set is a strong relationship.

A weak entity is also called a *subordinate* entity since its existence depends on another entity (called the *dominant* entity). This is called *existence dependence*. The weak entity therefore does not possess sufficient attributes to form a primary key. This is in contrast to a regular or strong entity that has key attributes and therefore does not exist because of another entity. The relationship of a weak entity to its owner entity type is called the *identifying relationship*.

It is interesting to look at the sources of weakness in a weak entity. Often it is because of the hierarchy of entity sets. For example, species name and genus name may form a key of an entity called plant.

Another example of weak entity is that a customer has several accounts with a bank. Every transaction on each account has transaction number but different transactions in different accounts could have the same number so the transaction number by itself is not sufficient to form a primary key. Therefore a transaction cannot be uniquely identified without the account number and is therefore a weak entity set. For a weak entity set to be meaningful, it must be part of a one-to-many relationship set. This relationship set should not have any descriptive attributes.

A weak entity set does not have a primary key but we need a means of distinguishing among the weak entities. The discriminator of a weak entity set is a set of attributes that allows distinction between instances of a weak entity to be made. Therefore, the primary key of a weak entity is found by taking the primary key of the strong entity on which it is existence-dependent combined with the discriminator of the weak entity set.

Although existence dependency does not imply ID dependency, a weak entity is often *ID dependent* on the dominant entity. This method of identifying entities by relationships with other entities can be applied recursively until a strong entity is reached. Although the relationship between the dominant entity and a weak entity is usually one-to-many, in some situations the relationship may be many-to-many. For example, a company may employ both parents of some children. The dependence is then may be many-to-many.

Terms that are useful:

(a) Weak entity—an entity that depends on another entity for identification.
(b) Weak entity relation—a relation that is used for identifying entities, for example, the relationship between employees and their dependants.
(c) Regular entity relation—a relation not used for identifying entities.

2.5.5 Naming Conventions

It is best to be consistent while giving names to entities, relationships and attributes. One commonly used practice is to use a singular name for all entities and use verbs for all relationships. In many respects it doesn't

matter which one (singular or plural) is used as long as it is used consistently throughout the entire database schema design.

In this book we will be using the following conventions:

Entities

Entity names should have the following characteristics:

- Each entity name should come from the entity description, which has meaning to the user community.
- Each entity name should be a noun, singular and current tense. It is something about which we want to keep information.
- The first letter should be uppercase.
- Where necessary, underscores should join the words.
- Names for the entities must be meaningful and must not conflict with other entity names. Names should not be abbreviated unnecessarily making them difficult to understand.
- Example of entity names are Account, Customer, Product, Employee, Department, Player.

Relationships

- Each relationship name should be a verb that fits the sentence structure.
- Where necessary, an attribute name may use underscores to join the words.
- Example of relationship names are Occupies, Married_to, Represents, Batting.

Attributes

Each attribute name should have the following characteristics:

- Generally an attribute name should be a noun.
- Where necessary, an attribute name may use underscores to join the words.
- An attribute name must be unique with no two attributes of an entity type having the same name. Attributes of different entities can have the same name.
- For example, PID, EmpID, Course_Number (these are identifiers), Status, Reason (these are categories), Name, Description, Comment (these are textual), Date, Date_of_Birth, Year (these are date related), etc.

2.5.6 Diagrammatic Notation for E-R Model

As noted earlier, Entity-Relationship modeling facilitates communication between the database designer and the client during the requirements analysis. To assist in such communication, the designer needs good communication tools. A diagram can be such a tool since it is said that a picture is worth a thousand words.

There are a number of software tools (often called CASE tools for Computer Aided Software Engineering) available in the market which provide facilities for drawing E-R diagrams. Each tool may have its own conventions for drawing diagrams and the conventions often vary slightly from tool to tool. A diagram representing an E-R model should include the following:

- *Entities, Relationships and Attributes*—These have been discussed earlier.

- *Cardinality Constraints*—These also have been discussed earlier. For binary relationships, cardinality may be 1:1, 1:*m* or *m*:*n*.
- *Participation Constraint*—The concept of participation specifies whether an entity instance in a relationship can exist without participating in the relationship. Participation is either mandatory or optional.
- *Generalization/Specialization*—This will be discussed later, they deal with the concepts of subclass and superclass.

In addition to the above, a diagram representing an E-R model also needs to deal with the following issues.

- *Are ternary relationships allowed or only binary?*—Some notations only allow binary relationships since that makes the diagrams simpler. Some ternary relationship can be transformed into a number of binary relationships but the semantics of the relationship often change when such transformation is carried out.
- *How is cardinality to be represented?*—Cardinality can be represented in two different ways. In the first case, a lecturer teaching a number of courses and each course being taught only by one lecturer will be represented by putting 1 next to the lecturer entity and *n* near the course. In the second case, the opposite is done.
- *Are relationships allowed to have attributes?*—Some notations do not allow attributes for relationships. In such a notation, a relationship with attributes must be represented as an entity.

There are a number of different diagrammatic notations suggested in the literature. We consider only two of them. The first notation is the one that Chen recommended in his original paper and is discussed next. The second is called the Crow's Foot Notation and is discussed in Section 2.5.10.

2.5.7 Entity-Relationship Diagrams (Chen's Notation)

We call Chen's notation an E-R diagram. The original notation included notations only for entities, relationships, attributes and cardinalities. It was later extended to include generalization and participation constraints.

Building Blocks of Entity-Relationship Diagram

In an E-R diagram, as shown in Fig. 2.4, the entity sets are represented by rectangular shaped boxes and relationships are represented by diamond shaped boxes.

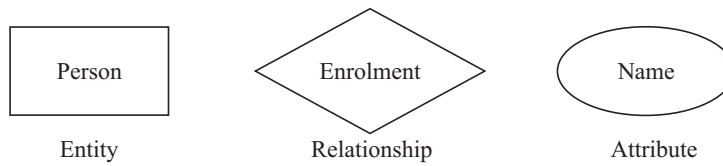

Figure 2.4 Elements of the Entity-Relationship diagram

Ellipses are used to represent attributes and lines are used to link attributes to entity sets and entity sets to relationship sets. We now present some examples of E-R diagrams.

A Simple Unary Relationship E-R Diagram

Consider a simple E-R diagram as shown in Fig. 2.5(a) that includes only one entity set which has a relationship with itself. In this relationship an employee and his/her supervisor are both instances of the entity set *Employee*. Therefore, the relationship between an employee and the supervisor is a relationship between two instances of the same entity type.

Figure 2.5(a) ERD for a unary relationship

A Simple Binary Relationship E-R Diagram

The E-R diagram as shown in Fig. 2.5(b) shows a simple relationship between two entity types *Employee* and *Department*. In this relationship an instance of the entity *Employee* works for an instance of the entity *Department*. It is not required that each instance of *Employee* be involved in this relationship. The same applies to the instances of the entity *Department* since some instances of *Employee* may not yet have been allocated to a department and some departments that are perhaps new may not have any employees.

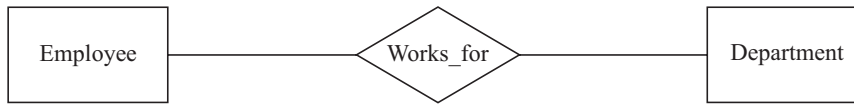

Figure 2.5(b) ERD for a binary relationship

A Simple Ternary Relationship E-R Diagram

The E-R diagram as shown in Fig. 2.5(c) shows a simple relationship between three entity types *Employee*, *Project* and *Department*. In this relationship an instance of the entity *Employee* works for an instance of the entity *Department* as well as an instance of the entity *Project*. Such relationships between three entities are called *ternary* relationships as opposed to binary relationships between two entities. It is not required that each instance of the three entities be involved in this relationship, since as noted above, some instances of

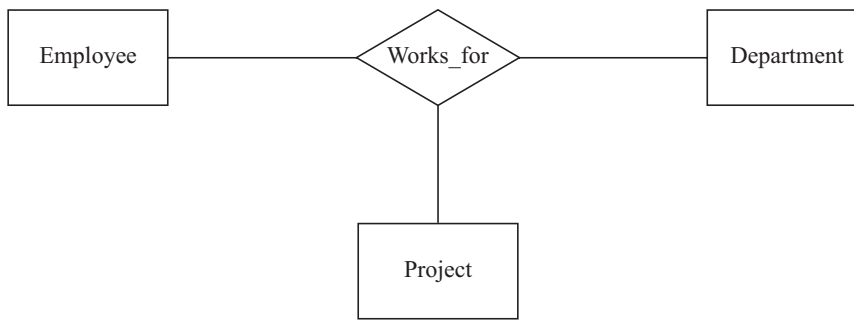

Figure 2.5(c) A ternary relationship

Employee may not yet have been allocated to a department and some departments that are perhaps new may not have any employees. Similarly, some projects may be so new that they have no employees or have not been allocated to a department.

It is important to note that ternary relationships add further complexity to a model. For example, for entity instances to participate in such relationships, each instance of the three entity types should be known. An instance of *Employee* who has been allocated to a *Department* but has no *Project* cannot participate in this relationship. Similarly, a *Project* that has been allocated to a *Department* but has no employees cannot participate in the relationship. We will discuss more about ternary relationships later in the section.

An E-R Diagram for Binary Relationship with Attributes

The E-R diagram shown in Fig. 2.5(d) presents a number of attributes for the model in Fig. 2.5(b). Another relationship *Manager* has been added between the entity *Employee* to itself. Although this E-R model is very simple, we can see that it is getting cluttered with more attributes. A more complex model obviously would require a lot of space for display.

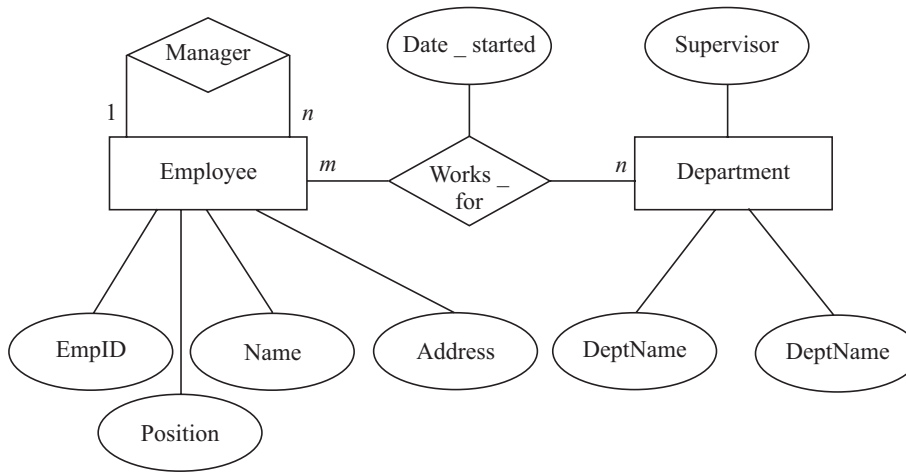

Figure 2.5(d) E-R Diagram for Fig. 2.5(b) with a number of attributes

An E-R Diagram that includes a Weak Entity and a Ternary Relationship

The E-R model shown in Fig. 2.5(e) is more complex with three additional entities including a weak entity and three additional relationships. The E-R diagram in Fig. 2.5(e) does not show the attributes because showing the attributes as well as the additional entities and relationships would make the diagram somewhat difficult to read. We should however note the following about the E-R diagram in Fig. 2.5(e):

1. The diagram shows that the relationship *Manager* is a one-to-many and the relationship *Works_on* is a many-to-many relationship and so on.
2. *Employment History* has been modeled as a weak entity since this entity depends on the entity *Employee*. If an instance of *Employee* is deleted then its dependent instances in *Employment History* would also be deleted.
3. A ternary relationship *PSP* exists between the entities *Project*, *Supplier* and *Parts*.

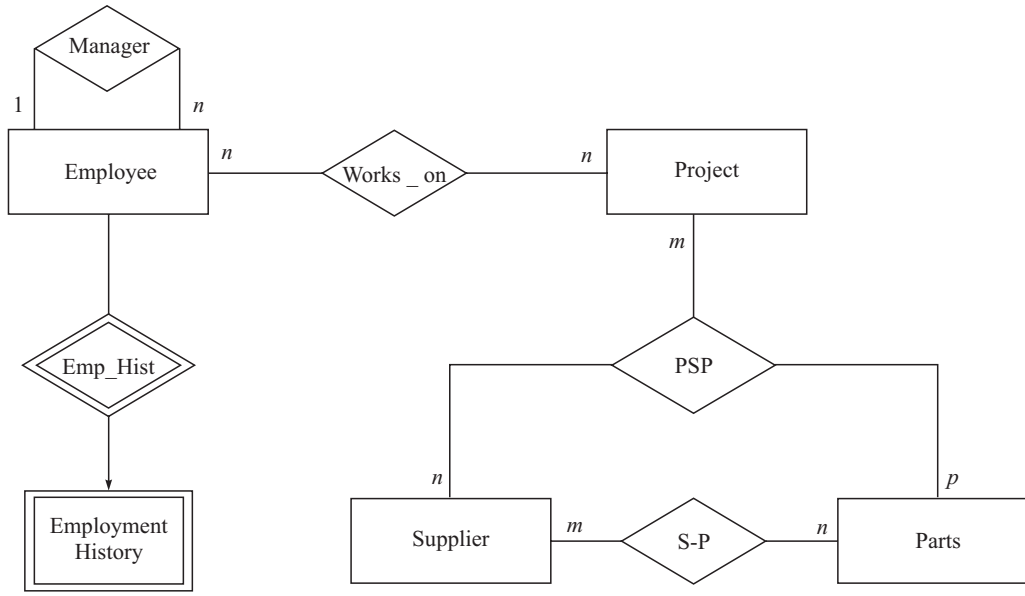

Figure 2.5(e) An E-R diagram showing a weak entity and a ternary relationship

An E-R Diagram for Two Binary Relationships between the Same Entities

We mentioned earlier that two similar entities can have more than one relationship. The simple E-R diagram in Fig. 2.5(f) presents two relationships, one being *Has_office_in* and the other *Has_proj_in* between the two entities *Staff* and *Building*.

An E-R Diagram with Two Unary Relationships for the Same Entity

A unary relationship is shown in Fig. 2.5(a) and 2.5(d). The E-R model in Fig. 2.5(g) shows that it is possible to have more than one unary relationship between a given entity and itself. The two relationships of the entity *Employee* with itself are *Coaches* and *Supervises*. Both coach and supervisor are employees. These relationships can of course have attributes. We have not shown any attributes in our diagram.

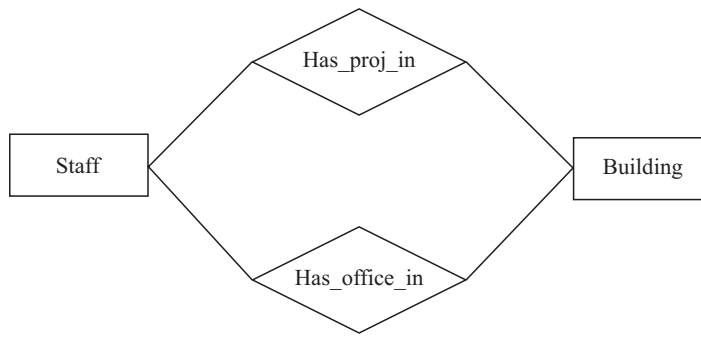

Figure 2.5(f) Two relationships between the same two entities

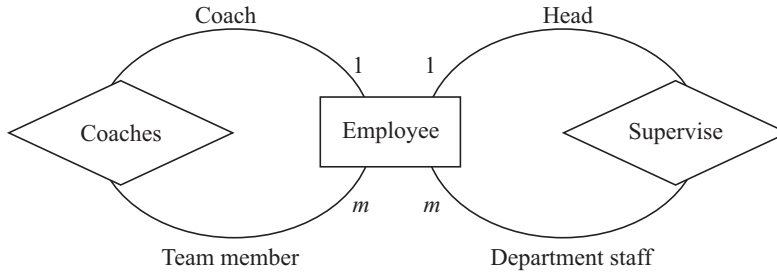

Figure 2.5(g) Two unary relationships

We briefly summarize the seven E-R diagrams presented in Figs 2.5(a) to 2.5(g). The E-R diagram in Fig. 2.5(b) only shows two entities and one relationship and not the attributes. Fig. 2.5(c) shows a ternary relationship. Figure 2.5(d) shows the same two entities but shows an additional relationship between employees and their manager. This additional relationship is between the entity set *Employee* and itself. In addition, the E-R diagram in Fig. 2.5(d) shows the attributes as well as the type of relationships (1:1, *m*:*n* or 1:*m*). The type of relationship is shown by placing a label of 1, *m* or *n* on each end of every arc. The label 1 indicates that only one of those entity instances may participate in a given relationship. A letter label (often *m* or *n*) indicates that a number of these entity instances may participate in a given relationship. The E-R diagrams may also show roles as labels on the arcs.

The E-R diagram in Fig. 2.5(e) shows a number of entities and a number of relationships. Also note that the entity *Employment History* is shown as a double box which indicates that *Employment History* is a weak entity and that its existence depends on the existence of the corresponding employee entity. The existence dependency of the entity *Employment History* on the entity *Employee* is indicated by an arrow. A ternary relationship between three entities is also shown. Note that each entity can participate in several relationships.

The E-R diagram in Fig. 2.5(f) shows two entities and two different relationships between them. Figure 2.5(g) shows two relationships also, but both these relationships are between the entity *Employee* and itself.

The above diagrams have shown entity sets and relationship sets between them. Figure 2.6 shows a number of instances of the relationship *Works_on* (*WO*) between instances of entities *Employee* (*E*) and *Project* (*P*). *E*1 to *E*6 are instances of the entity *Employee* while *P*1 to *P*4 are instances of the entity *Project*. *WO*1 to *WO*8 are instances of the relationship *Works_on*.

It should be noted that a primary key of relationship *WO*1 is (e_1, p_1) assuming e_1 and p_1 are the primary keys of the two entities *E*1 and *P*1 and so on for other instances of the relationships *WO*2 to *WO*8 shown in Fig. 2.6. It should be noted that the relationship in Fig. 2.6 is a many-to-many relationship. For example, the employee entity *E*1 is related with three project entities *P*1, *P*2 and *P*3 and the project entity *P*1 is related with two employee entities *E*1 and *E*6.

2.5.8 Ternary Relationships and Binary Relationships

As noted earlier, ternary relationships are more complex than binary relationships. One example of a ternary relationship is presented in Fig. 2.5(c). The three entities involved are *Employee*, *Department* and *Project*. We first define different types of ternary relationships. These relationships can be 1:1:1, 1:1:*m*, 1:*m*:*n* or *m*:*n*:*p*.

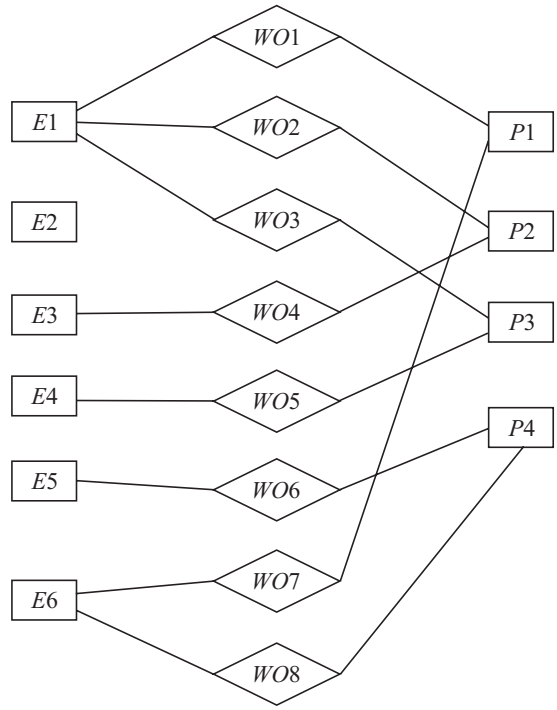

Figure 2.6 Instances of the many-to-many relationship *Works_on* (*WO*) between the entities *Employee* (*E*) and *Project* (*P*)

The ternary relationship in Fig. 2.5(c) is reproduced below in Fig. 2.7(a). In the simplest but unrealistic case the ternary relationship is 1:1:1, which means that for each employee there is only one department and only one project. Furthermore, a department has only one employee and only one project. And finally, each project has only one department and only one employee. This is obviously quite unrealistic but it is relatively simple to convert the 1:1:1 relationship to two binary relationships, one between *Employee* and *Project* and another between *Project* and *Department*.

If we now relax some of these constraints and allow a department to have more than one employee and more than one project while still requiring each employee to just belong to one department, each employee to work on only one project, each project to have only one employee and each project to be assigned to just one department, the relationship becomes somewhat more complex but it is still possible to transform the ternary relationship into two binary relationships. We leave this as an exercise for the reader.

The ternary relationship in Fig. 2.7(a) could have the following more realistic constraints:

1. An instance of *Employee* is related to only one instance of *Department* and only one instance of *Project*.
2. An instance of *Department* is related to many instances of *Employee* and many instances of *Project*.
3. An instance of *Project* is related to many instances of *Employee* and only one instance of *Department*.

If we now look at the three binary relationships, we find that the *Employee: Department* relationship is 1:*p*. The *Employee: Project* relationship is 1:*m*. Finally the *Department: Project* relationship is also 1:*n*. We

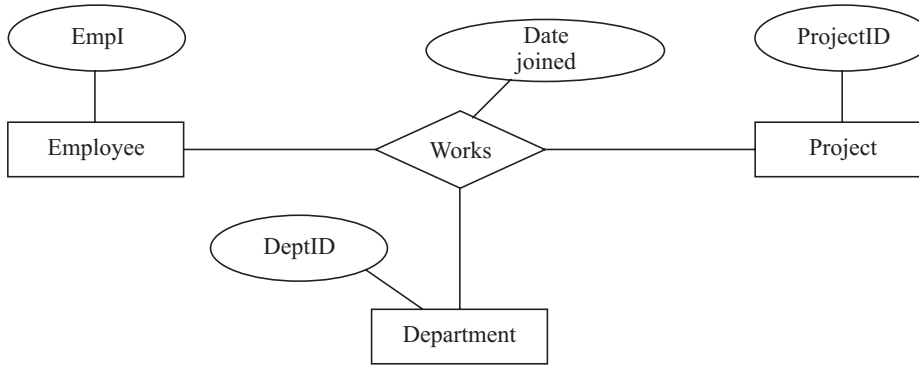

Figure 2.7(a) A ternary relation between *Employee*, *Project* and *Department*

therefore have three one-to-many binary relationships in the ternary relationship. Can they be represented by three binary relationships as shown in Fig. 2.7(b)?

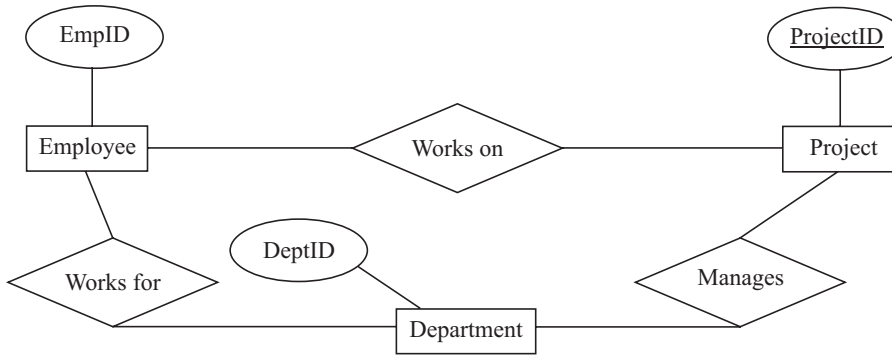

Figure 2.7(b) Three binary relations between the relations *Employee*, *Project* and *Department*

The three binary relationships shown in Fig. 2.7(b) are in fact not able to represent the same information as the ternary relationship in Fig. 2.7(a). What information can be represented in the E-R diagram shown in Fig. 2.7(a) that cannot be represented in the diagram shown in Fig. 2.7(b)? Is there information that the model in Fig. 2.7(b) can represent but the model in Fig. 2.7(a) cannot? We leave it to the reader to answer these questions.

Another E-R model that attempts to model the same information as the ternary relationship replaces the ternary relationship in Fig. 2.7(a) with a weak entity *Works* and three binary relationships between this entity and the other three entities.

2.5.9 E-R Diagram Conventions

A number of issues regarding conventions may arise while developing a database model. The entity-relationship model provides no guidance on how to deal with them. For example,

- Should one cross lines in the Entity-Relationship diagram when the model is complex?

- Should any derived attributes (attributes that can be computed from other attribute values) be included in the model?
- Should performance of the implemented database system be considered in modeling?
- What naming standards and abbreviations should be used? It should be clear what standards are to be used and why. Some suggestions have been made earlier in Section 2.5.5.
- Should a collection of attributes be used as a primary key or an artificial (called surrogate key) attribute be used as the primary key?

Given that no guidelines are available, we suggest such issues be resolved by the modeling team in consultation with the client and the DBA.

2.5.10 Crow's Foot Diagrammatic Notation

Although Chen's diagrammatic notation is most widely presented in literature, another widely used notation for an Entity-Relationship diagram is called the Crow's Foot notation. A similar notation is used by ORACLE in its CASE tool. In this notation, an entity is still represented by a rectangle but the attributes are written within the bottom part of the rectangle and the name of the entity is written in the top part of the rectangle while an asterisk (*) represents the attribute or the set of attributes that form the primary key. This notation is more compact than the notation used in the E-R diagrams. Some software available for E-R model use this notation.

This notation only includes binary relationships. No attributes of the relationships may be shown. Relationships are represented by lines joining entities. The cardinality of a relationship can be shown by drawing straight lines or lines with a crow's foot. Figure 2.8(a) shows the notation when used with a 1:1 relationship between *Employee* and *Office*. Note that the || symbol denotes 1 and only 1 participation.

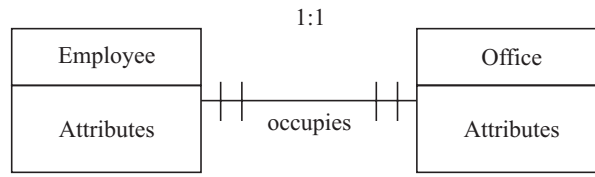

Figure 2.8(a) Crow's feet notation for two entities with 1:1 relationship

Figure 2.8(b) shows the notation when used with a *m*:*n* relationship between *Employee* and *Office*. Note that the | symbol denotes 1 or more participation. The crow's foot shows the many participation in a relationship.

There are many more details about the notation that we are not able to cover here. For example, the 1:1 and 1:*m* relationship attributes have to be represented by key propagation and a many-to-many relationship

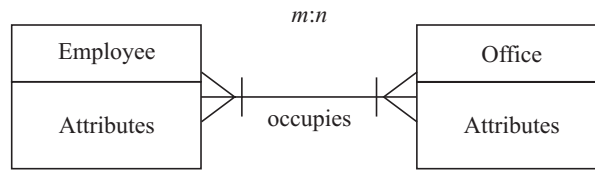

Figure 2.8(b) Crow's feet notation for two entities with 1:1 relationship

is represented by an entity called the *intersection entity* which must be put between the entities it relates. However, we will focus on using the Chen's Entity-Relationship diagrams.

2.6 GENERALIZATION, SPECIALIZATION AND AGGREGATION

The main idea in generalization/specialization is that one entity is a subset of another entity.

Although entity instances of one entity type are supposed to be objects of the same type, it often happens that objects of one entity type do have some differences. For example, as illustrated in Fig. 2.9, a company might have vehicle as an entity type but the vehicles could include cars, trucks and buses and it may then be necessary to include capacity of the bus for buses and the load capacity of the truck for trucks, information that is not relevant for cars. For cars, one may be interested in storing information regarding its type (for example, sedan or station wagon). In such situations it may be necessary to deal with the subsets separately while maintaining the view that all cars, trucks and buses are vehicles and share a lot of information. The specialization concept is a top-down concept that allows some other vehicles to be included in the entity *Vehicle* (for example, scooters) which are not an entity type in one of the specializations. Generalization is a bottom-up concept that does not allow such an option.

In Fig. 2.9 we assume that it was discovered (when the entity *Vehicle* was suggested) during initial modeling that there are likely to be a variety of vehicles and these vehicles will have some information that applies to only each group of vehicles. On the other hand, if entities: car, truck and bus, were already being considered during initial modeling and it was then realized that they were all vehicles which shared a fair bit of common information and that resulted in creation of the entity *Vehicle*, then that is generalization. Therefore, the concepts generalization and specialization are very similar; specialization being a top-down approach while generalization is the bottom-up approach.

Generalization therefore involves forming a higher-level entity type from pre-existing lower-level entity types. It is then assumed that an entity instance of the higher-level entity type must belong to one of the lower-level entity types. For example, if pre-existing entity types car and truck are combined to form a higher-level entity type vehicle then all vehicles are supposed to be either a car or a truck. Specialization on the other hand is formed by specifying subsets of a higher-level entity type. For example, if the higher level entity type was vehicle then the lower-level entity types car, truck and bus were designed then the higher-level entity type could include entity instances that do not belong to either of the lower-level entity types. So a scooter could be a vehicle although it would not be an entity instance of either of the lower-level entity types.

Primary purpose of generalization is to show similarities among a number of lower-level entity types, generalization involves the higher-level entity type inheriting attributes 'upwards' from the lower-level entity types. On the other hand, specialization involves the lower-level entity types inheriting attributes 'downwards' from the supertype entity.

In Fig. 2.9 we have shown the entity *Vehicle* which has instances v_1, v_2, … etc., while the entity instances of bus, truck and car have been given different names to differentiate them from the instances of the entity *Vehicle* but it should be noted that v_1 and t_2 are exactly the same entity. Similarly, v_2 and t_1 are exactly the same entity and so are v_3 and c_1. It should be noted that in generalization, every instance of each subtype is related to

an entity instance in the supertype while in specialization there may be some instances of supertype entity that are not related to any instance of the subtype entities, since their might be vehicles that do not belong to any of the subtypes.

Entity hierarchies are known by a number of different names. At least the following names are used in the literature, i.e., supertypes and subtypes, generalization and specialization and ISA hierarchies.

In the example above, entity type *Vehicle* will be termed a *supertype* entity and entities car or truck will each be called *subtype* entities. Subtype entities can be considered dependent entities since their existence depends on their parent, the supertype entity. Subtype entity sets are usually disjoint but can sometimes be overlapping. For example, in the somewhat unlikely situation where there was a vehicle that could be classified as a car as well as a truck then that vehicle would belong to two subtypes[2]. Another more likely example is of a supertype entity type person in a university with two subtype entity types staff and student. It is quite possible for some staff to be studying and therefore belong to both subtype entities.

There can be more than one level of supertype and subtype entities. For example, although the *Truck* entity type is a subtype, it can also be a supertype of subtype entities like forklift truck, refrigerated truck, fire truck, petrol truck, and container truck assuming these types have some information that differentiates them. The concept of entity hierarchy is important since it allows grouping of similar objects in a higher-level entity type (e.g., Vehicle) as well as allowing the user to concentrate on a lower-level entity type of interest (e.g., Bus).

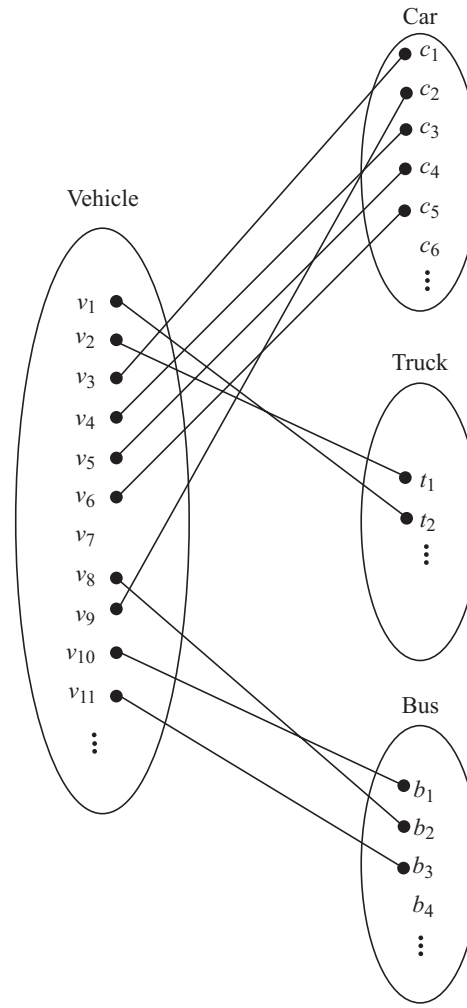

Figure 2.9 Illustrating specialization of entity vehicle

Aggregation abstraction is used in building composite entities from component entities. In aggregation, a new entity type (supertype) is defined from a set of entity types (subtypes) that represent its component parts. For example, engine, body, transmission, wheels, etc., may form an aggregation entity type vehicle. It is therefore another concept of entity hierarchy involving supertype and subtype entity types although it is different than supertype/subtype of specialization/generalization hierarchy. Supertype/subtype allows grouping of similar objects while aggregation allows grouping of several different components of the same thing to be grouped together.

2. There is in fact a vehicle called the *Ute* in Australia. It is a vehicle with the cabin of a car and the rear of a small truck. It was designed at the Ford Motor Company in Geelong, Victoria in1934. The *ute* has long been a favourite vehicle for farmers and tradesmen and is part of the Australian landscape. An Australian golfer is known to have even converted a BMW car into a ute!

Aggregation in E-R modeling may be implemented by including a relationship between the supertype entity and the component entities. It should be noted that when aggregation is used, deleting an instance of the supertype entity would normally involve deletion of all its component entity instances.

Example
Generalization

The example in Tables 2.4, 2.5 and 2.6 illustrates the concept of generalization. Table 2.4 is an entity *Person* in a university. It includes all persons at the university including students and staff since quite a lot of information about them is common. Not all such information which may include address, telephone number, date of birth, place of birth, date of joining the university, citizenship and so on is shown in the table because of space constraints.

Table 2.4 The supertype entity *Person*

ID	*First Name*	*Last Name*	*Gender*	*Status*	*Category*
67543	Arvind	Kumar	Male	FT	Student
76849	Bala	Narasimhan	Male	FT	Academic
86348	Salman	Khan	Male	PT	Both
72398	Deepaka	Dixit	Female	FT	Academic

Let us assume that the entity *Person* is a generalization of entities *Academic* and *Student*. All entity instances in *Person* also belong to one of the subtype entities, *Student* and *Academic*. The subtypes need not include information that is common to them and which is included in the supertype entity *Person*. The subtype entities only need to include information that is relevant to them because these entities will inherit the information from the supertype entity. In fact, the information in the subtype entities also need not be duplicated in the supertype entity. The supertype and subtype entities are related by their primary key.

Table 2.5 The subtype entity *Student*

ID	*Degree Enrolled*	*Date Enrolled*
67543	BSc	1 March 2007
86348	PhD	22 July 2006

Table 2.6 The subtype entity *Academic*

ID	*Highest Qualification*	*Discipline*	*Position*
76849	PhD	Maths	Professor
72398	PhD	Computer Science	Professor
86348	BCom(Hons)	Business	Tutor

Note that all entity instances in *Person* participate in one of the lower level entities and one entity instance participates in both *Student* and *Academic*. Therefore, this model allows overlaps in the subtype entities.

The participation constraint on a generalization and specialization is normally 1:1, since every entity instance in the subtype entities is in the supertype entity. Aggregation relationship sets have participation constraints. The constraint for each subpart class specifies how many of these parts may be related to the supertype.

2.7 EXTENDED ENTITY-RELATIONSHIP MODEL

The classical E-R model cannot fully capture the richness of real-life situations and the complexity of some newer database applications. Therefore the E-R model has been extended to provide greater semantics by including additional facilities. E-R modeling with additional modeling facilities is called the Extended Entity-Relationship model (or EER model). EER includes subset hierarchies as well as generalization hierarchies. We discuss some of these features of EER model in this section. We have discussed in the last section the concepts of generalization and specialization. These concepts are included in the EER model.

2.7.1 Supertypes, Subtypes and Inheritance

An example of generalization was presented in Fig. 2.9.

Let us consider another example. We have an entity *Student* in a large university database model. The university may decide to use just one entity type *Student* to store all the information about all students in its database. This can lead to some problems in a university like Monash which has more than 50,000 students that includes more than 15,000 postgraduate students (about 3500 research postgraduates and 12000 coursework postgraduates) and almost 20,000 international students. These different categories of students will have much information that is common to them all, for example, we assume the following information is a short list of what is common for an institute or a university in India:

- ID
- First name
- Family name
- Gender
- Citizenship
- Scheduled cast/Scheduled tribe/Backward class
- Current address
- Email address
- Telephone number
- Degree program enrolled in

There is little doubt that there will be quite a bit of information that will be different for each category of students. For example:

- For undergraduates, high school attended, current courses enrolled in
- For postgraduates, undergraduate degree, university of undergraduate degree, thesis topic, name of the supervisor, scholarship, etc.

- For international students, school/university attended, name of the country of origin, home address, home contact, etc.

There is much more information that could be listed.

We could model such a situation by describing each student by an entity type *Student* and in addition have other entity types *Undergraduate*, *Postgraduate*, *International*. Conceptually, all the common information is stored in *Student* while all the specific information is stored in entities *Undergraduate*, *Postgraduate*, and *International*. As explained in the last section, this process of modeling a group of objects and their subgroups is specialization. We could call this process as a top-down modeling process that started by identifying differences in different classes of *Student*.

In such a situation, all the attributes of *Student* will be applicable to all the subtypes but each subtype (*Undergraduate*, *Postgraduate*, and *International*) will have additional attributes that are not in the supertype entity *Student* as illustrated above. The subtype-supertype relationship specifies constraints on the entity sets since it states that the subtype must also be a member of the supertype entity and it must include all the attributes of the supertype. This is called *attribute inheritance*. The relationship between the supertype and subtype is often called *IsA* relationship.

There may be more than one level of subtype-supertype relationship. For example, *International* entity may have subtypes of *Int_undergraduate*, *Int_postgraduate* subtypes. Therefore, we have created a hierarchy in which entity *Int_undergraduate* is a subtype of *International* as well as a subtype of the entity *Student*. The hierarchy is therefore transitive which implies that an *Int_undergraduate* inherits properties from *International* as well as from *Student*.

2.7.2 Aggregation or Participation

As noted in Section 2.6, aggregation is used in modeling when an entity is the aggregate composed of other entities from the subpart classes. Aggregation therefore is used in building composite entities from component entities.

Aggregation in E-R modeling may be implemented by including a relationship between the supertype entity and the component entities. We assume that a component does not exist without the composite supertype entity which it is part of. So if the car is deleted from the database, all its parts also disappear. This type of part-relationship is called an exclusive relationship. This relationship of course is similar to the weak entity type relationship and therefore the dominant entity must also provide the identification to the weak component entities.

2.7.3 Extended Entity Relationship Diagrams

The EER Model continues to use rectangles and diamonds for entity and relationship types in the EER diagrams. EER model however also includes representation for constraints which are defined in the model and deals with situations like generalization and specialization.

All supertypes and subtypes may be represented as entity classes even if the level of the hierarchy is more than two. The common attributes are allocated to the supertype while attributes that are particular to a subtype

are allocated to the subtypes. Subtypes and supertypes should participate in relationships. All subtypes and supertypes should be named as other entities are named.

The IsA relationship can also be represented in an E-R diagram. It may be represented as a diamond and is a 1:*m* relationship that has no attributes.

EER Diagram Conventions and Constraints

The following conventions are used in the EER diagrams:

1. The supertype/subtype relationship may be complete if an instance of supertype must also be a member of at least one subtype. This must be shown in the EER diagram by using a double line.
2. The supertype/subtype relationship may be partial if an instance of supertype need not be also be a member of at least one subtype. This must be shown in the EER diagram by using a single line.
3. The subtypes may be disjoint if an instance of supertype may be member of only one of the subtypes. This may be shown in the EER diagram by putting a *D* in the circle.
4. The subtypes may be overlapping if an instance of supertype may be member of more than one of the subtypes. This may be shown in the EER diagram by putting a *O* in the circle.

Figure 2.10 shows an example of a supertype/subtype structure in modeling specialization using an extended E-R diagram. This diagram involves a supertype entity *Student* and three subtype entities *Undergraduate*, *Postgraduate*, and *International*. The diagram shows only few of the attributes of each entity type and the attributes of entity *Student* are inherited by all the subtype entities. The relationships are one-to-one since every instance of a subtype entity must be an instance of the supertype entity.

The diagram shows that not all students need to belong to one of the subtypes since a double line has not been drawn from the entity *student*. Also the diagram shows that overlapping is possible since a student may be in subtype *International* as well as in *Postgraduate*. Furthermore, we note that all attributes of *student* are inherited by the subtypes although each subtype has some attributes that are unique for that subtype.

Figure 2.11 shows another specialization. This diagram implies somewhat different relationships between the supertype entity and the subtype entities than what is shown in Fig. 2.10 because the subtypes are disjoint and each instance of a supertype must belong to one of the subtypes. The relationship between the supertype and subtype is one-to-one.

The double line in the diagram in Fig. 2.11 shows that each computer must be either a desktop or a laptop and no overlapping is allowed. Therefore no computer can be classified as a desktop as well as a laptop. The *D* in the circle in the middle shows that the subtypes are disjoint.

2.7.4 Benefits of Supertypes and Subtypes

Supertypes and subtypes help in dealing with complexity in model building. This may be done by asking questions about the possibility of generalization and specialization during the design process. Similar entity types may be generalized into a supertype while an entity type that has many slightly different entities may be specialized into a number of subtypes.

Supertypes and subtypes may also assist in improving the presentation of an E-R model and in improving communication with the client. The higher level diagrams do not need to include the details while the lower

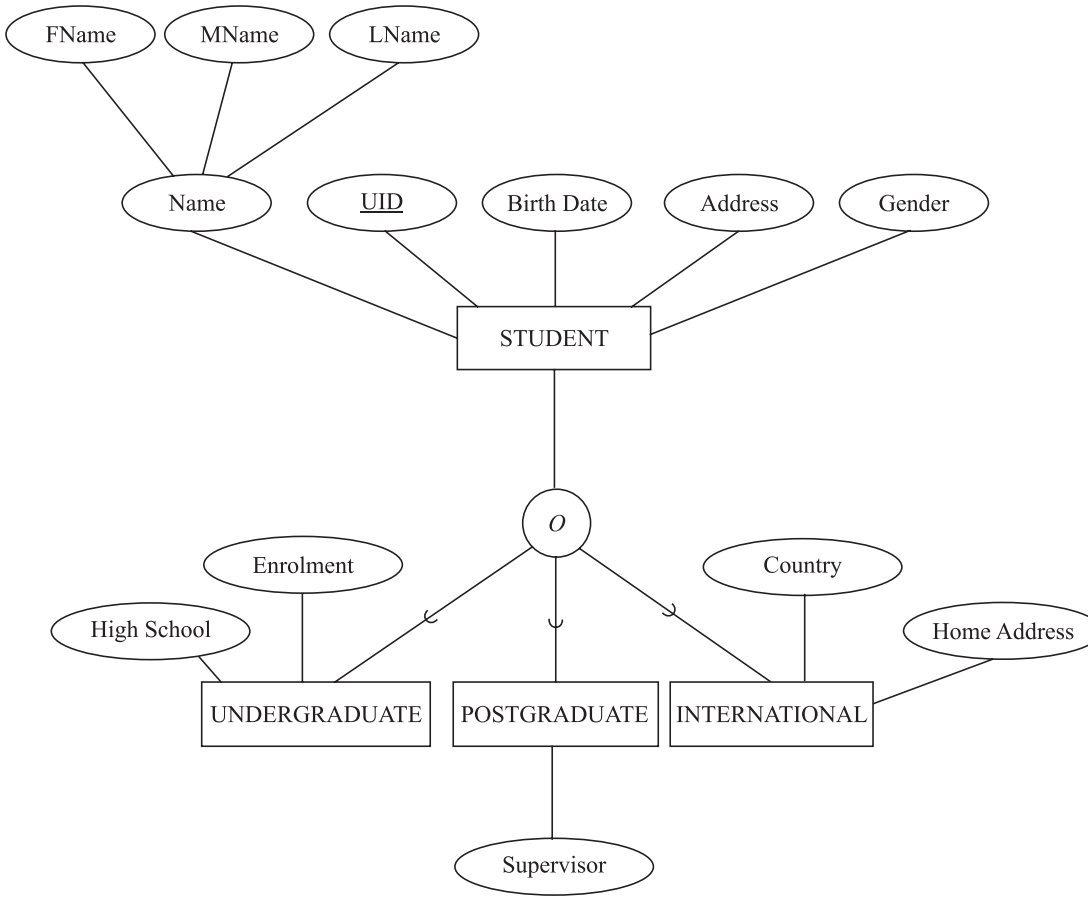

Figure 2.10 EER Diagram showing supertype and subtypes in specialization

level diagrams do. Not all such subtypes and supertypes need to be included in the database implementation; some might be just useful in helping communication.

2.8 THE DATABASE DESIGN PROCESS

As noted in Chapter 1, database design is a complex and challenging process that requires commitment of all the stakeholders including the senior management of the enterprise. In Chapter 1, we noted the following major steps which have now been modified to apply the E-R model approach.

1. *Requirements Analysis*—This step was described in Chapter 1. It involves studying the proposed database application and discussing with the client and other users what is required. A formal set of requirements specifications is prepared and documented.
2. *Identify Entity Sets*—The next step is to use the requirements specifications to identify entity sets that are of interest. This step is independent of the hardware and database software that is to be used.

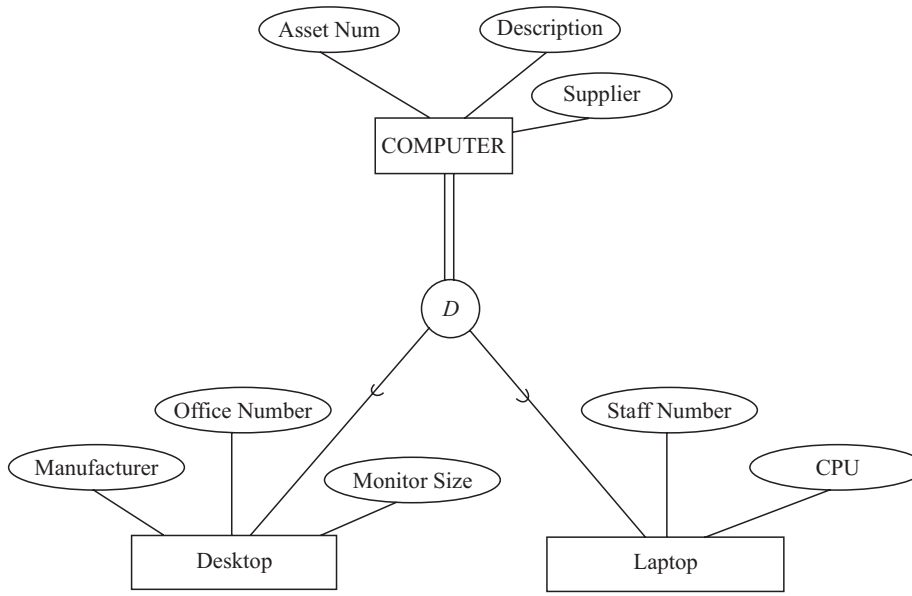

Figure 2.11 EER Diagram showing another specialization

The list of entities is then documented and discussed with the client and revised until an agreement is reached.

3. *Identify Relationship Sets*—Now identify the relationship sets that are of interest. For each relationship set, determine whether the relationship is 1:1, 1:*n* or *m*:*n*. These should now be discussed with the client to ensure that the database designer's understanding of the client's requirements is correct and the proposed database design meets the client's needs.
4. *E-R Diagram*—A E-R diagram is now prepared. The diagram should help the client better understand what the database design plans to implement. The model is revised if necessary until agreement between the database designer and the client is reached.
5. *Value Sets and Attributes*—Value sets and attributes for the entity sets and the relationship sets are now identified. Constraints on the values are identified. These may include domain constraints, integrity constraints and any other enterprise constraints. The model is revised if necessary until agreement is reached with the client.
6. *Primary Keys*—Primary key for each entity set is identified. These should be discussed with the client to ensure the identifiers are reasonable.
7. *Prototype*—A prototype may be developed to help communicate with the client to ensure that the design conforms to the requirements specifications.
8. *Implementation*—The E-R diagram is now mapped to the database schema for the particular DBMS and implemented.

As noted in Chapter 1, database design is an iterative process. Also, there is always the possibility that the client and/or the users will change their mind during the design process or even after thee implementation. The client may suddenly come up with new data and may suggest new applications and propose new processing.

2.9 STRENGTHS OF THE E-R MODEL

The E-R model is a very simple approach to designing a database which considers the real world as consisting of entities and relationships between them. The real world, of course, can be a lot more complex than the simple view used in E-R modeling but the E-R model approach continues to be widely used by database designers because it offers

1. *Simplicity*—As noted above, the approach is simple and easy to use with a client even if the client knows little about computing.
2. *Visual Representation*—An entity-relationship diagram provides a clear, high-level view of the proposed database.
3. *Effective Communication*—As noted above, visual representation of the model provides a clear, high-level view of the proposed database. This makes it easier to communicate with the client.
4. *Ease of Mapping to the Relational Model*—The E-R model was developed with the relational model in mind and the entity and relationship constructs map easily to the relational model as we will find in the next chapter.
5. *Focus on the Model*—The E-R model allows the database designer to focus on the complete and accurate modeling of the enterprise without having to think about efficiency of implementation. The efficiency issues should be considered in the implementation phase of database design.

2.10 WEAKNESSES OF THE E-R MODEL

As noted earlier, the E-R model is a very simple approach to designing a database. The simple approach is not always suitable for modeling database situations because of the following

1. Weaknesses of modeling in general. The database designer must understand the pitfalls of modeling since the model is not real. It is sometimes possible for people to forget that a model is only an abstraction of the reality, it is not reality itself. In fact, a model is often incomplete and could even be considered a distorted view of reality. If the database designer and the users after the system is operational start believing that the model is perfect then it can lead to a variety of problems. For example, an employee of the enterprise may insist a customer with somewhat unusual circumstances that the customer is wrong since the information in the database does not match with what the customer is saying. It could well be that the customer is right and in the unusual case the database is not correct. Just as, in physics, all laws are supposed to be hypotheses that can become invalid as soon as a better hypothesis is found. A database model is also a kind of hypothesis that could be replaced by a better model and a better hypothesis.

   Furthermore, no modeling technique is perfect. All modeling techniques have a number of weaknesses. This again means that a database model cannot fully capture a real life enterprise which can be very complex.
2. Although we have discussed the concept of an entity and presented a simple definition, a rigorous definition is not possible since there is no absolute distinction between entity types, relationship types

and even attributes. For example, an entity has been defined in many different ways. A sample of definitions from the literature are given below to illustrate the confusion.

- *An entity is a thing that can be distinctly identified*
- *Entities are things that exist independently of other entities*
- *An entity means anything about which we store information*
- *An entity is a person, place or thing about which we store information*

Even in his original 1976 paper, Chen notes that *'some people may view something (e.g. marriage) as an entity while other people may view it as a relationship. We think that this is a decision which has to be made by the enterprise administrator. He should define what are entities and what are relationships so that the distinction is suitable for his environment.'*

Therefore, it is not always easy to decide when a given thing should be considered an entity. For example, in the employee database, one may have the following information about each employee:

- Employee number
- Name
- Address
- Date of birth
- Citizenship
- Telephone number
- Salary

This is only a very small list of likely information that a company would want to store for each of its employees. Assume that there is an additional attribute called residence city. Although it is assumed that city is an attribute, it may be that city should really be an entity. The rule of thumb that should be followed in such situations is to ask the question "Is city an object of interest?" If the answer is yes, there must be some more information about each city in addition to the city name and then city should be considered an entity. If however we are only interested in the city's name, city should be considered an attribute of employee.

As noted earlier, each entity should contain information about its properties. If an object has no information other than its identifier that is of interest, the object should be an attribute but this is not always possible as we show later in this section.

3. A relationship type normally shows an association between two or more entity types but relationships are allowed to have attributes which makes them look similar to entity types. There is a distinction in that a relationship is unable to exist on its own. For example, a relationship *Works_on* cannot be expected to exist on its own without the entity sets *Employee* and *Project*. Nevertheless, a number of experts, including E. F. Codd, have noted that one person's entity can legitimately be regarded as another person's relationship. This flaw makes modeling using the E-R approach a bit of an art.
4. Although an attribute exists only as a property of an entity type, in some contexts an attribute can also be viewed as an entity. For example, some attributes like address, telephone number and some time even name can have several values. Address may be current address as well as permanent address. Telephone number may include business phone, home phone and mobile phone. Some people even carry more than one mobile phone. Consider a name like B. Ramamuralikrishna where B stands for place of birth, Bangalore. Assume that the person gets tired of using such a long name and starts using

B. R. Krishna or B. R. M. Krishna. A database may need to include all these versions of the name. How should one store them as attributes?

The guideline for multivalue attributes in entity-relationship is that such an attribute should be considered an entity. Although conceptually multivalue attributes create no difficulties, problems arise when the E-R model is translated for implementation using a relational database system discussed in Chapter 3. As an example, suppose we have an entity called project in a company database and one of the attributes is funding code. If the database model allows a number of funding codes for one project then we have a difficulty and it might be best to have another entity called funding code and then have a relation between the entity type project and the entity type funding code.

Although we have indicated above that an entity set should normally contain information about its properties, a multivalue attribute even if it has no properties other than its value has to be considered an entity.

5. Complexity. E-R models for nontrivial database applications can become rather complex since entity sets tend to have many attributes which contribute to the complexity of the model. The difficulty can be overcome by using a top-down approach so that the model is displayed at several different levels of abstraction with the top level just showing entities or even groups of entities and with the details provided by models at lower levels.

   Finally, the E-R model offers limited constraint representation, limited relationship representation, no representation of data manipulation and in some cases results in loss of information. A detailed discussion of weaknesses of the E-R model is presented in the book by Thalheim.

2.11 A CASE STUDY OF BUILDING AN E-R MODEL

We now consider a nontrivial example of building an E-R model. In this example we will attempt to model the information that is generated in a cricket ODI match. We cannot fully model the information generated in an ODI match so our model will be quite limited in scope.

The primary aim in presenting this long example is to provide the reader some understanding about the complex art and science of model building. For this reason the example will include early models with a variety of errors in them. The models will then be improved to hopefully overcome all the problems identified. We emphasise that no solution for a significantly complex problem is unique and there is never a perfect solution.

Normally, an ODI match information is presented as a scorecard as described in the Tables 2.7 to 2.13 below. The information in this example comes from an ODI match played between Australia and India in Sydney in March 2008. India won this match. We present a part of the scorecard which shows the batting and bowling performances of both innings. Surprisingly, there is no standard definition of a scorecard. For example, some may include the number of minutes a batsman was at the crease while others do not, but we assume that the scorecard given below is acceptable. *R* in the scorecard stands for number of runs scored by the batsman, *M* for the number of minutes the player was batting, *B* is the number of balls the batsman faced and 4s and 6s are the number of fours and sixes scored. *SR* is the strike rate obtained by dividing *R* by *B* (converted to a percentage).

Table 2.7 gives details of the Indian inning's scorecard.

Table 2.7 Indian inning's scorecard (Batting first, 50 overs maximum)

One Day International (ODI) Match held in Sydney, March 2008

Batsman	*How out?*	*R*	*M*	*B*	*4s*	*6s*	*SR*
RV Uthappa	c Hopes b Clark	30	99	49	1	0	61.2
SR Tendulkar	c Ponting b Clarke	91	176	121	7	0	75.2
G Gambhir	c Johnson b Clarke	15	21	16	1	0	93.8
Yuvraj Singh	c Hayden b Symonds	38	27	38	2	2	100
MS Dhoni	c Clarke b Bracken	36	52	37	2	1	97.3
RG Sharma	c Symonds b Clarke	2	6	5	0	0	40
IK Pathan	b Bracken	12	24	20	1	0	60
Harbhajan Singh	lbw b Lee	3	7	3	0	0	100
P Kumar	c Ponting b Bracken	7	7	7	1	0	100
PP Chawla	not out	6	8	2	1	0	300
S Sreesanth	not out	1	4	2	0	0	50
Extras	**(lb 5, w 12)**	**17**					
Total	**(9 wickets; 50 overs; 220 mins)**	**258**	**(5.16 runs per over)**				

Table 2.8 gives details of the fall of wickets for the Indian inning given in Table 2.7.

Table 2.8 Fall of wickets in the Indian inning

Fall of wickets: 1-94 (Uthappa, 20.5 ov), 2-121 (Gambhir, 25.2 ov), 3-175 (Yuvraj Singh, 34.6 ov), 4-205 (Tendulkar, 39.2 ov), 5-209 (Sharma, 41.1 ov), 6-237 (Pathan, 47.2 ov), 7-240 (Dhoni, 47.6 ov), 8-249 (Harbhajan Singh, 48.5 ov), 9-255 (Kumar, 49.3 ov)

Table 2.9 gives details of the Australian bowling in the Indian inning in the ODI match.

Table 2.9 Australian bowling in the Indian inning

Bowling	*O*	*M*	*R*	*W*	*Econ*
B Lee	10	0	58	1	5.8
NW Bracken	9	1	31	3	3.44
SR Clark	6	0	32	1	5.33
MG Johnson	6	0	33	0	5.5
JR Hopes	6	0	20	0	3.33
MJ Clarke	10	0	52	3	5.2
A Symonds	3	0	27	1	9

Looking at the scorecard we find that in the bowling figures, *O* stands for number of overs, *M* stands for number of overs in which no runs were scored (called maidens), *R* stands for the number of runs scored by the batsmen off the bowler's bowling. Finally, *W* is the number of players (wickets) that the bowler dismissed during the innings and *Econ* is the economy rate which is equal to *R*/*O*.

Table 2.10 gives details of the Australian inning's scorecard.

Table 2.10 Australian innings (Target: 259 runs from 50 overs)

Batsman	*How out?*	*R*	*M*	*B*	*4s*	*6s*	*SR*
AC Gilchrist	c Dhoni b Kumar	2	3	3	0	0	66.7
ML Hayden	run out (Yuvraj Singh/Harbhajan Singh)	55	122	68	7	0	80.9
RT Ponting	c Yuvraj Singh b Kumar	1	9	7	0	0	14.3
MJ Clarke	b Kumar	17	29	22	1	0	77.3
A Symonds	lbw b Harbhajan Singh	42	81	56	2	1	75
MEK Hussey	c Dhoni b Sreesanth	44	66	42	3	0	105
JR Hopes	c Chawla b Pathan	63	105	80	4	1	78.8
B Lee	b Kumar	7	20	12	0	0	58.3
MG Johnson	c Dhoni b Sreesanth	8	9	6	1	0	133
NW Bracken	c Chawla b Pathan	1	7	2	0	0	50
SR Clark	not out	0	2	0	0	0	-
Extras	**(lb 2, w 7)**	**9**					
Total	**(all out; 49.4 overs; 230 mins)**	**249**	**(5.01 runs per over)**				

Table 2.11 gives details of the fall of wickets in the Australian inning.

Table 2.11 The Fall of wickets in the Australian inning

Fall of wickets: 1-2 (Gilchrist, 0.3 ov), 2-8 (Ponting, 2.3 ov), 3-32 (Clarke, 8.6 ov), 4-121 (Hayden, 25.4 ov), 5-123 (Symonds, 25.6 ov), 6-199 (Hussey, 41.6 ov), 7-228 (Lee, 46.4 ov), 8-238 (Johnson, 48.2 ov), 9-247 (Bracken, 49.2 ov), 10-249 (Hopes, 49.4 ov)

Table 2.12 gives details of the Indian bowling in the Australian inning.

Table 2.12 Indian bowling in the Australian inning

Bowling	*O*	*M*	*R*	*W*	*Econ*
P Kumar	10	2	46	4	4.6
S Sreesanth	9	0	43	2	4.77

Contd.

Table 2.12 Contd.

Bowling	*O*	*M*	*R*	*W*	*Econ*
IK Pathan	8.4	0	54	2	6.23
Harbhajan Singh	10	0	44	1	4.4
PP Chawla	9	0	45	0	5
Yuvraj Singh	3	0	15	0	5

There is more information about a match that one may want to include in the database. Where was the match played? Was it a day match or a day-night match? On which date was the match played? Who were the umpires? Who was the match referee? Who was the TV umpire? Who won the toss? What was the result? Who won the Player of the Match award? Some may even want to know how many people attended the match. Also, one might want to include who the 12th man was and who the substitute fielder was, if substitutes were needed. There is also ball-to-ball information that could be included in an ODI database. There is thus an enormous amount of information that is generated in each ODI match. Some of this information is listed in Table 2.13.

Table 2.13 Other information about the ODI

• Toss: India won and chose to bat first
• Series: India won the best-of-3-finals 2-0
• Player of the match: P Kumar (India)
• Player of the series: NW Bracken (Australia)
• Umpires: AL Hill (New Zealand) and SJA Taufel
• TV umpire: BNJ Oxenford
• Match referee: JJ Crowe (New Zealand)
• Reserve umpire: TP Laycock

We will now try to simplify the above information for our case study. We don't want it to be so complex that the model becomes so large that it cannot be used in this book. The simplifications listed in Table 2.14 are made with a view that they do not reduce the case study to a toy example.

Table 2.14 List of assumptions made in designing the ODI database model

1. The number of extras made is not considered.
2. The date the match was played is not considered.
3. Questions like who were the umpires, who was the match referee and who was the TV umpire are ignored.
4. The toss result is not considered.
5. The Player of the Match award is not considered.
6. No information is stored about the number of people who attended the match.
7. No information about the 12th man and substitute fielders is considered in our model.

This list of assumptions is perhaps not complete. We should add any other assumptions we discover to this list.

The situation is still quite complex. First, all the information about fall of wickets (FOW) given above appears unstructured. We will now structure it so that we can understand it better. To do so we transform the given information to a tabular form as given in the two Tables 2.15 and 2.16, for the two innings. This information is now easier to comprehend and model.

Table 2.15 Fall of wickets in the Indian inning

FOW	*Player out*	*Score when out*	*Over when out*
1	Uthappa	1/94	20.5
2	Gambhir	2/121	25.2
3	Yuvraj Singh	3/175	34.6
4	Tendulkar	4/205	39.2
5	Sharma	5/209	41.1
6	Pathan	6/237	47.2
7	Dhoni	7/240	47.6
8	Harbhajan Singh	8/249	48.5
9	Kumar	9/255	49.3

The column *Score when out* is given as scores like 1/94, which means the score was 94 with one wicket down. It should be noted that the information on how many wickets were down is redundant but we will not deal with it at this time. The column *Over when out* has information like 20.5, which means that the player was out on the fifth ball of the 21st over.

Table 2.16 Fall of wickets in the Australian Innings

FOW	*Player out*	*Score when out*	*Over when out*
1	Gilchrist	1/2	20.5
2	Ponting	2/8	25.2
3	Clarke	3/32	34.6
4	Hayden	4/121	39.2
5	Symonds	5/123	41.1
6	Hussey	6/199	47.2
7	Lee	7/228	47.6
8	Johnson	8/238	48.5
9	Bracken	9/247	49.3
10	Hopes	10/249	49.4

Example Requirement

We want to look at possible ways of modeling the above ODI match information. We want all the information in a scorecard to be part of our model including fall of wickets and bowling performance. How do we model it?

Option 1

What is the simplest model that we can design for the problem? What are the entities? What are the relationships? One possibility is to try to model the match information in a similar way to what is presented in the scorecard. That is, we have an entity *Match* that provides some basic information about the match. The

batting performance can be represented by an entity *Batsman* and the bowling performance by *Bowler* and the fall of wickets by an entity *FOW*. The entities and relationships are then as follows:

Entities

The entities in the current E-R model are given in Fig. 2.12

Match
Batsman
Bowler
FOW

Figure 2.12 List of entities

Relationships

Figure 2.13 gives the list of relationships between the entities listed in Fig. 2.12.

M_Batting – between Match and Batsman
M_Bowling – between Match and Bowler
M_FOW – between Match and FOW

Figure 2.13 List of relationships

The E-R model then looks like that shown in the E-R diagram in Fig. 2.14.

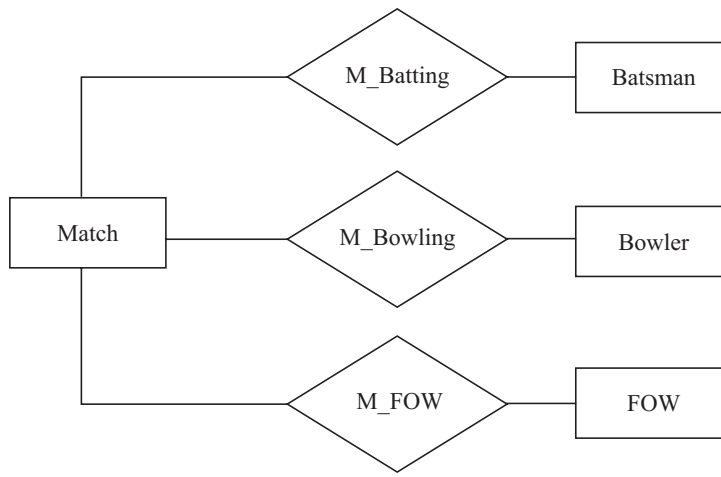

Figure 2.14 Option 1: E-R model for the ODI match database

We now list attributes for each entity.

Match

Match needs to have the attributes listed in Fig. 2.15.

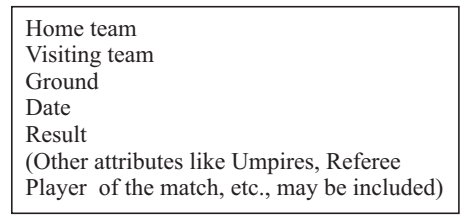

Figure 2.15 List of attributes of *Match*

Batsman

Batsman needs to have the attributes listed in Fig. 2.16.

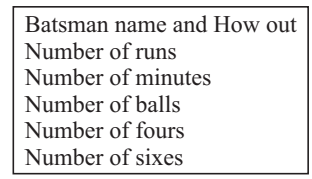

Figure 2.16 List of attributes of *Batsman*

Bowler

Bowler needs to have the attributes listed in Fig. 2.17.

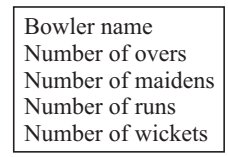

Figure 2.17 List of attributes of *Bowler*

FOW

FOW needs to have the attributes that are listed in Fig. 2.18.

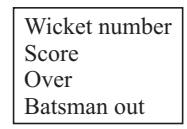

Figure 2.18 List of attributes of *FOW*

Note that we have not included information about the strike rate for batsmen or economy rate for bowlers since these can be computed given the other information.

All the relationships are one-to-many and we assume that there are no attributes for each of the three relationships. These relationships can therefore be represented by migrating the *Match* primary key to all the other relations. This concept will be discussed in detail in the next chapter.

Let us now discuss what is wrong with this design. Readers are encouraged to list all the mistakes in this design before reading Option 2 below.

Option 2

The above design has many problems. Here is a list of some of them:

1. Although *Match* has a primary key (consisting of the attributes Home Team, Visiting Team and Date), the other entities do not.
2. None of the entities includes the country that the player is representing.

To fix the first problem regarding primary keys we introduce an artificial identifier in the *Match* entity. We will call it the Match ID. Artificial identifiers may be included in other entities as well but we will fix the primary key problem in the remaining entities in another way. The second problem is solved by including the country attribute where needed. We now obtain the following model:

Match ID Home Team Visiting Team Ground Date Result (Other attributes, like Umpires, Referee, Player of the match, etc. may be included)

Figure 2.19 List of attributes of *Match*

Match

Match now has the attributes listed in Fig. 2.19 (Primary key is Match ID).

Batsman

Batsman now has the attributes listed in Fig. 2.20 (Primary key is Match ID, Player order. Why?).

Match ID Player order Batsman name and How out Country Number of runs Number of minutes Number of balls Number of fours Number of sixes

Figure 2.20 List of attributes of *Batsman*

Bowler

Bowler now has the attributes listed in Fig. 2.21 (Primary key is Match ID and Bowler name, assuming the name is unique).

Match ID Bowler name Country Number of overs Number of overs Number of maidens Number of runs Number of wickets

Figure 2.21 List of attributes of *Bowler*

FOW

FOW (Primary key is Match ID and Wicket number)

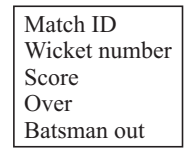

Figure 2.22 List of attributes of *FOW*

As noted earlier, the primary keys that we have identified in *Batsman*, *Bowler* and *FOW* are not the only way to identify each entity instance in these entities. We could have introduced artificial identifiers in each entity just as we did for the entity *Match*.

Option 3

Option 2 provides a reasonable solution but there are still problems with it because we have essentially copied the printed version of the scorecard and translated it into entities and relationships. It is possible to do better.

First of all do we really need an entity *FOW*? Most of this information already exists is in the entity *Batsman*. The only information missing is the score at which each batsman got out. We could include that information in the entity *Batsman* and we can then derive the information that is given in the *FOW* table. So the *Batsman* entity now has the attributes that are listed in Fig. 2.23.

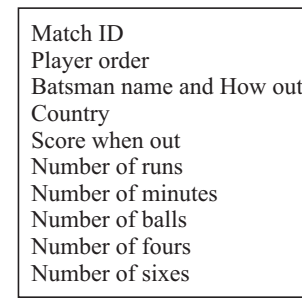

Figure 2.23 List of attributes of *Batsman*

There is no change in the entity *Match* or *Bowler* but the entity *FOW* is now not needed.

Perhaps a more serious problem still remains. This has to do with the attribute 'Batsman Name and How out' in the *Batsman* entity. Given that this attribute has values like 'SR Tendulkar c Ponting b Clarke' leads to the following difficulties:

- Each attribute value is considered atomic and it is then assumed that different components of the value cannot be looked at separately.
- The names Tendulkar, Ponting and Clarke are not unique because they are presented in a form suitable for people to read them and not for computer processing. For example, Ponting is written as RT Ponting in the batting performance of the Australian team. Such differences make it difficult to process this information while ensuring that Ponting and RT Ponting refer to the same player.
- Other problems may arise in uniquely identifying players if, for example, the player Sachin Tendulkar is included in the scorecard as SR Tendulkar.

These problems are not really serious if our model is simply designed to print the scorecard as is printed in the newspapers because our current model can do that. However, a question now arises and should be asked: 'Do we need, or may need in the future, to answer a query like how many runs were scored by MS Dhoni in the March 2008 ODI in Sydney?'

Option 4

To be able to uniquely identify players, we need to have a unique identifier for each player. We can introduce an attribute called Player ID for each player and if we wanted to be able to search on a player's last name then we could establish an entity called *Player*.

Let us assume the entity *Player* includes the attributes listed in Fig. 2.24.

Player ID
Player last name
Player first name
Country
When born
Place of birth
Year of first test

Figure 2.24 List of attributes of *Player*

The *Batsman* entity's attributes will change to those listed in Fig. 2.25.

Match ID
Player order
Player ID
How out
Score when out
Number of runs
Number of minutes
Number of balls
Number of fours
Number of sixes

Figure 2.25 List of attributes of *Batsman*

The *Bowler* entity will also change to as given in Fig. 2.26

Match ID
Player ID
Number of overs
Number of maidens
Number of runs
Number of wickets

Figure 2.26 List of attributes of *Bowler*

The *Match* entity will not change.

Option 5

Let us now critically review the solution. Clearly *Match* and *Player* need to be entities. Do *Batsman* and *Bowler* need to be entities? Could they be relationships between the entity *Player* and the entity *Match* since

each batting and bowling performance is related to a player and an ODI match? Making them relationships changes the E-R model to the simple model displayed in Fig. 2.27.

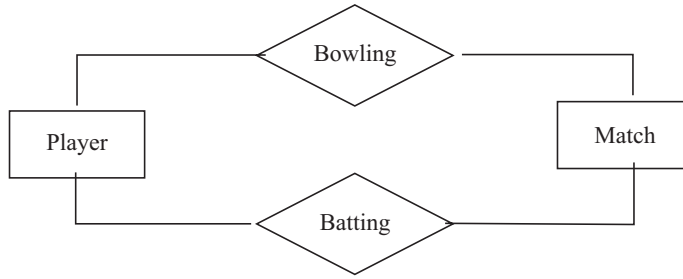

Figure 2.27 E-R diagram for Option 5

The two relationships are many-to-many since each player bats and bowls in many matches and each match has more than one batsman and bowler.

There is no change in the list of attributes for the entities *Player* and *Match*. The attributes for the relationships are given below:

The *Batting* relationship's attributes will change to those listed in Fig. 2.28.

Player ID
Match ID
Player order
How out
Score when out
Number of runs
Number of minutes
Number of balls
Number of fours
Number of sixes

Figure 2.28 List of attributes of *Batting*

The *Bowling* relationship's attributes will also change to those listed in Fig. 2.29.

Player ID
Match ID
Number of overs
Number of maidens
Number of runs
Number of wickets

Figure 2.29 List of attributes of *Bowling*

How do we find out *FOW*?

We leave this as the final solution. Although some problems remain and further modifications will be required.

Final Critique

As noted there are many problems with the final solution. It only shows that modeling is quite a challenging task. Let us list some of the problems that we are able to identify:

1. No information on extras, byes, leg byes, wides and no balls is included.
2. No information on the 12^{th} man and substitutes is included.
3. No information is included about batsmen who did not bat in an innings.
4. In fact it is not possible to list the teams that played in a given match.
5. It is not possible to represent who was the captain of the team in a match and who was the wicketkeeper.
6. The total runs scored in an innings may be derived but is not included as an attribute.
7. The 'How out' attribute does not provide sufficient information to print the scorecard.
8. The Duckworth–Lewis (or D/L) system of deciding a match winner when the game is interrupted by rain and the second team is not able to bat for the same number of overs as the first team is not considered.
9. More ODI matches are now being played at neutral venues (for example, Kuala Lumpur or Dubai). There are also triangular ODI series where two visiting teams play each other. These issues are not considered in our model.

We leave it to the reader to list other weaknesses of the final model and to attempt to improve the model.

2.12 EVALUATING DATA MODEL QUALITY

As we have noted, building a conceptual model of an enterprise can be very complex. Building a data model can be more of an art than a science. Once a model has been built, one may want to evaluate the quality of the model and validate how faithfully the model represents the enterprise that is being modeled. If more than one model is available then how does one select the best model?

Evaluating model quality is a difficult issue. Researchers who have investigated the issue of model quality have often proposed that the quality be judged based on a list of desirable properties. Unfortunately these lists almost always lack any fundamental basis and are often based on a person's modeling experience. Another difficulty with such lists of desirable properties is that the properties are often quite subjective and difficult to measure. For example if we consider readability to be an important quality feature then how can we measure readability so as to conclude that one model is more readable than another?

Nevertheless, a list is useful in providing some guidance on what a modeler should keep in mind while developing a model. We present a consolidated list based on some of the references cited at the end of this chapter. The properties include many commonsense syntactic and semantic properties that one would expect of high quality models. Firstly, one expects that the model be simple and easily understandable by the users of the model. For example, it is expected that the names of the entities and relationships reflect enterprise knowledge and therefore be self-documenting. The model has to be syntactically correct, that is, the model constructs have not been used incorrectly; for example, a construct using a relationship between an existing relationship and an entity has not been used. The model must be complete in that every piece of information

that is required in the database has been included in the model and no unnecessary information has been included. Minimality and nonredundancy require that information normally be included only once. If subclass and superclass structures are used in the model then each subclass must have some property that is not in the superclass for a subclass to be justified. Given that almost every computer system is modified during its lifetime as enterprise requirements change, some suggest that it is desirable that the model be flexible and extendible so that the model is adaptable to changing enterprise requirements, although it is not quite clear how one could judge if a given model is indeed flexible.

Table 2.17 List of data model quality criteria

• Completeness
• Correctness
• Consistency
• Minimality
• Readability
• Self-explanation
• Extensibility
• Performance

Table 2.17 provides a list of quality criteria.

These criteria can be described as given below:

1. *Completeness*—A good model should be complete as far as the client's requirements are concerned. Assessing completeness should be possible with the support of the client as well as assistance from a number of users who understand the proposed model well. An external expert may also be required. The temptation to incorporate requirements beyond what has been agreed to should be curbed.
2. *Correctness*—A good model should model all aspects of the proposed database system as well as business rules of the enterprise. Assessing correctness is not subjective but would require assistance of the client and may require assistance from an expert with knowledge of the database application as well as of database systems.
3. *Consistency*—A good model must not have any contradictions. Consistency, correctness and completeness are the three basic requirements of any good model. Consistency is perhaps more technical than correctness and completeness, although correctness and completeness can also be quite technical. The client may not be able to assist much in checking consistency. Assistance of a number of users who understand the proposed model well may be required. An external expert may also be required.
4. *Minimality*—A good model ought to be compact and should not include redundancy. To assess minimality it is necessary to have assistance from a number of users who understand the proposed model well. An external expert who has good knowledge of data modeling may also be required.
5. *Readability*—A good model should not only be understandable to users who have been involved in the database design but also by users who have not been involved (for example, by a user who takes over the responsibility of the database system after the data modeler is gone and the system has been implemented). Readability can be a somewhat subjective quality. The client's views about readability may also be useful.
6. *Self-explanation*—Related to readability, this criterion proposes for entity, relationship and attributes' names to be such that users can easily understand them. The client's views about self-explanation may be useful. Users may also be consulted since some of them are going to be unsophisticated users and their ability to understand the model is important.

7. *Extensibility*—A good model should be able to accommodate business changes because enterprise requirements change as the enterprise changes but it is almost impossible to judge extensibility without knowing what kind of extensions might be required.
8. *Performance*—Although a data model is developed without considering its performance, it should be possible to make some changes to a model without impacting the model's logical structure.

There are a number of other properties that some experts suggest should be added to this list. These include simplicity, validity, flexibility, understandability, implementability, expressiveness, normality (or normalization), scope, reusability, transformability and levels of detail. We do not discuss these since some of them overlap with the properties we have already described above. For example, are simplicity and understandability similar to readability and self-explanation? Is flexibility similar to extensibility?

For a good discussion of quality in database modeling, refer to Chapter 6 of the book by Batini, Ceri and Navathe.

SUMMARY

In this chapter we have discussed the following topics:

- Models have been used by people for a long time, for example, by architects, scientists and engineers.
- There are many benefits of modelling before building including focussing on the essentials, product or process improvement and exploring alternatives.
- Models can be of three different types, viz., descriptive, prescriptive or representative.
- Database modelling has at least two phases. The first phase involves building an enterprise conceptual model. In the second phase, the conceptual model is mapped to the database model to be used.
- A simple approach of database modeling is called the Entity-Relationship modeling (ERM).
- ERM involves identifying entities (like a person, a player or a book), relationships (like batting, marriage) and attributes (like age, address).
- Relationships may be unary (e.g., employee and supervisor), binary (like employee and the company) or ternary (like employee, project, and department).
- Binary relationships may be 1-1, 1-*m* or *m*-*n*.
- Attributes may be of several types including simple or composite, single-valued or multivalued, stored or derived.
- An E-R model may be represented as Chen's E-R diagram or as Crow's Foot diagram.
- Different notations are used for both types of diagrams. In the E-R diagram a rectangle represents an entity and a diamond a relationship. In the Crow's Foot notation, lines joining entities represent relationships.
- A number of examples have been used to illustrate the diagrams. One of the examples is about employee-department-project.
- All entities and relationships must have a primary key that identifies each instance of the entity set and each entity of a relationship set.
- The concepts of generalization, specialization and aggregation have been explained.

- Extended Entity Relationship modeling and diagrams have been presented.
- Strengths and weaknesses of the ERM have been described. The strengths include the simplicity of the approach while one of the weaknesses is the difficulty of precisely defining an entity.
- A case study to model ODI scorecard is described to illustrate the ERM is presented.
- Finally data quality of a model was discussed and a number of criteria, for example, completeness, correctness and consistency were discussed.

REVIEW QUESTIONS

1. Describe some fields in which modelling is used successfully. (Section 2.2)
2. What are different types of models? (Section 2.3)
3. Describe the phases of database modeling. (Section 2.4)
4. What is an Entity-Relationship model? (Section 2.5)
5. What is an entity and an entity set? (Section 2.5.1)
6. List some naming conventions for an entity and a relationship. (Section 2.5.1)
7. What is a relationship between entities? What are their roles? (Section 2.5.2)
8. Explain, 1-1 relationship, 1-*m* relationship and *m*-*n* relationship? (Section 2.5.2)
9. What is cardinality ratio? (Section 2.5.2)
10. What is an attribute and value sets? (Section 2.5.3)
11. Explain Chen's an Entity-Relationship diagram? (Section 2.5.5)
12. What are ternary relationships? Case a ternary relationship be converted into a number of binary relationships. (Section 2.5.6)
13. What is the purpose of generalization, specialization and aggregation? (Section 2.6)
14. Give two examples of additional features of the Extended Entity-Relationship model? (Section 2.7)
15. List the steps that should be taken in the database design process. (Section 2.8)
16. What are the strengths of the Entity-Relationship model? (Section 2.9)
17. What are the weaknesses in database modeling? (Section 2.10)
18. How can one evaluate an Entity-Relationship model's quality? (Section 2.12)

SHORT ANSWER QUESTIONS

1. What is an entity?
2. What is a relationship?
3. Name three entities that you know should be part of any university database.
4. Name the primary key for each of the entities.
5. Name a superkey of each of the three entities.
6. Name three attributes of each entity you have identified.
7. Identify any relationships that exist between the three entities you have identified.
8. Name the primary key of each of the relationships.

9. List the phases of database modeling.
10. Give examples of 1-1 relationships, 1-*m* relationships and *m*-*n* relationships?
11. What symbol represents a *m*:*n* relationship in a E-R diagram?
12. What is a superkey? How does it differ from a primary key?
13. What is cardinality ratio?
14. Give an example of an Entity-Relationship diagram?
15. What are ternary relationships?
16. Can every ternary relationship be represented as a set of binary relationships without adding any more entities?
17. Give an example of a binary relationship by showing an E-R diagram.
18. Give an example of three entities and a ternary relationship by drawing an E-R diagram.
19. Give an example of a unary relationship. Draw its diagram.
20. List some naming conventions for an entity.
21. Give an example of an extended Entity-Relationship model?
22. Explain the steps that should be taken in the database design process.
23. List two major weaknesses of the Entity-Relationship model.

MULTIPLE CHOICE QUESTIONS

1. Which of the following professionals build models in their profession?
   (a) Teachers (b) Architects (c) Engineers (d) Computer architects
   (e) All of the above
2. Which of the following are benefits of modeling?
   (a) It helps focus on the essentials. (b) It helps communication and understanding.
   (c) It helps explore alternatives. (d) It can help product or process improvement.
   (e) All of them
3. Models may be classified into a number of classes. Which one of the following is **not** one of these classes?
   (a) Descriptive (b) Prescriptive (c) Representative (d) Prototype
4. Which one of the following is **not** correct?
   (a) The E-R model was first proposed by E. F. Codd.
   (b) The E-R model is a popular high-level conceptual data model.
   (c) The E-R model is used for conceptual design of database applications.
   (d) The major strength of the E-R model is its simplicity and readability.
5. E-R modeling is part of the database design process. Which one of the following the E-R modeling process does **not** include?
   (a) Requirements analysis (b) Conceptual design
   (c) Logical design (d) Security and authorization design
6. Which one of the following is most **unlikely** to be an entity?
   (a) A person (b) A job or position (c) A travel plan (d) An address

7. Which two of the following are most **unlikely** to be relationships?
   (a) A lecture (b) An enrolment (c) A city (d) A shipment
   (e) A date
8. Which one of the following is **unlikely** to be an attribute?
   (a) Location (b) Enrolment (c) Address (d) Date of birth
9. Which one of the following is **not** included in the classical E-R model?
   (a) Entities (b) Relationships
   (c) Integrity constraints (d) Attributes
10. Which one of the following is **not** correct?
    (a) Each entity must have attributes. (b) Each relationship must have attributes.
    (c) Some attribute or attributes must be able to uniquely identify each entity instance.
    (d) Identifiers of the entities participating in the relationship are sufficient to uniquely identify each relationship instance.
11. Which of the following are correct?
    (a) Relationships represent real-world associations among one or more entities.
    (b) Relationships have a physical or conceptual existence other than that inherited from their entity associations.
    (c) Relationships are described in terms of their degree, connectivity and cardinality.
    (d) Each entity participating in a relationship plays a role.
    (e) All of the above are correct.
12. Depending on the nature of an attribute, it may belong to one of the pair of types below. Which of the following pair could **not** be considered to be an attribute type?
    (a) Simple or composite (b) Single-valued or multivalued
    (c) Stored or derived (d) Null or non-null
    (e) All of them
13. Which one of the following is a correct notation in E-R diagrams?
    (a) Entities are ovals (b) Relationships are rectangles
    (c) Attributes are diamonds (d)Weak entities are double rectangles
14. Which of the following is **not** correct?
    (a) The E-R model helps to avoid looking at the details and concentrate on the nature of the problem.
    (b) The E-R model avoids looking at database efficiency.
    (c) The E-R model helps carry out communication with the client.
    (d) The E-R model is the only technique available for database design.
15. Which two of the following are **not** correct?
    (a) A ternary relationship can sometimes be represented by two binary relationships.
    (b) Any two entities may have any number of binary relationships between them.
    (c) Generally attributes of relationships are assigned only to many-to-many relationships and not to one-to-one or one-to-many relationships.
    (d) Each entity participating in a relationship must have a relationship instance for each entity instance.

16. Which one of the following is correct?
    (a) All relationships may be converted to binary relationships.
    (b) All relationships may be converted to 1:1 relationships.
    (c) All relationship attributes may be attached to one of the participating entities.
    (d) All relationships may be represented by a table in a database.
17. For each of the following relationships, indicate the type of relationship (i.e., 1:1, 1:*m*, *m*:*n*).
    (a) Enrolment (a relationship between entities student and subject)
    (b) Works in (a relationship between entities department and staff)
    (c) Marriage (a relationship between entities person and person)
    (d) Manager (between employee and department)
    (e) Dependent (a relationship between entities person and person)
18. Which one of the following comments about the Entity-Relationship model is correct?
    (a) Differences between entity and attribute are always clear.
    (b) The procedure of identifying entities and attaching attributes to them is an iterative process.
    (c) Differences between an entity and a relationship are always clear.
    (d) Differences between an attribute and a relationship are always clear.
    (e) None of the above
19. Consider two E-R models about courses, lecturers and time of lecture. Each course has only one lecturer while each lecturer teaches several courses. Model A consists of two entities and one relationship joining them. The entities are lecturer and course and the relationship is lectures. Model B consists of three entities; the first and the third are the same as above but the second entity is called lecture. The first and second entity are joined by a relationship called 'gives' while the second and the third entities are joined by a relationship called 'of'.
    Which of the following are correct?
    (a) Both models allow a course to have more than one lecture from the same lecturer.
    (b) Model B is more appropriate if information about all lectures, past and present, is to be stored.
    (c) Model A does not allow lecture date and time to be stored.
    (d) Both models provide exactly the same information.
20. Which one of the following is correct?
    (a) A ternary relationship must either be 1:1:1, 1:1:*m* or *p*:*m*:*n*.
    (b) 1:1:1 ternary relationships exist commonly.
    (c) Crow's foot notation makes the E-R diagram more compact.
    (d) It is not possible to represent ternary relationships using Crow's foot notation.
21. Which one of the following is correct?
    (a) If a model includes a supertype vehicle and subtypes car, truck and bus then that must be generalization.
    (b) If a model includes a supertype vehicle and subtypes car, truck and bus then that must be specialization.
    (c) If a model includes a supertype computer and subtypes monitor, keyboard, motherboard then that must be specialization.
    (d) If a model includes a supertype vehicle and subtypes car, truck and bus then that may be generalization or specialization.

22. EER model extends the E-R model. Which one of the following is **not** in the EER model?
    (a) Supertypes and subtypes (b) Inheritance
    (c) Aggregation (d) Objects
23. Which of the following comments about the E-R model are correct?
    (a) Attributes of 1:1 relationships can be represented as attributes of either of the participating entity types.
    (b) Attributes of 1:*n* relationships can be represented as attributes of either of the participating entity types.
    (c) Attributes of a relationship involving more than two entity types can be represented as attributes of either of the participating entity types.
    (d) Ternary relationships normally do not have attributes.
    (e) None of the above.

EXERCISES

1. Identify which of the following are entities, relationships and attributes. List any assumptions that you have made in making your choices and list any attributes that are appropriate for each. What might be the primary key for each entity and relationship that has been identified?
    (a) Employees (b) Customers
    (c) Patients (d) Shares
    (e) Houses (f) Offices
    (g) Employment position (h) Lecture
    (i) Examination (j) Grade
    (k) Address (l) Telephone number
    (m) Dependents
2. Show how an entity, a relationship and an attribute are represented in E-R diagrams. Draw a diagram showing the entities university department, degree, course, lecture theatre, and instructor and a number of relationships between them.
3. Explain by drawing a diagram of how you would model a course that has many prerequisites and which is a prerequisite for many courses?
4. Explain by drawing a diagram of how you would model a student that is enrolled in many courses and each course has many enrolments.
5. Explain by drawing a diagram of how you would model a driving license which may be for driving a car, bus, truck, taxi, scooter or a tourist bus. Use a superclass/subclass model. List all the information that each entity should possess.
6. Develop an Entity-Relationship model for Indian movies. It should include where it was shot, details of the actors, directors, producers, the movie language and so on.
7. Can the E-R model express a relationship between two or more relationships? Would such a relationship ever be required in real life?
8. Construct an E-R diagram for student information in a Department of Computer Science in a university. This should include personal information about the student, enrolment information, and assessment information. Do not include information about library or finances.

9. Briefly describe the database design process described in this chapter. Explain why the design process is often iterative.
10. Build an E-R model for major banks of the world. Choose may be the top 500 banks or a 1000 or more. Study what are the major pieces of information about these banks available on the Web. Use the information to build a model. Are there any relationships that you can find?
11. Can a weak entity itself have a weak entity dependent on it? Give an example.
12. The two E-R diagrams (Figs 2.30 and 2.31), of the employees, departments and projects are given below. Consider the following queries:

    (a) Which employees work on project J1?
    (b) Which departments are involved in project J1?
    (c) Which parts are being supplied to project J1?
    (d) How many employees of department D1 working on project J1?
    (e) Are there parts that S1 can supply but is not supplying currently?
    (f) Are there parts that J1 uses but are not being supplied currently?

    - Which of the above queries can be made if the first model is used? _____ (Write appropriate query labels *A*, *B*, …, etc., in the space above)
    - Which of the above queries can be made if the second model is used? _____
    - Which of the above queries cannot be answered by any of the models? _____

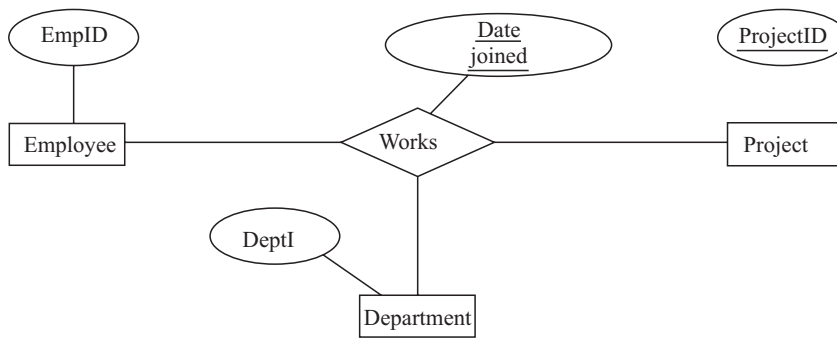

Figure 2.30 A ternary relation between Employee, Project and Department

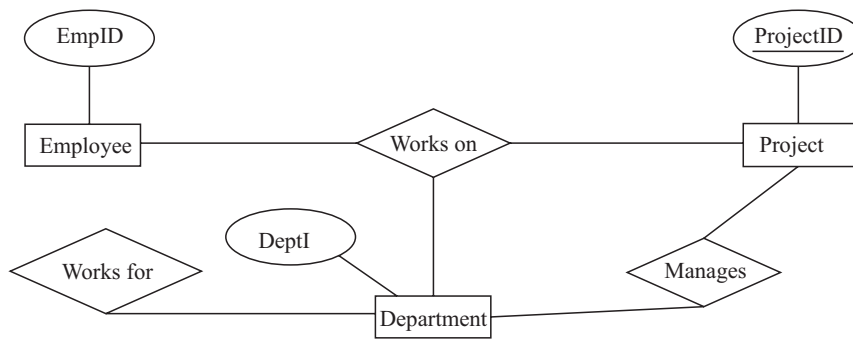

Figure 2.31 Three binary relations between Employee, Project and Department

13. Describe how the following two sets of entities may be included in a generalization hierarchy.
    (a) Book, journal, magazine, newspaper
    (b) Truck, car, bicycle, scooter, rickshaw, bus, airplane
14. Describe how the entity animal (in a zoo) can be developed into a specialization hierarchy.

PROJECTS

For each E-R modeling project, present assumptions, develop a model and discuss strengths and weaknesses of the model that you have designed.

1. Western India Medical Clinic (WIMC)
Design a database for WIMC located in Mangalore. The clinic has a number of regular patients and new patients come to the clinic regularly. Patients make appointments to see one of the doctors. Several doctors attend the clinic and they each have their own hours when they attend the clinic. Some doctors are general practitioners (GPs) while others are specialists (for example, cardiologists, dermatologists, endocrinologists). Doctors' hours are defined and fixed by each doctor but can be changed if necessary. Different doctors may charge different fees. Doctors send patients to other clinics for X-Rays and blood tests. These reports are sent from the clinics to doctors electronically. Doctors write medical certificates. Patients have families and the family relationships are important. Medical records of each patient need to be maintained. Information on prescriptions, insurance, allergies, etc., needs to be maintained.

Design an E-R model for the medical clinic.

2. Country/Rivers/Deserts/Mountains/Animals
A database of countries, their rivers, deserts, mountains and animals is to be developed. We want to store a variety of information about the countries depending on availability of information, e.g., name, population, leader's name, number of newspapers per 1000 people, amount of alcohol consumed per person, energy consumed per person, pollutants generated per person, etc. For rivers, we would want to store information like name, country or countries it passes through, source of the river, destination, how long, etc. For mountains, we would want to store information like name, height, country, part of which mountain range, etc. For animals, we need to store information like name, type (mammal, bird, etc.), found in which countries, and some of their characteristics.

Design an E-R model for this problem.

3. All India Mega Constructors (AIMC)
AIMC is an international company that specializes in large-scale construction. They build such things as bridges, dams, office blocks, hotels and factories. The company is divided into a number of departments. Each department specializes in one form of construction. For example, department 654 deals exclusively with the building of bridges, regardless of their geographical location. Head office, however, is interested in the activities of the various geographical areas in which the company operates. This is for political and financial reasons. Each individual construction activity is termed a project. Each project is given a code, which is used to uniquely identify the project in any reports.

Prior to the commencement of construction, the project is analysed into a number of tasks. This is done by the project leader and his or her assistants. Each task is given a code, unique to that project. For each task, a time estimate is made. A monthly budget is developed for the expected lifetime of the project. As each project progresses, actual expenditure is monitored.

There are several types of reports to be produced. For example, a departmental summary is often needed, a project status report may be required and a summary of projects in a particular area is often asked for.

Design an E-R model for the company.

4. The New Programming Company (NPC)
NPC started a venture capital company that decided to employ a number of unemployed computing graduates from the Northern India University (NIU). These graduates were unfortunate enough to graduate in the middle of the global financial crisis (GFC) in 2008. Having become part of the new venture, the graduates started advertising as the best programming company in India even before they had completed any programming job. Although their company name had the word 'new' in it, they claimed that had expertise in old programming languages like FORTRAN, COBOL, Algol, Pascal and Lisp.

The company has three branches called P1, P2 and P3 located in Patna, Kochi and Bhubaneswar. The company's Kochi branch has recently applied to develop software for Qatar Telecom.

The company operates in the following manner. Each time an employee starts on a task, he/she is given a job card on which the employee records the number of hours worked each day. The job card is returned to the supervisor once the assignment is completed. The job card has a variety of information on it including the job card ID, employee ID, job ID, and for each day, the date, the number of hours involved and the nature of work done.

Once the job is completed, NPC sends a bill for services to the customer company. The total charge is computed by adding the number of hours devoted to the job based on the job card multiplied by Rs. 750 which is the company's hourly rate. Once a payment is received the company issues a receipt.

At the end of each month, each employee's performance is evaluated based on the job cards. Total number of hours worked is computed and the income generated by each employee for the company is computed. Bonuses are awarded based on a complex formula for each employee who earns more than one lakh for that month.

Develop an Entity-Relationship model for the company.

LAB EXERCISES

1. Study the E-R diagram (Fig. 2.32) on the next page and do the following:
   (a) Describe the database that is modeled by the diagram.
   (b) Are the naming conventions being followed in this diagram? If not, suggest change of names wherever appropriate.
   (c) Find any mistakes that you can identify in the diagram. Explain and correct them.
   (d) List all the constraints that the model imposes on the database.

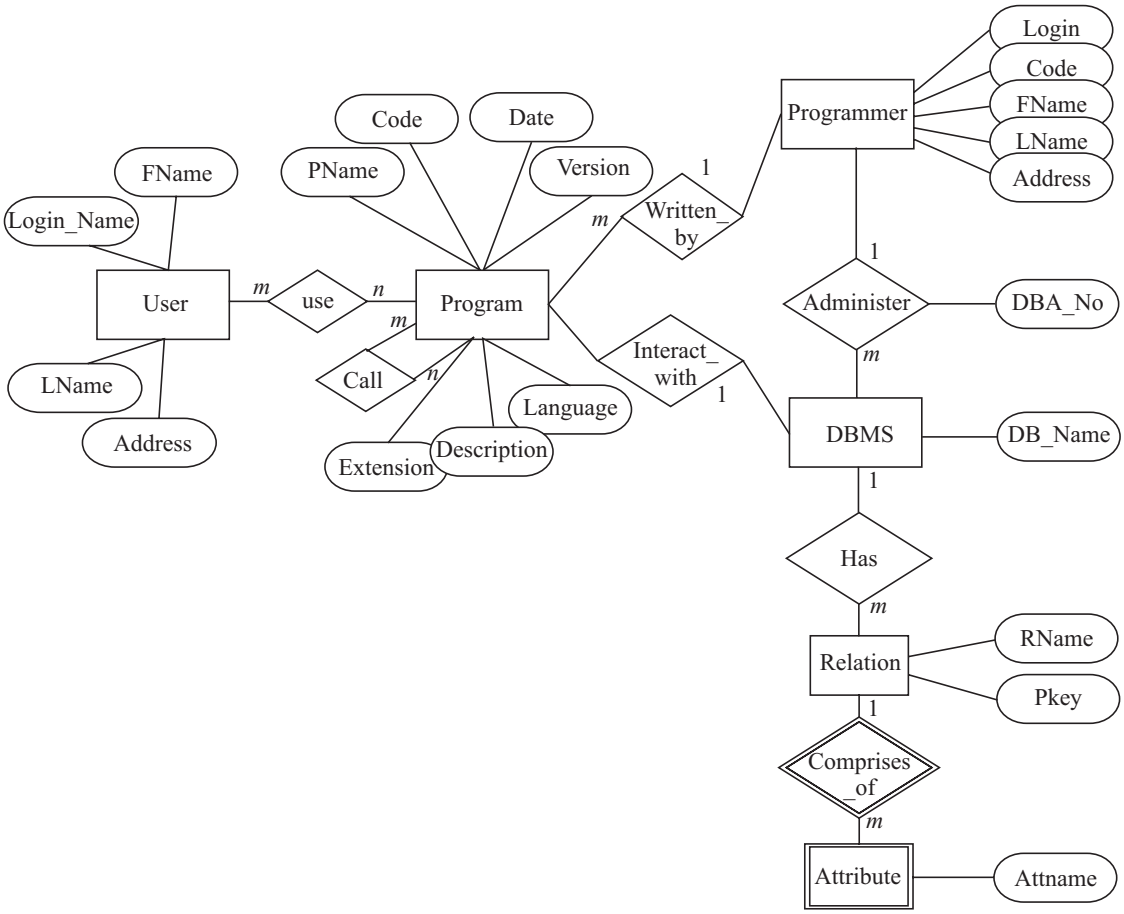

Figure 2.32

(e) Should the entity relations be a weak entity? If it is and therefore the relationship 'Maintains data in' is a weak relationship, what will be the primary keys of the weak relationships and weak entities.

(f) Modify the E-R model by using generalization

2. Consider the case study described in Section 2.11.
   (a) What were the requirements of the case study?
   (b) Were the requirements met by the model developed?
   (c) If not, modify the model to ensure that it does meet the requirements.
3. Consider the two E-R diagrams (Figs 2.33 and 2.34) below that model a simple horse-racing database.
   (a) Describe the database by studying these two diagrams.
   (b) Find any mistakes that you can identify in each of the diagrams. Explain and correct the mistakes in them.
   (c) Which one of the diagrams models the database better? Are there constraints that apply to one diagram but not the other? Are there queries that could be answered by one of the diagrams but not the other?

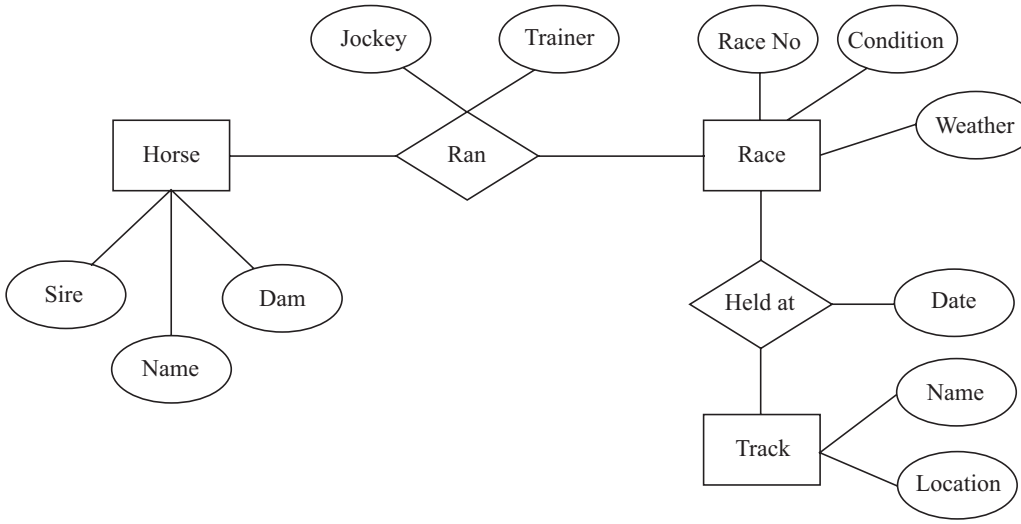

Figure 2.33

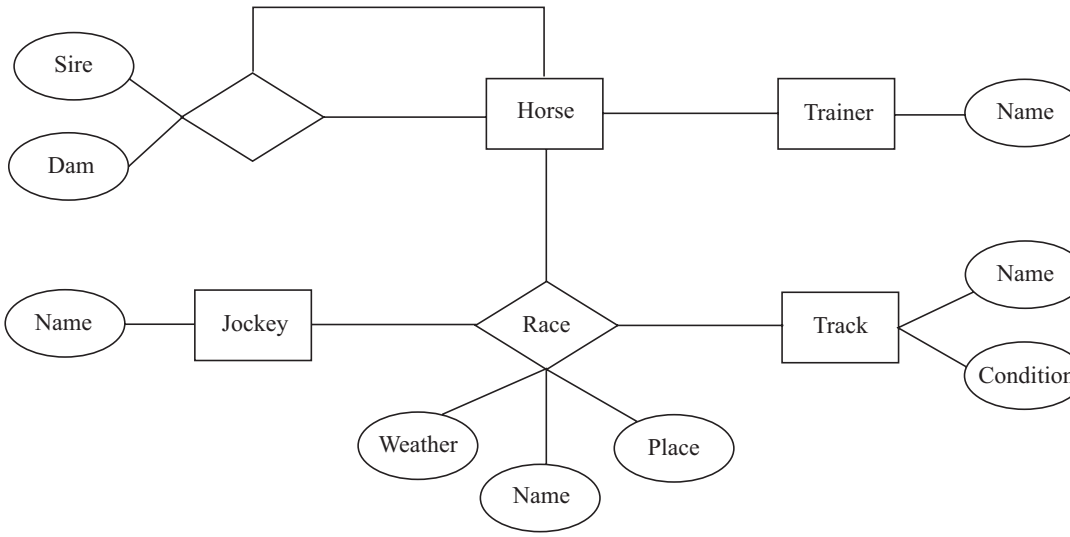

Figure 2.34

BIBLIOGRAPHY

For Students

The book by Batini, Ceri and Navathe and the book by Theorey are good introductions to E-R database modeling. The book by Baqui and Earp is also good. Most database textbooks have chapters on E-R modeling. The notes by Jung and Reddy are also worth consulting.

For Instructors

There are a number of good textbooks on database modeling. On Entity-Relationship modeling, the original 1976 paper by Chen is listed below. The books by Batini, Ceri and Navathe, by Thalheim and by Simsion are excellent. Halpin's book is different since it deals with the Object Relational Model (ORM) rather than with the E-R model and is worth reading if you want to look at another approach to database modeling. The paper by Herden looks at the quality of database models and presents an approach to measuring quality of a model. The papers by Burkett and Herden are good surveys of a number of approaches to database model quality.

Baqui, S., and R. Earp, *Database Design Using Entity-Relationship Diagrams*, Auerbach, 2003, 264 pp.

Batini, C., S. Ceri, and S.B. Navathe, *Conceptual Database Design: An Entity-Relationship Approach*, Benjamin Cunnings, 1992.

Burkett, W. C., "Database Schema Design Quality Principles". http://wwwscf.usc.edu/ wcb/dmq/dmqmain.html, 1998.

Chen, P. P., "The Entity-Relationship Model: Towards a unified view of data", *ACM Trans. Database Systems*, Vol. 1, pp 9-36, 1976.

Chen, P. P., *Entity Relationship Approach to Logical Database Design*, Monograph Series, QED Information Sciences Inc., Wellesley, Ma, 1977.

Chen, P. P. (Ed.), *Entity-Relationship to Systems Analysis and Design*, North-Holland, New York/Amsterdam, 1980.

Elmasri, R., and S. B. Navathe, *Fundamentals of Database Systems*, Fifth Edition, Addison-Wesley, 2006.

Halpin, T., *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design, using ORM with ER and UML*, Morgan Kaufmann, 2001.

Herden, O., "Measuring quality of database schemas by reviewing – concept, criteria and tool", *Proceedings of the 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Budapest (Ungarn), pp 59-70, 2001. http://www.iro.umontreal.ca/~sahraouh/qaoose01/Herden.PDF

Hernandez, M. J., *Relational Design for Mere Mortals: A Hands-on Guide to Relational Database Design*, Second Edition, Addison-Wesley, 2003.

Jung, A., and U. Reddy, "Entity-Relationship Modeling", School of Computer Science, University of Birmingham, 2009. http://www.cs.bham.ac.uk/~udr/db2/handout2.pdf

Ng, P., "Further analysis of the entity-relationship approach to database design", *IEEE Transactions on Software Engineering*, Vol. DE-7, No. 1, pp 85-98, 1981.

Simsion, G.C., and G. C. Witt, *Data Modeling Essential*, Third Edition, Morgan Kaufmann, 2005.

Spaccapietra, S. (Ed.), *Entity-Relationship Approach: Ten Years of Experience in Information Modelling*, Proceedings of the Entity-Relationship Conference, North-Holland, 1987.

Song, I. Y., M. Evans, and E. K. Park, "A comparative analysis of entity-relationship diagrams", *Journal of Computer and Software Engineering*, Vol. 3, No. 4, 1995, pp 427-459.

Theorey, T. J., *Database Modelling and Design: Logical Design*, Fourth Edition, Morgan Kaufmann, 1999.

Theorey, T. J., D. Yang, and J. P. Fry, "Logical Design Methodology for Relational Databases", *ACM Computing Surveys*, pp 197-222, June 1986.

Thalheim, B., *Entity-Relationship Modelling*, Springer, 2000.

The University of Texas at Austin, "Introduction to Database Modelling", *Information Technology Services*. http://www.utexas.edu/its/database/datamodeling.

CHAPTER 3

Relational Model

OBJECTIVES

- Introduce the relational model and its three components.
- Discuss the data structure used in the relational model.
- Define the properties of a relation and explain relational terminology.
- Describe how an Entity-Relationship model may be mapped to the relational model.
- Describe how information is retrieved and modified in the relational model.
- Explain how data integrity is maintained in the relational model.
- Discuss the advantages and disadvantages of using the relational model.
- Discuss the older database models, the network and hierarchical models.

KEYWORDS

Relational model, mapping E-R model, relations, tables, atomic domain, domains, tuples, cardinality, degree, attributes, rows, tables, columns, schema, constraints, candidate key, primary key, foreign key, existential integrity, referential integrity, domains, nulls, integrity, hierarchical model, network model.

Earth provides enough to satisfy every man's need, but not every man's greed.

Mahatma Gandhi

3.1 INTRODUCTION

In the last chapter we discussed the Entity-Relationship (E-R) model, a technique for building a logical model of an enterprise. Logical models are high level abstract views of an enterprise data. In building these models little thought is given to representing the database in the computer or retrieving data from it. At the next lower level, a number of data models are available that provide a mapping for a logical data model like the E-R model. These models specify a conceptual view of the data as well as a high level view of the implementation of the database on the computer. They do not concern themselves with the detailed bits and bytes view of the data.

In the 1960s, there were two widely used data models at this intermediate level. These were the *hierarchical model* and the *network model*. These are briefly discussed in Section 3.7. Readers interested in a more detailed discussion on the hierarchical and network models should refer to some of the older database books. As noted in the first chapter, a new model called the relational model was proposed by E. F. Codd in 1970. The model is based on the concept of *mathematical relation* which looks like a table of values. In the beginning the computing industry was sceptical of the relational approach; many computing professionals thought the approach could not be efficient for managing large databases. This however changed as prototypes of relational database systems were built and shown to be efficient. The first two prototypes were the Model R at IBM San Jose Research Lab in California and INGRES at the University of California at Berkeley. With the success of the prototypes, a number of commercial relational database management systems became available and the hierarchical and network models were superseded by the relational model. Some computing installations however continue to use the old DBMS to process their legacy data. During the last 30 years or so, a number of other new models have also been proposed. Some are based on an object oriented approach to viewing the data, others consider an object relational approach while the development of XML itself is leading to another way of looking at databases. These topics will be discussed in Chapters 14 and 17 respectively. The main focus of this chapter is the relational model.

All the data models provide facilities for representing entities and their attributes as well as relationships. The data about entities and their attributes is handled in a similar way in all the models. The representation is a record which is a collection of attribute values (including an identifier of the entity) although the structure of the record may be somewhat different. The relationships are however represented very differently in the models. We start by looking at the simplest model, i.e., the relational model.

One might ask the question why we should bother about the relational model when we already have the E-R model. In some ways, the two models are quite different. Using the E-R model allows us to define the structure of the enterprise data more easily than using the relational model but the simplicity of the relational model allows powerful query languages to be designed that could not be employed with the E-R model. Essentially the E-R model is used to build a conceptual view of the database that is implemented using the relational model.

This chapter is organized as follows. Section 3.2 deals with the data structure used in the relational model. Mapping of the E-R model to the relational model is described here. Data manipulation is briefly discussed since it is discussed in detail in Chapter 4. The final component of the relational model, data integrity is discussed in Section 3.4 including existential integrity, referential integrity, domains and nulls. Advantages of the relational model are presented in Section 3.5. Codd's rules for fully relational systems are then listed in Section 3.6 followed by a brief description of older database models; hierarchical and network models in Section 3.7.

3.1.1 The Relational Model

As noted earlier, the relational model was proposed by Codd in 1970. The model deals with database management from an abstract point of view. It provides specifications of an abstract database management system. The model is very easy to understand although it is based on a solid theoretical foundation that includes predicate calculus and the theory of relations. However, to use a database management system based on the relational model, users do not need to master these theoretical foundations.

Codd defined the relational model as consisting of the following three components:

1. *Data Structure*—A collection of data structure types for building a database.
2. *Data Manipulation*—A collection of operators that may be used to retrieve, derive, build a database or modify data stored in the data structures.
3. *Data Integrity*—A collection of rules that implicitly or explicitly define a consistent database state or change of state.

Each of the three components are discussed in Sections 3.2, 3.3 and 3.4 respectively.

3.2 DATA STRUCTURE

The beauty of the relational model is the simplicity of its data structure. The fundamental property of the relational structure is that all information about the entities and their attributes as well as the relationships is presented to the user as tables (called *relations*[1]) and nothing but tables. A database therefore is a collection of relations. Each row of each table consists of an entity occurrence or a relationship occurrence. Each column refers to an attribute. The model is called relational and not tabular because tables are a lower level of abstraction than the mathematical concept of relation. Tables give the impression that positions of rows and columns are important. In the relational model, it is assumed that no ordering of rows and columns is defined.

Figure 3.1 gives number of properties of a relation:

The properties listed in Fig. 3.1 are simple and based on practical considerations. The first property ensures that only one type of information is stored in each relation. The second property ensures that each column is named uniquely. This has several benefits. The names can be chosen to convey what each column is and the

1. We will use the two terms *table* and *relation* interchangeably.

- Each relation contains only one record type.
- Each relation has a fixed number of columns (attributes) that are explicitly named. Each attribute name within a relation is unique.
- No two rows (tuples) in a relation are the same.
- Each item or element in the relation is atomic, that is, in each row, every attribute has only one value that cannot be decomposed into smaller components.
- Rows have no ordering associated with them since a relation is a set of rows and sets have no ordering associated with them.
- Columns have no ordering associated with them (although most commercially available systems do) since each column has a unique name and that column' s value in each row can be easily identified by the column name.

Figure 3.1 Properties of a relation in the relational model

names enable one to distinguish between the column and its domain. Furthermore, the names are much easier to remember than the position of each column if the number of columns is large.

The third property of not having duplicate rows appears obvious but is not always accepted by all DBMS users and designers. The property is essential since no sensible context free meaning can be assigned to a number of rows that are exactly the same.

The fourth property requires that each element in each relation be atomic and that it cannot be decomposed into smaller elements. For example, if an attribute value is 1976–90 then it is not possible to look at just the first part of the value (i.e., 1976). In the relational model, the only composite or compound type (data that can be decomposed into smaller pieces) is a table. This simplicity of structure leads to relatively simple database languages.

The fifth property deals with ordering of the rows in a relation. A relation is a set of rows. As in other types of sets, there is no ordering associated with the rows in a table. Just like the second property, the ordering of the rows should not be significant since in a large database no user should be expected to have any understanding of the positions of the rows of a table. Also, in some situations, the DBMS should be permitted to reorganize the data and change row orderings if that is necessary for efficiency reasons. The columns are identified by their names. The rows on the other hand are identified by their contents, possibly the attributes that are unique for each row.

A table is a set of rows and is closely related to the concept of relation in mathematics. Each row in a relation may be viewed as an assertion. For example, a row in a relation *Player* may assert that a player by the name of *Sachin Tendulkar* has *player_id 89001* and represents *India*. Similarly, a row in a table *Match* may assert that one of the ODI matches was held on *2 March 2008* in *Sydney.* The tables in our ODI database will be presented a little later in this section.

As noted earlier, in the relational model, a relation is the only data structure. The relational terminology is defined in Fig. 3.2.

- *Relation* — It is essentially a simple table. It may be defined as set of rows (or tuples), each row therefore has the same columns (or attributes). Since a relation is a sort of tuple, there is no ordering associated with the tuples. (As noted earlier, we will continue to use both terms relation and table to mean the same).
- *Tuple* — It is a row in the relation (we will use both the teems tuple and row to mean the same).
- *Attribute* — It is a column in the relation.
- *Degree of a relation* — It is the number of columns in the relation.
- *Cardinality of a relation* — It is the number of rows in the relation.
- *N-ary relation* — It is the a relation with degree *N*.
- *Domain* — It is a set of atomic (that is, indivisible) values that each element in a column is permitted to take. Domains may be given unique names. The same domain may be used by a number of different columns.
- *Candidate key and Primary key* — It has been discussed in the last chapter and defined below.

Figure 3.2 Relational Terminology

A formal definition of candidate key is given below:

Definition—Candidate Key
An attribute (or a set of attributes) is called a candidate key of the relation if it satisfies the following properties:

(a) the attribute or the set of attributes uniquely identifies each tuple in the relation (called the uniqueness property), and
(b) if the key is a set of attributes then no subset of these attributes has the property (a) (called the minimality property).

There may be several distinct sets of attributes that may serve as candidate keys but every table must always have at least one.

Primary key may be defined as follows:

Definition—Primary Key
One (and only one) of the candidate keys is arbitrarily chosen as the primary key of the table. Primary key therefore has the properties of uniqueness and minimality.

We will often use symbols like R or S to denote a table and r and s to denote a row. When we write $r \subset R$, we are stating that row r is a row in table R.

3.3 MAPPING THE E-R MODEL TO THE RELATIONAL MODEL

In the relational model, information about entities and relationships is represented in the same way, that is, by tables. Since the structure of all information is the same, the same operators may be applied to them.

We consider the following ODI cricket database, developed in the last chapter, to illustrate the mapping of an E-R model to the relational model as well as basic concepts of the relational data model. We first reproduce the E-R diagram for the database as shown in Fig. 3.3.

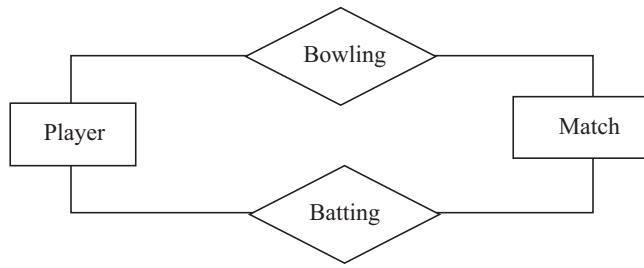

Figure 3.3 E-R diagram for ODI matches

We have not included the attributes in the E-R diagram in Fig. 3.3. The attributes are given below. Some of the attribute names have been changed so that there are no spaces in the names and the names are shorter for ease of display.

The entity *Match* has the attributes given in Fig. 3.4 (its primary key is *Match ID*). We have made some minor changes to the attribute names used in the last chapter to make the names a bit more meaningful.

We have changed the attribute names *Home Team* and *Visiting Team* to *Team1* and *Team2* respectively because many ODI matches are being held in third countries, for example, India and South Africa played on 1st July 2007 in Belfast. We will not discuss such matches in the examples that follow. We will also change the names of attributes to remove spaces within names that are not acceptable in SQL and will shorten several names and replace some by names used by the cricinfo (www.cricinfo.com) website, for example, http://contentaus.cricinfo.com/engvind/engine/series/258449.html to make it easier to use them in the examples that follow.

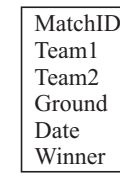

Figure 3.4 The attributes for the entity *Match*

Figure 3.5 The attributes for the entity *Player*

The entity *Player* has its attributes listed in Fig. 3.5 (The primary key is *Player ID*):

The relationship *Batting* has its attributes listed in Fig. 3.6 (The primary key is *Match ID* and *PID* together).

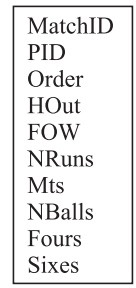

Figure 3.6 The attributes for the relationship *Batting*

The relationship *Bowling* entity has its attributes listed in Fig. 3.7 (The primary key is *MatchID* and *PID* together).

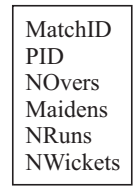

Figure 3.7 The attributes for the relationship *Bowling*

Now that we have all the details of the E-R model, we consider mapping it to the relational model. The mapping is quite straightforward since the relational model closely approximates the E-R Model. We will show this by transforming the E-R model in Fig. 3.3 (including the attribute lists in Figs 3.4 to 3.7) into a set of relations.

We will follow the steps given below for mapping an E-R diagram into a relational model.

3.3.1 Step 1–Mapping Strong Entities

Each strong entity in an E-R model is represented by a table in the relational model. Each attribute of the entity becomes a column of the table. For example, the entities *Player* and *Match* in Fig. 3.3 may be directly transformed into two tables of the same names given in Tables 3.1 and 3.2 respectively. We have populated these tables using real ODI data. We have only selected matches in which India has played. Note that in almost all entries one of the teams, either *Team 1* or *Team 2*, is the winner but other results are possible including abandoned, no result, draw or cancelled.

Table 3.1 below is the representation of the entity *Match* in the E-R model of Fig. 3.3.

Table 3.1 The relation *Match*

MatchID	*Team 1*	*Team 2*	*Ground*	*Date*	*Winner*
2324	Pakistan	India	Peshawar	6/2/2006	Team 1
2327	Pakistan	India	Rawalpindi	11/2/2006	Team 2
2357	India	England	Delhi	28/3/2006	Team 1
2377	West Indies	India	Kingston	18/5/2006	Team 2
2404a	Sri Lanka	India	Colombo	16/8/2006	Abandoned
2440	India	Australia	Mohali	29/10/2006	Team 2
2449	South Africa	India	Cape Town	26/11/2006	Team 1
2480	India	West Indies	Nagpur	21/1/2007	Team 1
2493	India	West Indies	Vadodara	31/1/2007	Team 1
2520	India	Sri Lanka	Rajkot	11/2/2007	Team 2
2611	England	India	Southampton	21/8/2007	Team 1
2632	India	Australia	Mumbai	17/10/2007	Team 1
2643	India	Pakistan	Guwahati	5/11/2007	Team 1
2675	Australia	India	Melbourne	10/2/2008	Team 2
2681	India	Sri Lanka	Adelaide	19/2/2008	Team 1
2688	Australia	India	Sydney	2/3/2008	Team 2
2689	Australia	India	Brisbane	4/3/2008	Team 2
2717	Pakistan	India	Karachi	26/6/2008	Team 2
2750	Sri Lanka	India	Colombo	24/8/2008	Team 2
2755	Sri Lanka	India	Colombo	27/8/2008	Team 2

Table 3.2 below is the representation of the entity *Player* in E-R model of Fig. 3.3.

Table 3.2 The relation *Player*

PlayerID	*LName*	*FName*	*Country*	*YBorn*	*BPlace*	*FTest*
89001	Tendulkar	Sachin	India	1973	Mumbai	1989
90001	Lara	Brian	West Indies	1969	Santa Cruz	1990
95001	Ponting	Ricky	Australia	1974	Launceston	1995
96001	Dravid	Rahul	India	1973	Indore	1996
96002	Gibbs	Herschelle	South Africa	1974	Cape Town	1996
92001	Warne	Shane	Australia	1969	Melbourne	1992
95002	Pollock	Shaun	South Africa	1973	Port Elizabeth	1995
99003	Vaughan	Michael	England	1974	Manchester	1999

Contd.

Table 3.2 Contd.

PlayerID	*LName*	*FName*	*Country*	*YBorn*	*BPlace*	*FTest*
92003	Ul-Huq	Inzamam	Pakistan	1970	Multan	1992
94004	Fleming	Stephen	New Zealand	1973	Christchurch	1994
93002	Streak	Heath	Zimbabwe	1974	Bulawayo	1993
90002	Kumble	Anil	India	1970	Bangalore	1990
93003	Kirsten	Gary	South Africa	1967	Cape Town	1993
95003	Kallis	Jacques	South Africa	1975	Cape Town	1995
94002	Vaas	Chaminda	Sri Lanka	1974	Mattumagala	1994
92002	Muralitharan	Muthiah	Sri Lanka	1972	Kandy	1992
97004	Vettori	Daniel	New Zealand	1979	Auckland	1997
25001	Dhoni	M S	India	1981	Ranchi	2005
23001	Singh	Yuvraj	India	1981	Chandigarh	2003
96003	Ganguly	Saurav	India	1972	Calcutta	1996
99002	Gilchrist	Adam	Australia	1971	Bellingen	1999
24001	Symonds	Andrew	Australia	1975	Birmingham	2004
99001	Lee	Brett	Australia	1976	Wollongong	1999
91001	Jayasuriya	Sanath	Sri Lanka	1969	Matara	1991
21001	Sehwag	Virender	India	1978	Delhi	2001
98001	Afridi	Shahid	Pakistan	1980	Khyber Agency	1998
98002	Singh	Harbhajan	India	1980	Jalandhar	1998
27001	Kumar	Praveen	India	1986	Meerut	NA
27002	Sharma	Ishant	India	1988	Delhi	2007

3.3.2 Step 2–Mapping Weak Entities

Although Fig. 3.3 E-R diagram does not include a weak entity, we must deal with weak entities if they are in the model. Each weak entity may be mapped to a relation that includes all simple attributes of the entity as well as the primary key of the dominant entity. The primary key of the relation representing a weak entity will be the combination of the primary key of the dominant entity and the partial key of the weak entity.

3.3.3 Step 3–Mapping Relationships

As noted in Chapter 2, there can be three types of relationships, viz., 1:1, 1:*m* and *m*:*n*. We deal with each of them.

Step 3(a)–Mapping-One-to-one Binary Relationships

Figure 3.3 does not include any 1:1 relationships but some situations are likely to give rise to such relationships.

We should note that although each relationship may be mapped to a table in the relational model, not all relationship types need to be represented by separate tables. For 1:1 key propagation may be used. In a 1:1 relationship, each instance of the first entity in a relationship is only related to one instance of the other entity. Therefore it is straightforward to represent the relationship by either putting the primary key of the first entity in the second entity or the primary key of the second entity in the first. This is called *key propagation*.

Such situations of 1:1 relationship may occur in *Department* and *Manager* entities since each department must have a manager and each manager must manage one department. We may choose one of the entities say entity *Department* which should already been mapped into a relation and include the primary key of the other entity, *Manager* (which would also have been mapped to a relation) in the *Department* relation. This is then a foreign key in the relation (that represents *Department*). If the relationship has simple attributes, those may also be included in this relation that represents *Department*.

In some situations when mapping 1:1 relationship between entities *Department* and *Manager*, both entities and the relationship may be merged into one and represented by one relation. As noted earlier, the other extreme solution of course is to represent the relationship as a table that includes primary keys of both entities as well as all the attributes of the relationship.

Step 3(b)–Mapping One-to-many Relationships

Figure 3.3 does not include any 1:*m* relationships but many situations give rise to such relationships.

In a 1:*m* relationship, one entity instance of the first entity may be related to a number of instances of the second entity but each instance of the second entity is related to only one instance of the first entity. This may be the case if we had entities *Employee* and *Manager*. Key propagation is therefore also possible in 1:*m* relationships. We cannot put the primary key of the related instances of *Employee* in the entity *Manager* because for this instance of the entity *Manager* there are likely to be a number of related instances of the *Employee*. We can however propagate the primary key of *Manager* to the related instance because each instance of *Employee* is related to only one instance of *Manager*. The primary key of *Manager* in *Employee* then becomes a foreign key in the relation *Employee*. If the relationship has simple attributes, those may also be included in the relation *Employee*.

Step 3(c)–Mapping Many-to-many Relationships

Key propagation is not possible for *m*:*n* relationships because it is not possible to propagate primary keys of instances of a related second entity to an instance of the related first entity, since there may be more than one such instance of the second entity. Similarly, it is not possible to propagate primary keys of a related first entity to the related second entity since there may be more than one such instance of the first entity. Many-to-many relationships therefore must be represented by a separate table.

Therefore each binary *m*:*n* relationship must be represented by a new relation. The primary keys of each of the entities that participates in the relationship are included in this relation. The combination of these keys will make up the primary key for the relation. Also include any attributes of the relationship in this relation.

For example, when the relationship *Batting* in Fig. 3.3 is transformed into Table 3.3 *Batting*, the primary key of the table is the combination of the primary keys of the two entities *Match* and *Player*. Similarly when *Bowling* in Fig. 3.3 is transformed into Table 3.4 *Bowling*, the primary key of the table is the combination of

the primary keys of the two entities *Match* and *Player*. The primary keys of both tables *Batting* and *Bowling* therefore are the same which does not create any problems.

The relationships *Batting* and *Bowling* are both many-to-many relationships since each batsman and each bowler plays in more than one match and each match has more than one batsman and more than one bowler. We therefore transform these two relationships into Table 3.3 and Table 3.4 respectively.

We have again populated the two tables, Table 3.3 and Table 3.4, with real ODI data. The data in both these tables is very limited given the space limitations. We have included part of the batting and bowling information from only two ODI matches, one between Australia and India in Brisbane in March 2008 (Match ID 2689) and the other between Sri Lanka and India played in Colombo on 27th August 2008 (Match ID 2755). India won both these matches. More data for these tables is presented in the appendix at the end of Chapter 5. This data may be useful for some examples and exercises in Chapters 4 and 5.

The relationship *Batting* is mapped to the relational model as shown in Table 3.3.

Table 3.3 The relation *Batting*

MatchID	*PID*	*Order*	*HOut*	*FOW*	*NRuns*	*Mts*	*NBalls*	*Fours*	*Sixes*
2755	23001	3	C	51	0	12	6	0	0
2755	25001	5	C	232	71	104	80	4	0
2755	91001	1	C	74	60	85	52	8	2
2755	94002	7	LBW	157	17	23	29	1	0
2755	92002	11	NO	NO	1	11	7	0	0
2689	89001	2	C	205	91	176	121	7	0
2689	23001	4	C	175	38	27	38	2	2
2689	25001	5	C	240	36	52	37	2	1
2689	99002	1	C	2	2	3	3	0	0
2689	95001	3	C	8	1	9	7	0	0
2689	24001	5	C	123	42	81	56	2	1
2689	99001	8	B	228	7	20	12	0	0
2689	27001	9	C	255	7	7	7	1	0
2755	27001	9	B	257	2	7	6	0	0
2689	98002	8	LBW	249	3	7	3	0	0
2755	98002	8	RO	253	2	7	4	0	0

The relationship *Bowling* in the E-R model given in Fig. 3.3 is mapped to the relational model in Table 3.4.

The above database schema may also be written as in Fig. 3.8.

The E-R diagram in Fig. 3.3 has now been mapped into the relational schema in Fig. 3.8 consisting of four tables. Each table schema presents the structure of the table by specifying its name and the names of its attributes enclosed in parenthesis. The primary key of a relation may be marked by underlining although we have not done so in Fig. 3.8.

Table 3.4 The relation *Bowling*

MatchID	*PID*	*NOvers*	*Maidens*	*NRuns*	*NWickets*
2689	99001	10	0	58	1
2689	24001	3	0	27	1
2689	23001	3	0	15	0
2755	94002	9	1	40	1
2755	92002	10	0	56	1
2755	91001	4	0	29	0
2755	23001	10	0	53	2
2755	98002	10	0	40	3
2689	98002	10	0	44	1

Match(MatchID, Team1, Team2, Ground, Date, Winner)
Player(PlayerID, LName, Country, YBorn BPlace FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRuns, Mts, NBalls, Fours, Sixes)
Bowling(MatchID, PID, NOvers, Maidens, NRuns, NWickets)

Figure 3.8 Database schema for the ODI cricket database

Now to summarize the mapped relational model, we note that the four tables, Table 3.1 to Table 3.4, *Match*, *Player*, *Batting* and *Bowling*, have degree 6, 7, 10 and 6 respectively and cardinality 20, 29, 14 and 7 respectively. The primary key of the table *Match* is MatchID, of table *Player* is PlayerID, of table *Batting* is (MatchID, PID) and finally the primary key of the table *Bowling* is (MatchID, PID). The table *Match* has another candidate key (Ground, Date) as well one or two others. If we can assume the names to be unique then *Player* has (LName, FName) as a candidate key. If the names are not unique but the names LName, FName and FTest together are unique, then the three attributes together form a candidate key. Note that both PlayerID and (PlayerID, Country) cannot be candidate keys, only PlayerID can.

3.3.4 Key Propagation

We further illustrate the use of key propagation by considering the simple E-R model in Fig. 3.9.

Figure 3.9 A simple database

Let the relationship *Occupies* be many-to-one, that is, one employee may occupy only one office but a room may be occupied by more than one employee. Let the attributes of the entity *Employee* be employee number (*e_num*), name (*name*) and *status*. Attributes of the entity *Office* are office number (*o_num*), *capacity* and

building. *Occupies* has attributes *date*, *e_num* and *o_num* that include the primary keys of the two entities *Employee* and *Office*. One way to represent the above database is to have the three tables whose schema is given in Fig. 3.10.

Employee(e_num, name, status)
Occupies(e_num, o_num, date)
Office(o_num, capacity, building)

Figure 3.10 Three relations representing the E-R diagram in Fig. 3.9

There is of course another possibility. Since the relationship *Occupies* is many-to-one, that is, each employee occupies only one office, we can represent the relationship by including the primary key of the table *Office* in the entity *Employee*. This is propagation of the key of the entity *Office* to entity *Employee*. We then obtain the database schema given in Fig. 3.11.

Employee(e_num, name, status, o_num, date)
Office(o_num, capacity, building)

Figure 3.11 Two relations representing the E-R diagram in Fig. 3.9

Note that key propagation would not work if the relationship *Occupies* is many-to-many.

As noted earlier, if the relationship *Occupies* is 1:1, key propagation is possible in both directions.

3.3.5 A More Complex Example of Mapping an E-R Diagram

We now look at a larger example. An E-R diagram for a programming database project was presented in the last chapter. We reproduce the E-R diagram in Fig. 3.12. Although it looks quite complex, it is not really as complex as one would find for a relatively small real-life database. For a database of a large company, the E-R diagram might even cover the whole wall of a large room!

We now try to map the diagram in Fig. 3.12 into the relational model using the method we have already described. We first look at all the entities and then at all the relationships.

Step 1–Strong Entities

There are five strong entities according to the Entity-Relationship diagram in the figure. These five entities are directly converted into relations as below. The relations are shown with their attributes. Note that different entities may have the same attribute names since this does not create any confusion.

1. User (FName, LName, Login_Name, Address)
2. Program (PName, Code, Date, Version, Extension, Description, Language)
3. Programmer (Login, Code, FName, LName, Address)
4. DBMS (DB_name)
5. Relation (RName, PKey)

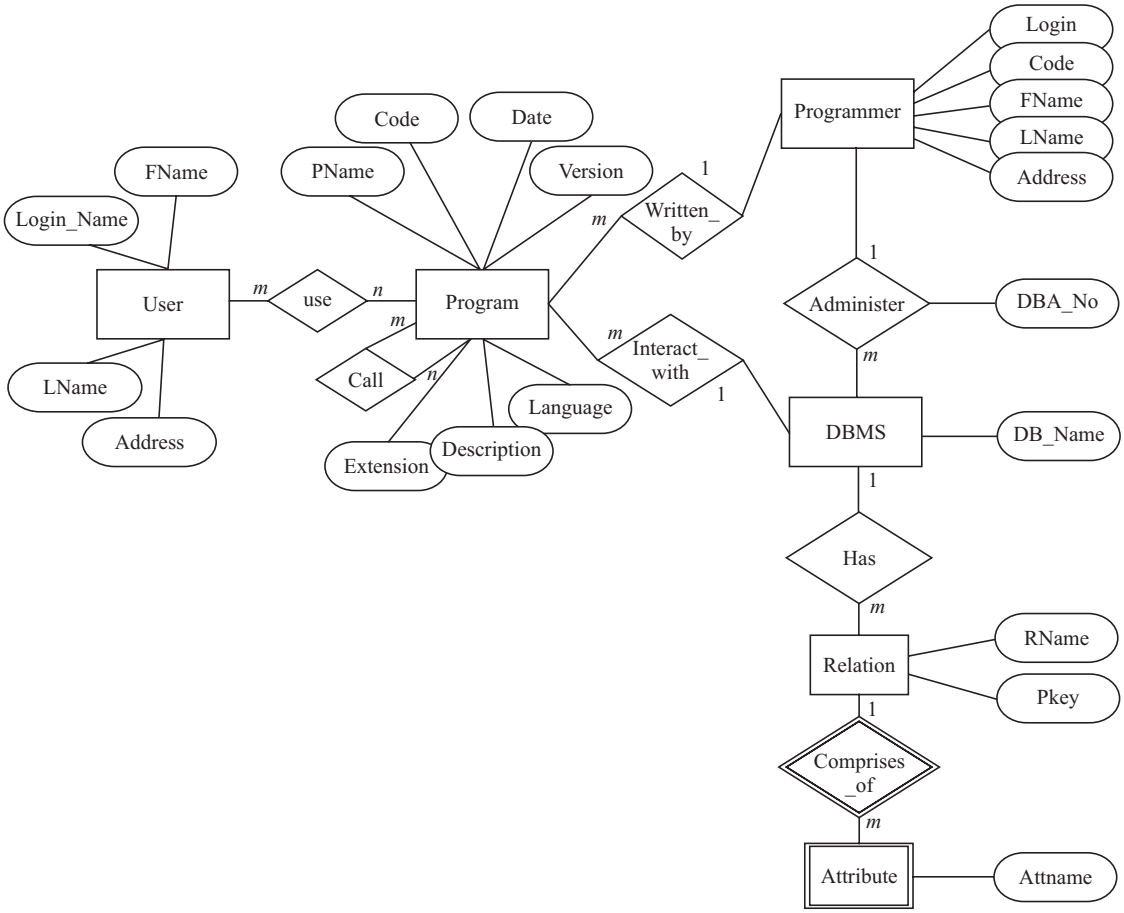

Figure 3.12 An E-R Diagram for a programming database

Step 2–Weak Entities

We do have a weak entity called attribute since it depends on there being a relation whose attributes this entity represents. Note that the primary key of the dominant entity is in this relation.

6. Attribute (RName, Attname)

Step 3(a)–One-to-one Relationships

There are no 1:1 relationships in this model

Step 3(b)–One-to-many Relationships

Two solutions are possible for each of these relationships. One possibility is key propagation. The other possibility is to map each relationship into a new relation.

(i) Either represent the relationship Written_by using key propagation which involves moving primary key of programme (*Login*) to *Program* or the relationship may be mapped to a new relation that has

primary key of both entities that it joins; Written_by (PName, Login). There is a clear assumption here that a program has only one programmer.

(ii) Either represent the relationship Interact_with by key propagation which involves moving DB_name to Program or the relationship may be mapped to a new relation that has primary key of both entities that it joins Interact_with (PName, DB_name)

(iii) Either represent the relation Administer by key propagation which involves moving Login to DBMS or the relationship may be mapped to a new relation that has primary key of both entities that it joins Administer (Login, DB_name). There is a clear assumption here that each DBMS has only one administrator.

Therefore we have the following three relations if we do not use key propagation:

7. Written_by (PName, Login)
8. Interact_with (PName, DB_name)
9. Administer (Login, DB_name)

Step 3(c)–Many-to-many Relationships

Now we deal with the four *m*:*n* relationships. All such relationships must be mapped to new tables. These tables include primary keys of both joining relations as well as any attributes that the relationship might have.

10. Use (Login_name, PName)
11. Call (PName1, PName2)
12. Has (DB_name, Relation, RName, PKey)
13. Comprises_of (RName, Attname)

This E-R model therefore translates to 13 tables in the relational model.

Once the E-R diagram has been mapped into relational tables, the tables should be created using the CREATE TABLE command discussed in Chapter 5 but an important task remains before the tables may be created. For each attribute we need to determine the attribute types.

3.4 DATA MANIPULATION

The data manipulation part of the relational model makes set processing facilities available to the user. Since relational operators are able to manipulate tables, the application programs do not need to use loops. For example, to read one row and then the next and then the one after until there are no more rows left. Avoiding loops can result in a significant increase in the productivity of application programmers.

The primary purpose of a database in an enterprise is to be able to provide information to the users within the enterprise. The process of querying a relational database is in essence a way of manipulating the tables that are the database. For example, in the cricket database presented in Tables 3.1 to 3.4, one may wish to find the names of all players from India, or a list of all the matches played by Brian Lara.

We will show how such queries may be answered by using data manipulation languages that are available for relational databases. Two formal languages were presented by Codd when he proposed the relational

model. These are called *relational algebra* and *relational calculus*. They are discussed in the next chapter, Chapter 4. These formal languages are followed by a detailed discussion of the commercial query language SQL in Chapter 5.

3.5 DATA INTEGRITY

We noted at the beginning of the chapter that the relational model has three main components: data structure, data manipulation and data integrity. The aim of data integrity is to specify rules that implicitly or explicitly define a consistent database state or change of state. These rules may include facilities like those provided by most programming languages for declaring data types which constrain the user from operations like comparing data of different data types and assigning a variable of one type to another of a different type. This is done to stop the user from doing things that generally do not make sense and to help in ensuring that the data in a database is consistent. In a DBMS, integrity constraints play a similar role.

The integrity constraints are necessary to avoid situations like the following:

1. Some data has been inserted in the database but it cannot be identified (that is, it is not clear which object or entity the data is about).
2. A player is shown to have played in a match but no data about him is available in the table *Player* that has information about players.
3. During processing a query, a player ID is compared with a match ID (this should never be required).
4. A player has retired from cricket but is still shown to be playing in a match.
5. In the table *Batting*, a player is shown to be batting at order greater than 11.
6. In the table *Batting* a player's FOW is lower than the number of runs he scored.
7. In the table *Bowling* a bowler has bowled more than 10 overs in an ODI.

We may define an integrity constraint as a condition specified on a database that restricts the data that can be stored in the database. A database system then enforces these constraints. Many different kinds of constraints may be defined on a database to overcome problems like those listed above. We discuss some constraints in this section and others in Chapter 11.

First we note that integrity constraints on a database may be divided into the following two types:

1. *Static Integrity Constraints*—These are constraints that define valid states of the data. These constraints include designations of primary keys.
2. *Dynamic Integrity Constraints*—These are constraints that define side-effects of various kinds of transactions (for example, insertions and deletions) carried out in the database. Full discussion of such constraints is outside the scope of this book but we will discuss the SQL *trigger* operation and others in Chapter 11.

We now discuss certain static integrity features of the relational model. We discuss the following features:

1. Primary Keys
2. Foreign Keys and Referential Integrity
3. Domains
4. Nulls

3.5.1 Entity Integrity Constraint (or Primary Key Constraint)

We have earlier defined the concepts of candidate key and primary key. From the definition of a candidate key, it should be clear that each table must have at least one candidate key even if it is the combination of all the columns in the table, since all rows in a table are distinct. Often a table may have more than one candidate key.

As discussed earlier, the primary key of a table is an arbitrarily but permanently selected candidate key. The primary key is important since it is the sole identifier for the rows in a table. Any row in a database may be identified by specifying the table name, the primary key and its value. Also for a row to exist in a table it must be *identifiable* and therefore it must have a primary key value. After all, each relation has information about real world things and therefore each row must be identifiable. Also no two rows in the table *Player* can have the same value of the primary key of the table. Furthermore, primary key values serve as references to the rows that they identify. Given the fundamental importance of the primary keys, the database design should always identify the primary key of a table.

Not having a primary key value specified is equivalent to saying that some entity has no identity. This is not acceptable.

We have earlier defined a candidate key of a relation as an attribute (or a set of attributes) if it satisfies the properties in Fig. 3.13.

- The attribute or the set of attributes uniquely identifies each tuple in the relation (called the *uniqueness* property).
- If the key is a set of attributes then no subset of these attributes has the uniqueness property (called the *minimality* property).

Figure 3.13 Essential properties of a candidate key

The first of these two properties of a candidate key (and therefore of the primary key) of a table requires that each key value uniquely identify only one row in the table. For example, more than one player may have the same name but they must have a unique player ID.

The second property requires that no collection of attributes that include the primary key may be called a key in spite of that collection being able to uniquely identify each tuple since they include the primary key which by itself can uniquely identify each row.

As a result of the candidate key properties listed in Fig. 3.13, the relational data model imposes the integrity constraints given in Fig. 3.14 on primary keys.

- No component of a primary key value can be null.
- No two rows can have the same primary key.
- Attempts to change the value of a primary key must be carefully controlled.

Figure 3.14 Integrity constraints for primary keys

The first constraint is necessary because when we want to store information about some entity, then we must be able to identify it to avoid difficulties. For example, if a table *Batting* has (MatchID, PID) as the primary key then allowing rows like those shown in Table 3.5 is going to lead to ambiguity since two rows

in the table may or may not be for the same player or for the same match and the integrity of the database is therefore compromised. Having NULLS for attributes other than the primary key attributes does not create any difficulties although one might find the missing information odd in this case.

Table 3.5 Problems with NULLs in primary key

MatchID	*PID*	*NOvers*	*Maidens*	*NRuns*	*NWickets*
2689	NULL	10	NULL	58	1
NULL	24001	NULL	0	27	1
2689	23001	3	0	15	0

The problem identified above may be overcome by SQL commands. For example, we consider the following CREATE TABLE command in SQL.

```
CREATE TABLE Match
      (MatchID INTEGER,
      Team1 CHAR(15),
      Team2 CHAR(15),
      Ground CHAR(20),
      Date CHAR(10),
      Result CHAR(10)
      UNIQUE (MatchID)
      CONSTRAINT MatchPK PRIMARY KEY (MatchID))
```

Figure 3.15 Applying a primary key constraint in SQL

Such commands are discussed in more detail in Chapter 5 and Chapter 12.

The second constraint deals with changing of primary key values. Since the primary key is the identifier of an entity instance, changing it needs very careful controls. For example, consider the information in the fragment of table *Bowling* in Table 3.6. Suppose we decide to change the primary key of the first row from (2689, 24001) to (2690, 24005) for some reason (it is difficult to think of a reason in this instance). Clearly this should not be allowed without strict controls.

Table 3.6 Problems with changing the primary key

MatchID	*PID*	*NOvers*	*Maidens*	*NRuns*	*NWickets*
2689	24001	3	0	27	1
2689	23001	3	0	15	0
2755	91001	4	0	29	0

Codd has suggested the following three possible approaches:

Method 1

Only a select group of users should be authorized to change primary key values.

Method 2

Updates on primary key values are banned. If it was necessary to change a primary key, the row should first be deleted and a new row with a new primary key value but the same other values should be inserted. Of course, this does require that the old values of attributes be remembered and be reinserted in the database.

Method 3

A different command for updating primary keys is made available in the DBMS. Making a distinction between altering the primary key and another attribute of a table would remind users that care needs to be taken in updating primary keys.

3.5.2 Foreign Keys and Referential Integrity

We have discussed some problems related to modifying primary key values. Additional problems can arise in modifying primary key values because one table may include references to another. These references include the primary key value of a row which may be referred to in one or more other tables of the database. The integrity constraints generally include that every value of the primary key of one table (sometimes called the *referenced* table) that appears in another table (i.e., the *referencing* table) must occur as a value of the primary key in the referenced table. Furthermore, when a primary key is modified (for example, a row from the table might be deleted), each of these references to a primary key must either be modified accordingly or be replaced by NULL values. Only then can we maintain referential integrity.

Before we discuss referential integrity further, we define the concept of a *foreign key*. The concept is important since a relational database consists of tables only (no pointers) and relationships between the tables are implicit, based on references to primary keys of other tables. These references are called *foreign keys*.

Definition—Foreign Key
The foreign key in table R is a set of columns (possibly one) whose values are required to match those of the primary key of some table S. R and S are not necessarily distinct.

It should be noted that the foreign key and primary key must be defined on the same domains. As an example, let us consider the database schema in Fig. 3.16.

Match(MatchID, Team1, Team2, Ground, Date, Winner)
Player(PlayerID, LName, FName, Country, YBom, BPlace, FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRuns, Mts, NBalls, Fours, Sixes)
Bowling(MatchID, PID, NOvers, Maidens, NRuns, NWickets)

Figure 3.16 Database schema for the ODI cricket database

We now note the following:

1. The table *Match* has the primary key MatchID but no foreign key.
2. The table *Player* has the primary key PlayerID but no foreign key.
3. The table *Batting* has the primary key (Match ID, PID) and foreign keys MatchID and PID.
4. The table *Bowling* has the primary key (Match ID, PID) and foreign keys MatchID and PID.

Several questions now arise. What do the foreign keys refer to? Could (MatchID, PID) be foreign keys as well as primary keys in tables *Batting* and *Bowling*?

Well, the answers are relatively straightforward. MatchID is referring to the primary key of the table *Match* while PID is referring to the primary key of the relation *Player*. (MatchID, PID) together are not foreign keys because *Batting* and *Bowling* do not refer to each other.

In some cases like the following, there may be a foreign key that refers to its own table, for example, if the *supervisor-no* is an *empno*.

Employee (empno, empname, supervisor-no, dept)

Therefore, foreign keys are the implicit references in a relational database. The following constraint is called the referential integrity constraint:

Definition—Referential Integrity
If a foreign key F in table R refers to and matches the primary key P of table S then every value of F must either be equal to a value of P or be wholly NULL.

The justification for the referential integrity constraint is simple. If there is a foreign key in a relation (that is, the table is referring to another table) then the foreign key value must match with one of the primary key values in the table to which it refers. That is, if an object or entity is being referred to, the constraint ensures the referred object or entity exists in the database. It is not required that the foreign key values be always non-null. Sometimes a foreign key may be null, for example, a supervisor may not yet be assigned for an employee. Often a foreign key may be a component of a primary key and then the existential integrity constraint prohibits the appearance of nulls.

A foreign key in some DBMS may refer the column(s) of another table that have a combined UNIQUE constraint as well as the NOT NULL constraint.

Just as primary key updates need to be treated in a special way, one must deal with foreign key updates carefully. Foreign key updates do not need to be restricted in quite the same way as primary key updates. The system does need to ensure that the update values appear in the referenced table. Otherwise the update should be rejected.

The foreign key constraint may be specified in SQL as given in Fig. 3.17.

```
CREATE TABLE Match
(MatchID INTEGER,
Team1 CHAR(15),
Team2 CHAR(15),
Ground CHAR(20),
Date CHAR(10),
Winner CHAR(10)
UNIQUE (MatchID)
PRIMARY KEY (MatchID)
FOREIGN KEY (MatchID) REFERENCES Match
FOREIGN KEY (PID) REFERENCES Player)
```

Figure 3.17 Applying the foreign key constraint in SQL

3.5.3 Domains

We have noted that not all commercial database systems provide facilities for specifying domains. Domains could be specified as in Table 3.7.

Table 3.7 Specifying domains

CREATE DOMAIN NAME1	CHAR(10)
CREATE DOMAIN Player1	INTEGER
CREATE DOMAIN NAME2	CHAR(10)
Etc.	

Note that *NAME1* and *NAME2* are both character strings of length 10 but they belong to different (semantic) domains. It is important to denote different domains to

(a) constrain unions, intersections, differences and equi-joins of tables, and
(b) let the system check if two occurrences of the same database value denote the same real world object.

The constraint on union-compatibility and join-compatibility is important so that only those operations that make sense are permitted. For example, a join on class number and student number would make no sense even if both attributes are integers. The user should not be permitted to carry out such operations (or at least be warned when it is attempted).

3.5.4 Nulls

The issue of null values in a database is extremely important. A value in a database may be null for several reasons. The most obvious reason is that the value is not known at the present time. For example, a new employee's home telephone number will not be known until a phone is connected to his/her residence. Another reason why some values in a database may be null is that the attribute may not be applicable. For example, an employee database may have an attribute spouse which is not applicable for a single employee. For female employees the maiden name column might be inapplicable if the employee is single or if the employee did not change her name after marriage. In yet another situation, some information may not be known and will never be known. It is possible that an older migrant or a refugee may not know their date of birth. The person may not even know where he/she was born. We know of a situation when a student did not even know what his nationality was. His mother was from Bangladesh and his father from Burma and neither of the two countries was willing to provide him with a passport!

Having null values in a database creates a variety of issues beginning from simple ones like how to define the average of a collection of values if some are null to comparing null values with non-null values.

Further discussion of this interesting and complex issue is beyond the scope of this book.

3.6 ADVANTAGES OF THE RELATIONAL MODEL

Codd in 1970 indicated that there were four major objectives in designing the relational data model. These are

1. *Data Independence*—To provide a sharp and clear boundary between the logical and physical aspects of database management.

2. *Simplicity*—To provide a simpler structure than was being used at that time. A simple structure is easy to communicate to users and programmers and a wide variety of users in an enterprise can interact with a simple model.
3. *Set-Processing*—To provide facilities for manipulating a set of records at a time so that programmers are not operating on the database record by record.
4. *Sound Theoretical Background*—To provide a theoretical background for the database management field.

3.7 RULES FOR FULLY RELATIONAL SYSTEMS

If a system supports all relational algebra operators and the two integrity constraints (entity integrity and referential integrity) then the system may be called *fully relational*. Codd provided 12 rules (plus rule 0) to evaluate the properties of a relational model. A fully relational DBMS needs to satisfy these rules. The rules are

1. *Rule* 0—For any system that is advertised as, or claims to be, a relational database management system, that system must be able to manage databases entirely through its relational capabilities.

   This basic rules states that all database management should be based on the relational model. The rule is included only because when relational database systems were becoming popular, some of the nonrelational database systems started advertising their systems as relational systems.

Data Structure Rules

2. *Rule* 1—All information in a relational database is represented explicitly at the logical level and in exactly one way, by values in tables.

   There are many advantages in having this simple data structure. The data manipulation language can be simple. Additional software that interfaces with a relational DBMS can be based on the assumption that all data is represented in tables. The early nonrelational systems had much more complex data structures and the application programs were then also complex.
3. *Rule* 2—Each and every datum (atomic value) in a relational database is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name.

   This rule requires that the only way to access data in a relational model is by specifying the name of the table, the column(s) and its primary key value(s). No other traversing of the database, for example row-by-row, should be required.
4. *Rule* 3—Null values (distinct from the empty character string or a string of blank characters are distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.

   Most database systems have some data that is unavailable or inapplicable. The relational model requires that this data be treated more systematically than by defining a special value, for example -9999, for each attribute.
5. *Rule* 4—The database description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.

This rules deals with the database metadata which should be able to be manipulated in the same way as the regular data making it easier for database users to interrogate it.

Data Manipulation Language Rules

6. *Rule* 5—A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible using some well-defined syntax, as character strings and that is comprehensive in supporting all of the following items:

    - Data definition
    - View definition
    - Data manipulation (interactive and by program)
    - Integrity constraints
    - Transaction boundaries (such as begin, commit and roll-back)

    This rule requires that the user has access to at least one language that carries out all relational database operations
7. *Rule* 6—All views that are theoretically updatable are also updatable by the system.

    This is a difficult requirement since it is not easy for a database system to determine if the update request by the user to update a view is reasonable.
8. *Rule* 7—The capability of handling a base table or a derived table as a single operand applies not only to the retrieval of data but also to the insertion, update and deletion of data.

    As we will show in the next two chapters, insertion, update and deletion are able to use a table as a single operand enabling a set of tuples to be inserted, updated or deleted in one command.

Data Independence Rules

9. *Rule* 8—Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods.

    This rule deals with data independence, one of the most important characteristic of a relational database system. It may be defined as the ability to separate the database definition from the physical storage structure of the database. Therefore any changes in the physical structure of the database should not impact the application programs.
10. *Rule* 9—Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit unimpairment are made to the base tables.

    This again relates to data independence.
11. *Rule* 10—Integrity constraints specific to a particular relational database must be definable in the relational data sub-language and storable in the catalog, not in the application programs.

    This rule requires that integrity constraints should not be included in the application programs and therefore application programs are easier to write. Also, it ensures that all application programs are subjected to exactly the same integrity constraints.
12. *Rule* 11—A relational DBMS has distribution independence.

    This rule relates to distributed database systems which are discussed in Chapter 13.

13. *Rule* 12—If a relational system has a low level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational languages (multiple-records-at-a-time).

    This rule essentially requires that no low level language be used since that may bypass the rules of a relational system.

3.8 EARLIER DATABASE MODELS – HIERARCHICAL AND NETWORK MODELS

As noted earlier, the relational database model replaced the earlier models in the1980s. Two database models that were popular in the late 1960s and 1970s and now superseded by the relational DBMS is also now discussed. Some database systems even today use these old database systems. IMS, IDMS and TOTAL are examples of DBMS that were based on these models and were used widely. Some DBMS that were not based on either of these two models were also in use in the 1970s. These included ADABAS from a German company called Software AG. ADABAS was an inverted list database.

3.8.1 Network Data Model

The network model was developed by Charles Bachman and his colleagues at the General Electric Company in 1964. The model was standardized in 1971 by the CODASYL group (Conference on Data Systems Languages). The model used directed acyclic graphs (called Bachman diagrams) with nodes and edges where nodes represent the entity sets and edges (or links) represent the relationships between the entity sets. The links could be either 1:1 or 1:*m* so many-to-many relationships had to be decomposed into simpler relationships. The model allowed each child record to have multiple parents allowing 1: *n* relationships. The major problem with the network model was the complexity of designing a database system and its maintenance.

In the network model, we can represent each player as a node and their batting and bowling records also as nodes. They are then connected by edges or links. Finding batting and bowling information for a player was relatively easy since it only required searching for the relevant player node record and then finding his bowling and batting records by following the edges. Finding the names of players who scored a century involved searching the nodes for batting and finding nodes that had a score of more than 99 and then following the edges to find player nodes.

As noted above, a number of widely used database systems were based on the network model. These included IDMS (Computer Associates), IDS II (Honeywell) and DMS-1100 (Unisys).

Some typical commands used in a network database system are presented in Fig. 3.18.

A simple example of a network model database is presented in Fig. 3.19.

Command Name	Description
FIND	Locate record
GET	Transfer record to memory
OBTAIN	Combine FIND and GET
STORE	Insert a new record
ERASE	Delete record

Figure 3.18 Examples of commands used in a network database model

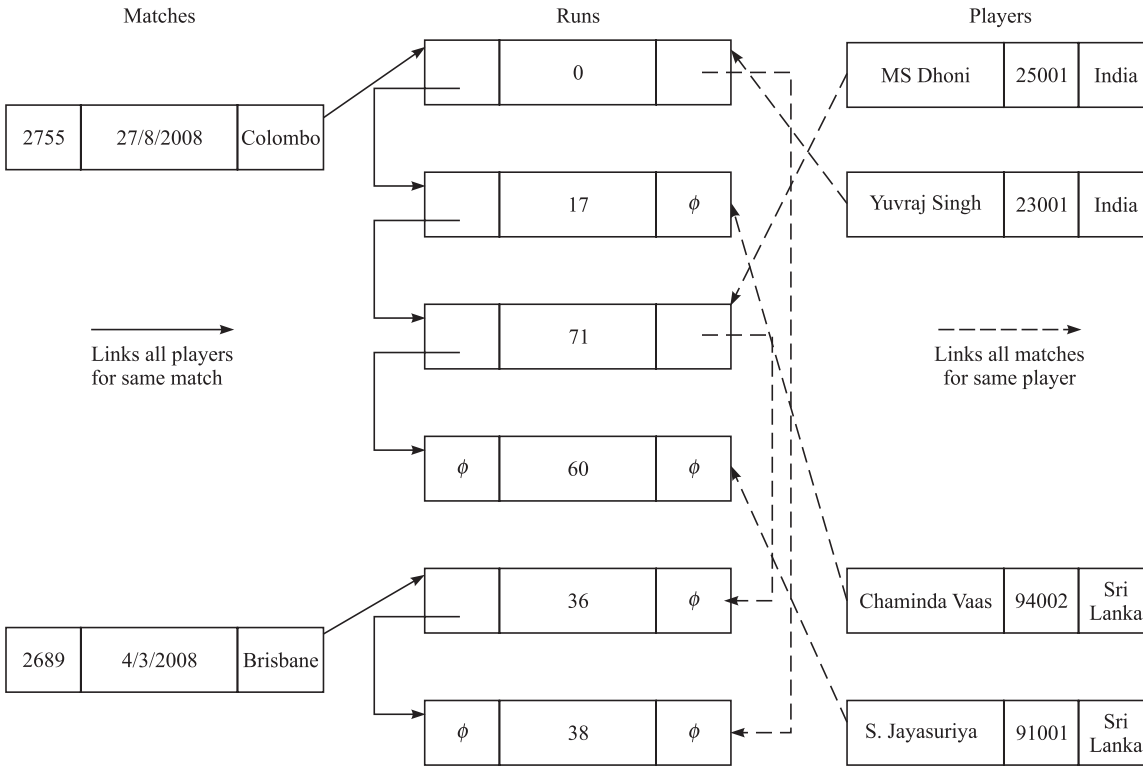

Figure 3.19 A typical network database model

Figure 3.19 shows a few records in a network model for the cricket database. There are two matches at the left-hand side and four players on the right-hand side. The players and matches are connected by the records in the middle which gives details of batting. We have only shown runs in these records but other information could be included. All the batting records of a match are linked in a chain with solid lines while all the batting records of a player are linked by the dotted lines on the right-hand side. Similar records could include all the bowling information. These chains allow one to traverse through all batting records of a match as well as all batting records of a player and these chains must be followed if one wanted to know how many runs Jayasuriya scored in match 2755. This is clearly very different than what happens in the relational model.

We leave it as an exercise for the reader to work out how insertions, deletions and updates take place in the two models and also compare these models with the relational model.

In summary, the network model was conceptually simple, efficient for some database problems and provided sufficient flexibility in that all relationships could be represented without much redundancy. Given that the network model was navigational, the performance of network DBMS was generally not high and some changes to the underlying database were difficult to implement. For example, changing relationships required physical reorganization of the data as well as changes to the application programs and therefore provided only limited data independence. Also these early DBMS were not robust. For example, a failure in the system occasionally left dangling references to data which had to be recovered.

3.8.2 Hierarchical Data Model

The hierarchical model was another model that formed the basis of the earliest database systems. It was a somewhat more restrictive model and could be considered a subset of the network model.

The hierarchical model was developed at IBM and was the basis of the IMS database system. IBM never supported the network model. IMS was based on the thinking that all views in life are hierarchical and therefore any database application could also be modelled as a hierarchy. The basic building block of the hierarchical model was called a *Physical Database Record* (PDBR) which consisted of a root segment and a hierarchy of segments (called *child segments*). Each segment was like a row in a table consisting a set of related fields. Each PDBR was defined by a database description (DBD) that was coded by a programmer.

Since IBM already had some file processing packages, the IMS hierarchical model used them for storage and retrieval of data. These included the Sequential Access Method (SAM), Indexed Sequential Access Method (ISAM), Virtual Sequential Access Method (VSAM) and Overflow Sequential Access Method (OSAM). The IMS data manipulation used Data Language I (DL/I) that had a set of commands used with a host language like COBOL. DL/I commands essentially involved traversing the hierarchical database structure.

The hierarchical model was based on a tree structure which may be defined as follows.

Definition—Tree
A tree is a hierarchical structure that is a finite set of nodes connected by edges such that there is one specially designated node called the root of the tree. The remaining nodes are partitioned into n ($n \geq 0$) disjoint sets, each of which is also a tree.

It represented data as relationships between tree nodes called parent records (also called *segments* in the hierarchical model) and child records (or segments) as a tree. These relationships between the two record types may only be one-to-many. For example, we could represent each player as a parent segment and their batting and bowling records as child segments. Finding batting and bowling information for a player is relatively easy since it only requires searching for the relevant parent record and then finding his bowling and batting child records, but many other types of queries are more difficult. For example, finding names of players who scored a century in Melbourne is more difficult because it involves going through each player's parent record then searching the child records to check if there is a record that has more than 99 runs and is about a match in Melbourne. In summary, the following should be noted about the hierarchical model:

1. The model was efficient for searching the parent records and their child records since there was less redundant data.
2. Searching through the child records was more complicated and very different than in the relational model.
3. There were difficulties in representing many-to-many relationships.

Some commands used in IMS, a well-known hierarchical database system, are presented in Fig. 3.20.

In using the hierarchical model to represent the cricket database including the tables *Player*, *Match*, *Batting* and *Bowling* difficulties arise because the hierarchical model assumes all information to be hierarchical in nature. In the cricket database it is not clear which entity may be considered the parent entity and which as the child entity.

Command Name	Acronym	Description
GET UNIQUE	GU	Retrieve record
GET NEXT	GN	Sequential retrieval
GET NEXT WITHIN PARENT	GNP	Sequential retrieval of same parent
DELETE	DLET	Delete existing segment
INSERT	ISRT	Add new segment

Figure 3.20 Examples of commands used in a hierarchical database model

We will attempt to model the information assuming that the *Match* entity is parent, *Player* is the child of the entity *Match* and *Batting* and *Bowling* are child entities of *Player*.

The cricket database in the hierarchical model is presented in Fig. 3.21.

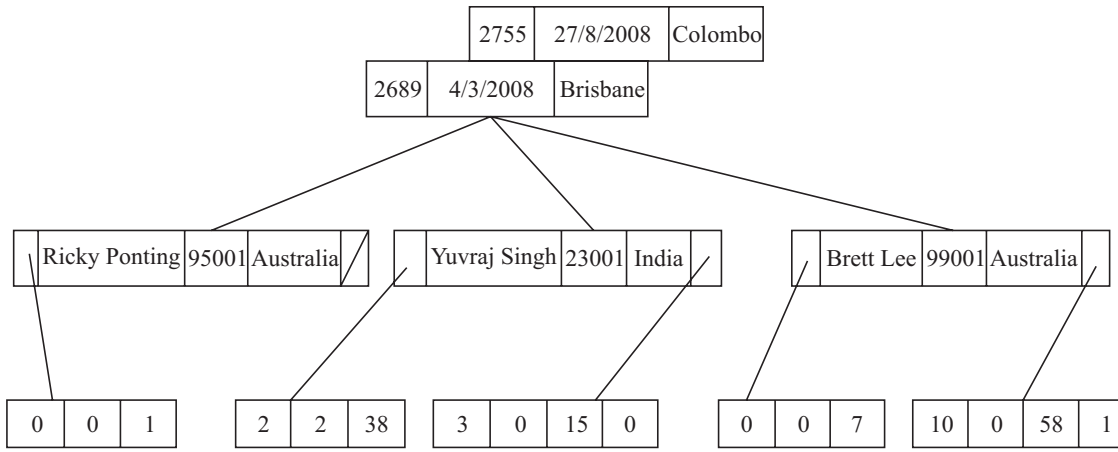

Figure 3.21 A typical hierarchical database model

Figure 3.21 shows only a few records. There are two matches at the root. One of them is shown to have three children players that the parent entity points to and each player has one or two children which are the batting or bowling records of the player. The first field is a pointer to the batting child record and the last field is a pointer to the bowling child record.

Given the above structure, the hierarchical model requires the following:

1. A parent record type may own an arbitrary number of chid record types.
2. No single occurrence of a record type may have more than one parent record type.

In summary, software based on the hierarchical model provided an efficient representation of hierarchical structures, efficient single key search (if the hierarchical structure was the application view of the data), fast update performance for some updates. It was a conceptually simple model. On the other hand, hierarchical database systems had difficulty with many-to-many relationships, it lacked flexibility (nonhierarchical relationships were difficult to represent), performance for nonhierarchical accesses was poor, lack of maintainability, for example, changing relationships often required physical reorganization of data as well as changes to the application programs.

3.8.3 Comparison of Hierarchical, Network and Relational Models

Both the network and hierarchical models were developed in the 1960s and were conceptually simple when they were proposed but suffered from a number of serious disadvantages partly because both were navigational systems. The disadvantages included low level of data independence, difficulty in representing many-to-many relationships. For a large database, the logical data model became very complex and since the data manipulation languages were procedural the application programs became complex as well and difficult to maintain. In both hierarchical and network models, physical and logical data independence was difficult to achieve, since the data manipulation languages and therefore the application programs often contained a wide variety of information about the physical organisation of the database.

Hierarchical model's single parent rule forces redundancy. It can be eliminated in the network model since the network model allows more flexibility and a record type may have an arbitrary number of parents.

The three models are compared in Fig. 3.22 below.

Item	Hierarchical	Network	Relational
Data structure	Hierarchy	Tree of records	Tables
Data Access	Navigational	Navigational	Non-navigational
Method of access	Procedural First access the root and then navigate to the record of interest.	Procedural Data access may start from any entity.	Nonprocedural SELECT, FROM, WHERE
User must know	Structure of the database	Structure of the database	Names of the tables
Performance	Was fast for some simple queries	Was fast for some simple queries	Good for many types of queries
Mathematical basis	None	None	Has mathematical foundation
Query language	Use Dl/I through COBOL	COBOL based commands	SQL
Data integrity	None	None	Basic integrity is part of the model
Modifying the database	Difficult	Difficult	Relatively easy
Data independence	None	None	Part of the model

Figure 3.22 A comparison of the three database models

SUMMARY

This chapter has covered the following points:

- The relational data model was invented by E. F. Codd in 1970. The three major components of the model are data structure, data manipulation and data integrity.
- Data structure component of the relational model required that all information presented in a database be as relations (essentially tables).
- Each relation must meet a number of properties, for example, atomic values, no duplicate tuples, no ordering of tuples, etc.
- Relational terminology includes terms like a relation (or a table), tuple (or a row), degree (number of columns), cardinality (number of rows), attribute (column), domain, candidate key and primary key.
- The primary key of a relation has the properties of uniqueness and minimality.
- An Entity-Relationship model discussed in Chapter 2 may be mapped to relational model. Mapping includes mapping each entity and each many-to-many relationship to a table.
- Data integrity in the relational model includes constraints relating to primary keys (called existential integrity) and foreign keys (called referential integrity) as well as domains and NULLs.
- Relational data model has the advantages of data independence, simplicity, set processing and sound theoretical background.
- Codd defined 13 rules that a fully relational model must satisfy including data structure rules, data manipulation rules and data independence rules.
- The basic concepts of two older database models, the network model and the hierarchical model, have been presented. The hierarchical model is based on the concept that all data has hierarchical nature.
- Disadvantages of the hierarchical and network database models have been described and the models are compared to the relational model. These include different data structures, different database languages, lack of integrity in the older models, and low level of data independence in the older models.

REVIEW QUESTIONS

1. What are the major components of the relational model? (Section 3.1)
2. What is the meaning of the terms relation, tuple, degree, cardinality, attribute, domain, candidate key and primary key? (Section 3.2)
3. Give two example to illustrate the properties of a primary key? (Section 3.2)
4. Present an E-R model and show how it should be mapped to the relational model. How is an entity mapped? How is a relationship mapped? (Section 3.2.1)
5. Describe constraints related to the primary key, foreign key, domains and NULLs. (Section 3.4)
6. List some of the advantages of the relational model. (Section 3.5).
7. List five rules for fully relational systems. (Section 3.6)
8. Describe the main features of the hierarchical data model. (Section 3.7)
9. Describe the main features of the network data model. (Section 3.7)

10. Using an example, illustrate the features of the hierarchical and the network model. List some of the disadvantages of these models. Why do you think the products based on these models could not compete with the products based on the relational model? (Section 3.7)

SHORT ANSWER QUESTIONS

1. What are the three components of the relational model?
2. List three properties of a relation.
3. What is the data structure used in the relational model?
4. List three major properties that a relation must have?
5. What is the cardinality of a relation? What is the cardinality of the relation in Fig. 3.3?
6. What is the degree of a relation? What is the degree of the relation in Fig. 3.2?
7. Define the candidate key of a relation.
8. How is an entity in an E-R model mapped to the relational model?
9. How is a many-to-many relationship in an E-R model mapped to the relational model?
10. How is a one-to-one relationship in an E-R model mapped to the relational model?
11. How is a weak entity in an E-R model mapped to the relational model?
12. What is a primary key and existential integrity?
13. What is a foreign key and referential integrity?
14. Why DBMS products based on the network model and the hierarchical model were not able to compete with relational DBMS?
15. What is the major difference between the network model and hierarchical model?

MULTIPLE CHOICE QUESTIONS

1. The main components (more than one) of the relational model are
   (a) data structure
   (b) data manipulation
   (c) data recovery
   (d) data integrity
2. Which one of the following is correct?
   (a) A relation may have duplicate rows.
   (b) A relation may have duplicate columns.
   (c) A relation may have ordering associated with the rows.
   (d) A relation may have non-atomic attribute values.
3. Which one of the following is allowed in the relational model?
   (a) Repeating groups
   (b) Pointers
   (c) Record identifiers other than the primary key
   (d) Specification of domains

4. Which two of the following are correct? Rows in the relational data model are required to be unordered because
   (a) it makes the database more efficient.
   (b) the user should not be burdened with having to remember which row is next to which row.
   (c) numbering rows is expensive.
   (d) changes in ordering may have to be made for a variety of reasons.
5. Which one of the following is correct? In using a relational database consisting of a number of tables, a user only needs to know
   (a) the number of rows in each relation.
   (b) the number of columns in each relation.
   (c) the access paths to the data in each relation.
   (d) the names of the tables and the names of their relevant columns.
6. The concept of domain is fundamental to the relational data model. Which of the following statements regarding domains is **not** correct?
   (a) Conceptually, domains play a role similar to data types in programming languages.
   (b) Domains are user defined while the data types are built into the system.
   (c) A domain is a set of scalar or atomic values, all of the same type.
   (d) There can be no more than one attribute in each relation defined on each domain.
7. Which two of the following properties must belong to the primary key of a relation?
   (a) It identifies each row uniquely.
   (b) It is an attribute or a set of attributes on which an index is built.
   (c) It is an attribute or a set of attributes for which missing values are not allowed.
   (d) A proper subset of it cannot be a key.
8. Which one of the following is correct?
   (a) Each candidate key is a primary key.
   (b) Each primary key is a foreign key.
   (c) Each foreign key is a primary key in some relation.
   (d) Each foreign key is a candidate key.
9. Which one of the following is correct?
   (a) Referential integrity requires that a foreign key may not be NULL.
   (b) Referential integrity requires that a foreign key may not be partially NULL.
   (c) Existential integrity requires that a primary key may not be NULL although it may be partially NULL.
   (d) A high level of integrity requires that there be no NULL values in the database.
10. Consider the following relation that stores information about offices, telephones and occupants of offices:

    office(OfficeNum, TelephoneNum, Occupant)

    Each office has a unique office number.

    There is only one telephone in each office and the telephone number is unique. Each office is occupied by only one person although a person may have more than one office. Occupant is a unique identification for persons occupying rooms.

Which one of the following is correct?

(a) OfficeNum and TelephoneNum are both candidate keys.
(b) OfficeNum, TelephoneNum and (OfficeNum, TelephoneNum) are all candidate keys.
(c) (OfficeNum, occupant) and (TelephoneNum, occupant) are the candidate keys.
(d) OfficeNum must be the primary key.

11. Consider again the relation *office(OfficeNum, TelephoneNum, Occupant)*. Now assume that each office may have more than one occupant and more than one telephone and a telephone may be shared between two or more occupants. Furthermore, each occupant may have more than one office. OfficeNum, TelephoneNum and occupant names are unique.

Which one of the following is correct?

(a) OfficeNum and TelephoneNum are both candidate keys.
(b) OfficeNum, TelephoneNum and OfficeNum, TelephoneNum are all candidate keys.
(c) (OfficeNum, occupant) and (TelephoneNum, occupant) are the candidate keys.
(d) (OfficeNum, TelephoneNum, occupant) is the candidate key.

12. Which of the following are true? The value of an attribute in a table may be NULL because

(a) the value is not known.
(b) the value is not applicable.
(c) the value will never be available.
(d) All of the above

13. Which one of the following is correct?

(a) Every subset of a relation is a relation.
(b) Every subset of a tuple is a tuple.
(c) Every subset of an attribute is an attribute.
(d) Every subset of a foreign key is a foreign key.

14. Which of the following are included in Codd's rules for fully relational systems?

(a) NULL values are independent of data type.
(b) All views that are theoretically updatable are also updatable by the system.
(c) A relational database has distribution independence.
(d) All of the above

15. Which of the following are correct?

(a) Hierarchical model was developed before the relational model.
(b) Object-oriented database model was developed before the relational model.
(c) The network database model was developed before the relational model.
(d) The Entity-Relationship model was developed before the relational model.

16. Which one of the following is **not** correct?

(a) The network model used acyclic graphs with nodes and edges.
(b) The hierarchical model was based on a tree structure in which there was one specially designated node called the root of the tree.
(c) Hierarchical model had difficulty representing many-to-many relationships.
(d) The network model is navigational while the hierarchical model is not.

17. When a row is deleted which one of the following techniques should be used to maintain integrity?
    (a) The row is deleted and nothing else is done.
    (b) The row is deleted and the references to the deleted primary key, if any, are replaced by NULL.
    (c) The delete operation is not allowed if the row's primary key is a target of a foreign key.
    (d) The row is deleted as well as the rows from other tables that have foreign keys that have the deleted primary key as their target.

EXERCISES

1. What are the properties of a relation? Discuss each property and explain why it is important.
2. Define a candidate key, a primary key and a foreign key. Can the primary key have null values? Can the foreign key have null values? Can one relation have more than one candidate key? Can one relation have more than one primary key? Can one relation have more than one foreign key? Explain and give examples to illustrate your answers.
3. Discuss the following concepts and explain their importance.
    (a) Integrity constraint
    (b) Referential integrity
    (c) Existential integrity
4. Relate each of the relational terms on the left-hand side to their informal equivalents on the right-hand side. Note that not all terms on the right-hand side have a matching term on the left-hand side.

Table 3.8 Terminology

Relation	*Pointer*
Tuple	Table
Primary key	Record or row
Candidate key	Link
Attribute	Field or column
Domain	File
Cardinality	A unique identifier
Degree	Pool of legal values Current values Number of attributes in a table Number of tuples Number of fields in the primary key

5. Consider the following student database:
   Student(ID, Name, Address, City, Tutor)
   Enrolment(ID, Code, Marks)
   Course(Code, Title, Department)

Formulate the following queries in relational algebra using the student database above. Assume that the attribute *Tutor* is the ID of the tutor.

(a) List all the student names.
(b) List all school and course names.
(c) Find student names of students that have no tutor.
(d) Find student names of students that are doing computer science courses.
(e) Find student names of students that have Suresh Chandra as tutor.

6. Consider the following relations in a student database:

Table 3.9 The relation Student

ID	*Name*	*Address*	*City*	*Tutor*
8700074	Svaty Krishna	9, Gandhi Hall	Green Park	NULL
8900020	Arun Kumar	90, Nehru Hall	Green Park	8700074
8801234	Preeti Mukerjee	80, Tata Hall	Green Park	8612345
8612345	Kamal Gupta	11, College Street	Noida	NULL
9082545	Praveen Kothari	54, M. G. Road	Chennai	8700074
9038593	B. V. Narasimhan	25 Chandni Chowk	Delhi	8612345

Table 3.10 The relation *Enrolment*

ID	*Code*	*Marks*
8700074	CS100	72
8900020	MA111	54
8900020	CS150	NULL
8700074	CH100	NULL
8801234	CS300	89
8801234	BU299	NULL

Table 3.11 The relation *Course*

Code	*Title*	*Department*
CS300	Database Management	Computer Science
CS100	Introduction to Computer Science	Computer Science
CH100	Introduction to Chemistry	Chemistry
MA111	Linear Algebra	Mathematics
BU150	Introduction to Economics	Business
BU299	Project	Business

For each relation, write the degree of the relation, its cardinality, all the candidate keys and the primary key.

7. Consider the following two relations in an example database with EmpID and DeptID as primary keys of the two tables respectively:

Table 3.12 The table Emp

EmpID	*Ename*	*ManagerID*	*DeptID*
126	Krishna Kumar	129	22
NULL	Prakash Patel	NULL	22
129	Milkha Singh	NULL	26
135	Rabindra Singh	173	99
149	Nizam Khan	129	NULL
173	David Abraham	NULL	NULL

Table 3.13 The table Dept

DeptID	*Dname*	*MgrID*
22	Sales	128
26	Production	126
75	Finance	129
81	Deliveries	149
89	NULL	173

   (a) Identify all foreign keys.
   (b) Do the above relations satisfy the existential and referential integrity constraints? Discuss.
   (c) If the tuple with EmpID = 129 in the *Emp* table is deleted, what should be done to maintain integrity of the database?

8. Consider the cricket database in Tables 3.1 to 3.4. List some integrity constraints that should apply to these tables.
9. A simple database is to be designed to store information about rooms and telephone numbers in an enterprise. The only information to be stored is room numbers and telephone numbers in each of them. Room numbers and telephone numbers are assumed to be unique.

   One way to represent the information is to have a relational schema *room(RoomNum, PhoneNum)*. Is this relation adequate to represent the information in each of the following cases? If not, present an alternative database design consisting of one or more relational schemas for each of the cases below.

   Consider the following four situations:

   (a) Each phone number is assigned to only one room and each room has a unique phone number.
   (b) A phone number may (with extensions) be assigned to more than one room and a room may have more than one phone each with a different number.

(c) Each phone number is assigned to only one room but some rooms do not have any phones while others may have one or more phones.
(d) Each phone number is assigned to only one room and each room has at least one phone. Some rooms have more than one phone.

PROJECTS

We now present a number of projects that may be used by instructors as a basis for designing projects for their course. We have used similar projects successfully for many years in our classes. A methodology for projects which may suit some instructors is proposed below.

Project Implementation Suggestion
It is proposed that each project should consist of the following three stages:

Stage 1
In the first stage, students are required to design an Entity-Relationship model for one (possibly allocated by the lecturer) of the database applications described below as well as others that are suggested by the lecturer.

Students should make suitable assumptions where necessary when implementing their project.

The following should be included with the Entity-Relationship diagram:

1. A discussion of the needs of the enterprise. A collection of typical queries that the system is expected to answer.
2. A list of all assumptions.
3. A list of all entities and relationships and their attributes and primary keys (most of this information can be included in the Entity-Relationship diagram).
4. A discussion of the strengths and weaknesses of the proposed solution.
5. And, of course, the complete Entity-Relationship diagram.

The design and the accompanying documentation should be submitted by the given date.

Stage 2
Revise the E-R diagram based on comments received from the lecturer. Build a set of ten sensible queries, some easy, some more complex that use subqueries, GROUP BY and aggregation functions. Submit these by the due date for Stage 2.

Stage 3
Implement the database using Oracle (or another system that is available). Populate the database. Formulate the queries and run them using the database. Submit a report including the results of the queries by the given date. The report should clearly identify objectives, discuss methodology, present assumptions and present results and discuss conclusions.

Bonus marks may be given to students who build a suitable user interface for their system.

A significant component of the assignment marks should be awarded on the basis of the quality of documentation.

1. A Small Airport

Design a database for a small airport, for example, in a place like Haridwar in Uttarakhand or Puri in Orissa. The database is designed to keep track of private airplanes, their owners, airport employees and pilots. Each airplane has an identification number, is of a particular type and has a parking space in a particular hangar. Each plane has a purchase date, its type has a model number, and a capacity in terms of number of passengers. Each hangar has a number and the number of planes that may park in it. The database also needs to keep track of owners of the planes and the employees that have worked on each plane and on what date. The maintenance record of each plane that includes the number of hours spent servicing the plane and a code for the type of service done. Some planes provide charter services to large companies like Infosys or Reliance. The database stores details of each such flight.

2. College of Pharmacy

Design a database system for a College of Pharmacy, Faculty of Medicine of the North Indian University (NIU) located in Mussorie[2]. The faculty has research projects in various stages of development, i.e., current, pending and complete. Each project is funded by a single grant by the Indian Government. Usually, the major portion of this research grant is used to pay the salaries of research assistants and study's subjects. The various drugs and equipment used in the experiments are often provided by one or more drug companies, for example, Dr. Reddy's Pharmaceutical company or Ranbaxy Laboratories Ltd. A project studies the effects of the drug on many subjects, some of whom have unequal therapeutic regimens. Several employees of the College of Pharmacy work on each project, which is led by a principal investigator (PI). Each principal investigator may control several projects. When a study is completed, a research report is written describing the results of the study.

3. Bus Company Routes

Design a database system for managing information about routes supported by a bus company called The All India Bus Company with head office in Nagpur. Each route served by the company has a starting place and an ending place, but it can go through several intermediate stops. The company is distributed over several branches. Not all the cities where the buses stop have a branch. However, each branch must be at a city located along the bus routes. There can be multiple branches in the same city and also multiple stops in the same city. One bus is assigned by the company to one route; some routes can have multiple buses. Each bus has a driver and a conductor, who are assigned to the bus for the day.

4. Bus Company Office

Design the database for the administration and reservation office of a bus company called The All India Bus Company. Each passenger can book a seat on a given portion of the routes serviced by each bus; routes have a starting place, an ending place and several intermediate places. Passengers can specify whether they want to be in the smoking or non-smoking section. Some passengers can get on the bus even if they do not have a reservation, when some seats are left vacant. With each reservation, the last name, initials and telephone number of the passenger are stored. Sometimes, trips are not made because of bad weather conditions; in this case, passengers holding reservations are notified. At the end of the trip, the driver's assistant reports to the company the total number of tickets purchased on the bus by passengers and also report these figures to the administrative office of the branch at the route's destination.

2. All names used in these project descriptions are fictitious

LAB EXERCISES

1. List the properties of the relational model. For each property describe the consequences if a database system did not follow that property.
2. Map the E-R diagram (Fig. 3.23) given on the next page to relational model tables. List any problems you discover with the model and explain how to fix them.
3. Would some additional syntactical or semantic integrity help in maintaining database integrity? Try to think of situations when a database's integrity is violated and suggest integrity constraints that will prevent that violation.
4. Discuss why the relational databases became so popular in the 1980s and DBMS based on network and hierarchical models disappeared.

BIBLIOGRAPHY

For Students

A number of books listed below are easy to read for students. A fascinating account of the history of IBM's System R is available in the paper by Chamberlain et al.

For Instructors

Codd's papers and his out-of-print book (which may be downloaded from the Web) are the authoritative documents on the relational model. C. J. Date's books are also a good source of information. The 1982 book by Ullman is a somewhat advanced treatment of database systems.

Chamberlain, D., et al., "A History and Evaluation of System R", *Communications of the ACM*, Vol. 24, No. 10, 1981, pp 632-646.

Codd, E. F., "A relational model of data for large shared data banks", *Communications of the ACM*, Vol. 13, No. 6, 1970, pp 377-387.

Codd, E. F., "Relational Database: A practical foundation for productivity", *Communications of the ACM*, Vol. 25, No. 2, 1982, pp 109-117.

Codd, E. F., "Extending the relational model to capture more meaning", *ACM Transactions on Database Systems*, Vol. 4, No. 4, 1979, pp 397-434.

Codd, E. F., *The Relational Model for Database Management: Version 2*, Addison-Wesley Publishing Company, Inc, 1990. (This book is out of print now but may be downloaded from http://portal.acm.org/citation.cfm?id=77708.)

Date, C. J., *An Introduction to Database Systems*, Eighth Edition, Addison-Wesley, 2003.

Date, C. J., *Database in Depth: Relational Theory for Practitioners*, O'Reilly Media, 2005.

Maier, D., *The Theory of Relational Databases*, Computer Science Press, 1983. http://web.cecs.pdx.edu/~maier/TheoryBook/TRD.html

Ramakrishnan, R., and J. Gehrke, *Database Management Systems*, Third Edition, McGraw-Hill, 2002.

Silberchatz, A., H. Korth, and S. Sudarshan, *Database Systems Concepts*, Fifth Edition, McGraw-Hill, 2005.

Drug
DrugID
DrugName
Company
used in
1
1
Supervisors
SupName
Sup ID
Address
DOB
Control
1
m
Project
ProjName
ProjID
Stage
m
Fund
n
Grant
GrantID
Origin
Amount
TotalExpense
Credit
1
used In
m
Equipment
EquipName
EquipID
Supplier
Quantity
CostPerItem
TotalCost
Employee
DOB
Address
EmpID
EMPName
m
Work On
1
1
AssocWith
m
DrugSubject
Exercise
Diet
StressRelief
Disease
DOB
PrimaryEffect
SecondEffect
Sex
Result
SubName
SubID
Address

Figure 3.23

Garcia-Molina, H., J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, Prentice Hall, 2002.

Kifer, M., Bernstein A., and P. M. Lewis, *Database Systems: An Application-Oriented Approach*, Second Edition, Addison-Wesley, 2006.

Ullman, J. D., *Principles of Database Systems*, Computer Science Press, 1982. (May be downloaded from http://www.filestube.com/d/database+systems+ullman)

CHAPTER 4

Relational Algebra and Relational Calculus

OBJECTIVES

- Describe how information is retrieved and modified in the relational model.
- Describe relational algebra.
- Present examples of using operators used in relational algebra.
- Describe relational calculus.
- Explain tuple relational calculus (TRC) and present examples.
- Explain domain relational calculus (DRC) and present examples.
- Present foundations for SQL.
- Discuss relative advantages and disadvantages of relational algebra and relational calculus.

KEYWORDS

Relation, relational algebra, relational calculus, selection, restriction, projection, cross product, join, natural join, equi-join, division, union, intersection, difference, procedural language, declarative language, well-formed formulas, queries, tuple relational calculus, TRC, domain relational calculus, DRC, universal quantifier, existential quantifier.

I measure the progress of a community by the degree of progress which women have achieved.

B. R. Ambedkar (Author of the Indian Constitution, 1891–1956)

4.1 INTRODUCTION

Codd in 1970 presented the relational model and with it two formal languages, called relational algebra and relational calculus. Both data manipulation languages provide facility to retrieve information from a relational database. Relational algebra is a procedural language that uses certain primitive operators that take tables as inputs and produce tables as outputs. The language may be considered operational, that is, it provides a step-by-step procedure for computing the result. Each relational algebra operator is either unary or binary, that is, it performs data manipulation on one or two tables. Relational algebra is not similar to programming languages like C or C++, that is, it does not include a looping construct.

The basic relational algebra commands were introduced in the 1970 paper by Codd. These are as follows:

- SELECT
- PROJECT
- UNION
- PRODUCT
- JOIN
- DIVIDE

The relational calculus as opposed to the relational algebra is a declarative language. A statement in relational calculus essentially specifies which tuples (or their columns) are to be retrieved. Therefore relational calculus does not provide a step-by-step procedure. We consider two slightly different forms of relational calculus. These are called *Domain Relational Calculus* (DRC) and *Tuple Relational Calculus* (TRC).

This chapter is organized as follows. The next section, Section 4.2 deals with relational algebra. All relational algebra operators are explained using a number of examples. Section 4.3 deals with relational calculus. The domain relational calculus and the tuple relational calculus are explained along with a number of example queries.

4.2 RELATIONAL ALGEBRA

Database queries in relational algebra are based on specifying operations on tables. The relational algebra expressions are similar to algebraic expressions (for example, $2*a + b*c$) that we are familiar with except that we are now dealing with tables and not numbers or simple variables.

We will define a number of operations that can be applied to tables of all sizes for selecting data from a relational database. The operations include the usual set operations like union and intersection as well as operations for selecting a number of rows or one or more columns from a table.

The result of applying one or more relational operations to a set of tables (possibly one) is always a new table.

We first discuss three relational algebra operations that are sufficient for formulating a large number of useful queries. These operations are *selection*, *projection* and *join.* We will also discuss *Cartesian product* since it is needed in defining the join. Further relational operators are discussed later in the section.

4.2.1 Unary Relational Operator— Selection (also called Restriction)

Unary relational algebra operators operate on a single specified table and display results in table form all or some rows or column of the table.

The operation of selecting certain rows from a table is called *selection* (or *restriction*). Usually one wants to select rows that meet a specified condition, for example, players from India. Selection may then be used. It is a unary operator since it operates only on a single table.

We illustrate this simple concept by showing a table in Fig. 4.1 which has five columns labelled *A*, *B*, *C*, *D* and *E* and five rows. A selection condition (for example, attribute *C* not being equal to c_2) results in selecting three rows of the table.

More formally, selection of a table *R* is a subset table *S* that includes all those rows in *R* that meet a condition specified by the user. The number of columns (degree) in *S* is the same as that of *R*. The number of rows (cardinality) of *S* is equal to the number of rows in *R* that meet the specified condition.

A	*B*	*C*	*D*	*E*
a_1	b_1	c_2	d_1	e_1
a_2	b_2	c_1	d_1	e_1
a_3	b_1	c_2	d_2	e_2
a_4	b_1	c_1	d_3	e_3
a_5	b_1	c_3	d_1	e_4

Figure 4.1 Selection—Selecting three rows in a table

The result of applying selection to a table *R*, such that the column *A* (which may be one or more columns) has value *a* will be denoted by *R*[*A*=*a*]. Similarly, *R*[*A*>*a*] is a selection on *R* such that column *A* has values greater than *a*. A more mathematical way of writing selection is to use the notation $\sigma_{A=a}[R]$ to denote *R*[*A*=*a*][1]. More generally $\sigma_p[R]$ is used to denote a restriction on table *R* which selects rows that satisfy the condition *p*.

Let us now consider two examples of the use of the operator selection. Suppose we wish to find all the Indian players that are in the database, we do so by selecting rows in table *Player* where the value of the attribute *Country* is India. We write the selection as in Fig. 4.2.

$$Player\ [country = \text{'India'}]\ or\ \sigma_{country = \text{'India'}}[Player]$$

Figure 4.2 Using SELECTION in relational algebra

1. The Greek symbol is easy to remember since the operator selection and the Greek letter sigma (σ) both start with the letter *s*. We will from time to time use Greek symbols since they are often used in advanced database textbooks but this notation may be skipped if the reader feels uncomfortable with it.

The result of applying the above selection operation to Table 3.2 *Player* is presented in Table 4.1.

Table 4.1 Result of a selection operator

PlayerID	*LName*	*FName*	*Country*	*YBorn*	*BPlace*	*FTest*
89001	Tendulkar	Sachin	India	1973	Mumbai	1989
96001	Dravid	Rahul	India	1973	Indore	1996
90002	Kumble	Anil	India	1970	Bangalore	1990
25001	Dhoni	M. S.	India	1981	Ranchi	2005
23001	Singh	Yuvraj	India	1981	Chandigarh	2003
96003	Ganguly	Saurav	India	1972	Calcutta	1996
21001	Sehwag	Virender	India	1978	Delhi	2001
98002	Singh	Harbhajan	India	1980	Jalandhar	1998
27001	Kumar	Praveen	India	1986	Meerut	NA
27002	Sharma	Ishant	India	1988	Delhi	2007

Similarly, if we wish to find out what matches have been played in Australia, we select those rows from table *Match* that have column *Team1* value equal to 'Australia' assuming that *Team1* is the home team and these matches are not being played at a neutral country. We could write it in relational algebra as in Fig. 4.3.

$$Match\ [Team1 = \text{'}Australia\text{'}]\ or\ \sigma_{\text{Teaml} = \text{'Australia'}}[Match]$$

Figure 4.3 Using the selection operation in relational algebra

The result of applying the selection operation in Fig. 4.3 to Table 3.1 *Match* is presented in Table 4.2 which has only two rows in it.

Table 4.2 Result of applying a selection operator to *Match*

MatchID	*Team1*	*Team2*	*Ground*	*Date*	*Winner*
2688	Australia	India	Sydney	2/3/2008	Team2
2689	Australia	India	Brisbane	4/3/2008	Team2

4.2.2 Another Unary Relational Operator—Projection

Projection is the relational algebra operation of selecting certain columns from a table *R* to form a new table *S*. For example, one may only be interested in the list of names from a table that has a number of other attributes. Projection may then be used. Like selection, projection is a unary operator that is applied to only one table.

We illustrate the concept of projection by showing the table given in Fig. 4.4 which has four columns labeled *B*, *C*, *D* and *E* and five rows. A projection of the table on attributes *C* and *E* results in selecting the two

highlighted columns of the table. Note that the first and second rows in the table with two columns are duplicates.

B	*C*	*D*	*E*
b_1	c_1	d_1	e_1
b_2	c_1	d_1	e_1
b_1	c_2	d_2	e_2
b_1	c_1	d_3	e_3
b_1	c_3	d_1	e_4

Figure 4.4 Projection—Projecting two columns in a table

R[*A*] is a projection of *R* on *A*, *A* being a list of columns, say three columns *a*, *b* and *c* (the columns do not need to be distinct). Although the table *R* has no duplicate rows (since no table is permitted to have duplicate rows), a projection of *R* might have duplicate rows as illustrated by the projection in Fig. 4.4. The duplicate rows, if any, are removed from the result of a projection. The degree of *S* =*R*[*A*] is equal to the number of columns in *A*. The number of rows in *S* is equal to the number of rows in *R* if no duplicates are present in *R*[*A*]. If duplicates are present then the number of rows in *S* is equal to the number of rows in *R* minus the number of duplicates removed from the result. Removal of duplicates from large tables can be computationally expensive.

A projection of *R* on columns *a, b, c* is denoted by *R*[*a,b,c*]. Some authors use $\pi_{a,b,c}[R]$ to denote the projection[2]. More generally, $\pi_s(R)$ is a projection of table *R* where *s* is a list of columns, possibly only one column.

Let us now consider a simple example. If we wish to obtain a list of names of players, we may apply the projection operator to the table *Player* on attributes *FName* and *LName*. We write this projection as in Fig. 4.5.

Player [Fname, LName] or $\pi_{\text{Fname, LName}}$ [*Player*]

Figure 4.5 Using the projection operator in relational algebra

The result of applying the projection operator as given in Fig. 4.5 to Table 3.2 *Player* would lead to the result presented in Table 4.3.

Table 4.3 The result of applying the projection in Fig. 4.5 to Table 3.2

FName	*LName*
Sachin	Tendulkar
Brian	Lara
Ricky	Ponting
Rahul	Dravid
Herschelle	Gibbs
Shane	Warne
Shaun	Pollock
Michael	Vaughan
Inzamam	Ul-Huq

Contd.

2. Once again, the Greek symbol for projection is easy to remember since the operator projection and the Greek letter pi (π) both start with the letter *p*.

Table 4.3 Contd.

FName	*LName*
Stephen	Fleming
Heath	Streak
Anil	Kumble
Gary	Kirsten
Jacques	Kallis
Chaminda	Vaas
Muthiah	Muralitharan
Daniel	Vettori
MS	Dhoni
Yuvraj	Singh
Saurav	Ganguly
Adam	Gilchrist
Andrew	Symonds
Brett	Lee
Sanath	Jayasuriya
Virender	Sehwag
Shahid	Afridi
Harbhajan	Singh
Praveen	Kumar
Ishant	Sharma

Note that in the above query we have changed the ordering of the attributes to print the names of the players as first name followed by the last name. Similarly, a list of player, last names and their player ID could be obtained by applying the projection given in Fig. 4.6 to the table *Player*, Table 3.2.

$$Player\ [PlayerID,\ LName]\ or\ \pi_{\text{PlayerID, LName}}\ [Player]$$

Figure 4.6 Another example using the projection operator

The result of applying the projection operation given in Fig. 4.6 to Table 3.2 is presented in Table 4.4. To save space we only show the first 18 players, not all the 27 players.

To find the player IDs of all the players that have batted in either match in the database, we apply the projection given in Fig. 4.7 to the Table *Batting*, Table 3.3.

$$Batting[PID]\ or\ \pi_{\text{PID}}\ [Batting]$$

Figure 4.7 Projection applied to the *Batting* table

Table 4.4 The result of applying the projection in Fig. 4.6 to Table 3.2

PlayerID	*LName*
89001	Tendulkar
90001	Lara
95001	Ponting
96001	Dravid
96002	Gibbs
92001	Warne
95002	Pollock
99003	Vaughan
92003	Ul-Huq
94004	Fleming
93002	Streak
90002	Kumble
93003	Kirsten
91003	Kallis
94002	Vaas
92002	Muralitharan
97004	Vettori
More…	More…

The result of applying the projection given in Fig. 4.7 to Table 3.3 *Batting* is presented in Table 4.5.

We obtain Table 4.9 once the duplicate rows from the resulting table are removed. Also, it should be noted that appropriate names of the columns and tables must be used in the relational algebra expressions. The player ID is given the name *PID* in the table *Batting* to save space to fit the table on a page. Also, note that this projection cannot give us a list of player names that have batted since the table *Batting* does not include player names. We will later discuss how to obtain player names of those who batted using information from two tables, Tables 3.2 and 3.3.

It should be noted that the ordering of rows in Tables 4.7, 4.8 and 4.9 is not defined and should not be expected to be the same as displayed in Tables 3.1, 3.2 and 3.3.

Table 4.5 The result of applying the projection given in Fig. 4.7 to Table 3.3

PID
91001
94002
92002
89001
23001
25001
99002
95001
24001
99001
27001

The two unary operators, selection and projection, may be used together to first select some rows of a table and then select only some of the columns of the selected rows. We will consider such examples later in the chapter.

4.2.3 Binary Relational Operation—Cartesian Product

The *Cartesian product* (also known as the *cross product*) of any two tables R and S (of degree m and n respectively) is a table $R \times S$ of degree $m + n$. This product table has all the columns that are present in both the tables R and S and the rows in $R \times S$ are all possible combinations of each row from R combined with each row from S. The number of rows in $R \times S$ therefore is pq if p is the number of rows in S and q in R. As noted earlier, the number of columns of $R \times S$ is $m + n$ if m and n are the number of columns in R and S respectively.

Consider the Cartesian product of the tables *Player* and *Batting*. We first reproduce the schema for the two tables below:

Player(*PlayerID*, *LName*, *FName*, *Country*, *YBorn*, *BPlace*, *FTest*)
Batting(*MatchID*, *PID*, *Order*, *HOut*, *FOW*, *NRuns*, *Mts*, *NBalls*, *Fours*, *Sixes*).

The product will have the following attributes:

Product(*PlayerID*, *LName*, *FName*, *Country*, *YBorn*, *BPlace*, *FTest*, *MatchID*, *PID*, *Order*, *HOut*, *FOW*, *NRuns*, *Mts*, *NBalls*, *Fours*, *Sixes*)

We cannot show what this table looks like since the number of columns is too large to fit on a page. The number of rows is also quite large. How many rows does the product have? Therefore, we will use simpler relations that have only a few columns and only a few rows as shown in Tables 4.6, 4.7 and 4.8. These tables have been given the names *Player2*, *Batting2* and *Bowling2* to differentiate them from *Player*, *Batting* and *Bowling* in Tables 3.3, 3.4 and 3.5.

Table 4.6 The relation *Player2*

PlayerID	*LName*	*FName*
89001	Tendulkar	Sachin
24001	Symonds	Andrew
99001	Lee	Brett
25001	Dhoni	MS

Table 4.7 The relation *Batting2*

MID	*PID*	*Order*	*NRuns*
2689	89001	2	91
2689	99001	8	7
2689	24001	5	42
2755	25001	5	71

Table 4.8 The relation *Bowling2*

MatchID	*PID*	*NWickets*
2689	99001	1
2755	91001	0
2689	24001	1

Table 4.9 shows the Cartesian product of Tables 4.6 and 4.7.

Table 4.9 Cartesian product of tables *Player2* and *Batting2*

PlayerID	*LName*	*FName*	*MID*	*PID*	*Order*	*NRuns*
89001	**Tendulkar**	**Sachin**	**2689**	**89001**	**2**	**91**
24001	Symonds	Andrew	2689	89001	2	91
99001	Lee	Brett	2689	89001	2	91
25001	Dhoni	MS	2689	89001	2	91
89001	Tendulkar	Sachin	2689	99001	8	7
24001	Symonds	Andrew	2689	99001	8	7
99001	**Lee**	**Brett**	**2689**	**99001**	**8**	**7**
25001	Dhoni	MS	2689	99001	8	7
89001	Tendulkar	Sachin	2689	24001	5	42
24001	**Symonds**	**Andrew**	**2689**	**24001**	**5**	**42**
99001	Lee	Brett	2689	24001	5	42
25001	Dhoni	MS	2689	24001	5	42
89001	Tendulkar	Sachin	2755	25001	5	71
24001	Symonds	Andrew	2755	25001	5	71
99001	Lee	Brett	2755	25001	5	71
25001	**Dhoni**	**MS**	**2755**	**25001**	**5**	**71**

The number of columns of the Cartesian product in Table 4.13 is 7 (=3 + 4) and the number of rows is 16 (4 × 4). The product may not appear to be a terribly useful operator to the reader since it combines many rows that have no association with each other, except the four rows highlighted. We will now show that a Cartesian product followed by a selection operation that selects the highlighted rows is a very important operator called the *join*. This is discussed below.

4.2.4 Binary Relational Operator–Join

The join operator may be considered as the most powerful of the relational algebra operations. The join of two related tables R and S ($R \bowtie S$) is a restriction of their Cartesian product $R \times S$ such that a specified condition is met. The result is a single table. The join is normally defined on a column a of table R and a column b of table S, such that the attributes are from the same domain and are therefore related. The specified condition (called the *join condition* or *join predicate*) is a condition on columns $R.a$ and $S.b$[3].

The most commonly needed join is the join in which the specified condition is equality of the two columns ($R.a = S.b$) in $R \times S$. This join is called an *equi-join*. The rows highlighted in Table 4.9 are the rows that

3. The terminology $R.x$ is used to denote column x of table R.

form the equi-join. Since the equi-join by definition has two equal columns one of these may be removed by applying the projection operator. The resulting table is called the *natural join*.

Equi-Join

As noted earlier, the number of columns in the equi-join is $m+n$ if the degrees of R and S are m and n respectively. The degree of the natural join is $m + n - 1$ since one column has been removed from the equi-join. The number of rows in the result table of the join depends on the number of rows that satisfy the join condition.

To illustrate equi-join and natural join, we consider the tables *Player2* and *Batting2* (Tables 4.10 and 4.11). There is an implicit association between the two tables since the table *Player2* provides personal information about players while *Batting2* provides information about their batting. One may be wondering why the two tables are separate. The reason becomes clear if we consider what would happen when the two tables are combined. The personal information of a player is now repeated for each *Batting* (and *Bowling*) row for the player. This is not desirable not only because it wastes storage. This is discussed in more detail in Chapter 6.

The Cartesian product of *Player2* and *Batting2* has two columns that are from the domain *PlayerID*. If the column names in the two tables were the same (they are not in this case), we would need to give one of them a different name in the Cartesian product since each column name in a table must be unique.

If we look at Table 4.9, we find that the Cartesian product table has many rows that are unlikely to be of any use. The relationship between the two Tables 4.6 and 4.7 is based on the two columns *PlayerID* and *PID* and therefore all the rows in the Cartesian product table that do not have the same values of these two columns are of little use.

If we apply a selection to the Table 4.9 such that the values of the two attributes *PlayerID* and *PID* (the highlighted columns) are the same, we obtain Table 4.10.

Table 4.10 Equi-join of tables *Player2* and *Batting2*

PlayerID	*LName*	*FName*	*MID*	***PID***	*Order*	*NRuns*
89001	Tendulkar	Sachin	2689	**89001**	2	91
99001	Lee	Brett	2689	**99001**	8	7
24001	Symonds	Andrew	2689	**24001**	5	42
25001	Dhoni	MS	2755	**25001**	5	71

Note that players will not appear in the above table if they have not batted in a match in our database. Also a player appearing in the *Batting2* table and not in the *Player2* table (which should not normally happen) will also not appear in the equi-join table.

Table 4.10 is an *equi-join* since it has been obtained from the Cartesian product by applying the restriction that the values of the two columns that are from the same domain be equal. These are the columns on which the tables are said to be joined. Note that a player may appear in Table 4.6 *Player2* but may not appear in the join Table 4.10 if he did not have an entry in Table 4.7 *Batting2*.

Natural Join

If we now apply a projection to Table 4.10 so that one of the two identical columns is removed, the resulting table is the *natural join* of the Table 4.6 and Table 4.7. The natural join is shown in Table 4.11.

Table 4.11 Natural join of tables *Player2* and *Batting2*

PlayerID	*LName*	*FName*	*MID*	*Order*	*NRuns*
89001	Tendulkar	Sachin	2689	2	91
99001	Lee	Brett	2689	8	7
24001	Symonds	Andrew	2689	5	42
25001	Dhoni	MS	2755	5	71

Table 4.11 provides the names and the runs scored in each match by the players in Table 4.6 and 4.7.

One might be wondering if a natural join is possible between two tables when the tables have more than one column in common. The formal definition of natural join does in fact include that possibility. The natural join may be defined as follows.

Definition—Natural Join

The natural join of two tables R and S (written as $R \bowtie S$) is obtained by applying a selection and a projection to the Cartesian product $R \times S$ as follows:

1. *For each column 'a' that is common to both tables R and S, we select rows that satisfy the condition R.a = S.a.*
2. *For each column 'a' that is common to both tables R and S, we project out the column S.a. Therefore if there are 'm' columns common to both tables, 'm' duplicate columns are removed from the Cartesian product.*

4.2.5 Join on More Than One Column

To illustrate the natural join of tables that have more than one common column, we consider the tables *Batting2* and *Bowling2*. These two tables have two columns in common, namely the match ID and the player ID. What would be the meaning of joining *Batting* and *Bowling* requiring that both the columns be equal?

We first present the Cartesian product of the two relations (Tables 4.7 and 4.8) in Table 4.12. Note that we have to use the column name *BPID* since duplicate column names are not allowed.

To obtain the natural join we first find the equi-join by deleting all the rows in which the match IDs and player IDs do not match. The highlighted rows are the only rows where both columns match and we obtain Table 4.13 by applying a selection to obtain the highlighted two rows. All other rows in the table are deleted as a result of the equi-join condition.

Table 4.12 Cartesian product of *Batting2* and *Bowling2*

MID	*PID*	*Order*	*NRuns*	*MatchID*	*BPID*	*NWickets*
2689	89001	2	91	2689	99001	1
2689	**99001**	**8**	**7**	**2689**	**99001**	**1**
2689	24001	5	42	2689	99001	1
2755	25001	5	71	2689	99001	1
2689	89001	2	91	2755	91001	0
2689	99001	8	7	2755	91001	0
2689	24001	5	42	2755	91001	0
2755	25001	5	71	2755	91001	0
2689	89001	2	91	2689	24001	1
2689	99001	8	7	2689	24001	1
2689	**24001**	**5**	**42**	**2689**	**24001**	**1**
2755	25001	5	71	2689	24001	1

Table 4.13 Equi-join of the tables *Bowling2* and *Batting2* on two common attributes

MID	***PID***	*Order*	*NRuns*	***MatchID***	***BPID***	*NWickets*
2689	**99001**	8	7	**2689**	**99001**	1
2689	**24001**	5	42	**2689**	**24001**	1

To obtain the natural join, the two duplicate columns that are highlighted in Table 4.13 are removed from the table and we obtain Table 4.14 as the result:

Table 4.14 Natural join of the tables *Bowling2* and *Batting2* on two common attributes

MID	*PID*	*Order*	*NRuns*	*NWickets*
2689	99001	8	7	1
2689	24001	5	42	1

So what does Table 4.14 tell us?

Essentially this table provides us with information for those players that have both batted and bowled in a particular ODI match. The table provides us with their batting performance as well as their bowling performance. If a player only bowled in a match and did not bat in that match or only batted but did not bowl then that player would not appear in this table.

Let us summarize. We obtained the natural join by taking the Cartesian product and then removing those rows that did not have matching values of the common columns resulting in an equi-join followed by removal of the duplicate attributes. The result is called the natural join.

We repeat that an equi-join or a natural join of two matching tables is possible only if both tables have one or more columns that are from the same domains. It would make no sense to have an equi-join on number of wickets in *Bowling2* with batting order in *Batting2*, even if both columns are integers.

Before concluding this discussion of the operation join, we note that when a join is needed to process a query, the procedure used is much more efficient than that which we have used to illustrate the join above in tables 4.10 to 4.14. A simple and more efficient join algorithm involves the following steps:

1. Take the first row from the first table.
2. Search the second table for a row that matches the value of the column on which the tables are being joined in the first table. When a match is found, the two rows are put together to form a row of the join table.
3. The search of the second table is continued for further matches till the second table is exhausted.
4. Now, the next row of the first table is selected and steps 2 and 3 are carried out. This process is continued until the first table is exhausted.

This algorithm is called the *nested scan* method.

In the above algorithm, we have assumed that we are forming an equi-join. A number of other join algorithms are available to process equi-joins more efficiently. These algorithms will be discussed in Chapter 7.

As noted earlier, a major criticism of the relational model for several years was that systems based on the model would always be inefficient because of the high cost of performing joins. In recent years it has been shown that joins can be computed efficiently, particularly so when indexes are available. This has resulted in a number of commercial relational DBMS products that have been shown to have very good performance.

In addition to the operators that we have discussed, there are a number of other relational operators. These are discussed now.

4.2.6 Theta Join

A join of R and S with join condition $a = b$ or $a < b$ is usually written as in Fig. 4.8.

$R[a = b]S$ or $R[a < b]S$

Figure 4.8 Specifying the join condition

Let *theta* be one of the operators $\neq$, $<$, $>$, $\geq$, $\leq$. We may now define a more general concept of *theta-join* of table R (on attribute a) and table S (on attribute b). The theta-join is a restriction on the Cartesian product of R and S such that the condition $a\ \theta\ b$ is satisfied where θ is one of the above operations. We do not consider theta join any further.

4.2.7 Outer Join

The join operations presented so far only deal with tuples from the two tables that match the join condition. The remaining tuples from both tables are eliminated in computing the join although these tuples can often have useful information. For example, if we are joining the *Player* table with the *Batting* table using *PID* equality as the joining condition then a player that has never batted will not be included in the join. This might be useful information in some situations. For instance, if a student table is joined with an enrolment table,

we may be interested in finding out the names of students who have not enrolled. In joining enrolment with courses, we may be interested in finding names of courses that have no enrolments.

There are a number of variants of the outer join. As noted above, the join of the *Player* table with the *Batting* table using *PlayerID* equality as the joining condition does not include the players that have never batted. A join including this information is called the *left outer join*. If the join also included batting information on players that batted but were not in the *Player* table (normally there should be no such rows in *Batting*) then the join is called a full outer join. If the join did not include information about players that did not bat but included information about players that batted but were not in the *Player* table then the join is called a *right outer join.*

We leave it as an exercise to find the left outer join of *Player2* and *Batting2*. Outer join is discussed again in Chapter 5 with its implementation in SQL.

4.2.8 Set Operations

Set Union

The union operator is the usual set union of two tables R and S which are union-compatible. Two tables are called union-compatible if they are of the same degree m and are written as

$$R(a_1, a_2,.. a_m)$$
$$S(b_2, b_2, \ldots b_m)$$

For each i (i = 1, 2, ... m), a_i and b_i must have compatible domains (e.g., either both are integers or both are strings of the same length). The number of columns in the union of R and S, $R \cup S$, is the same as that of R and S and the number of rows is $a + b$ if a and b are the number of rows in R and S respectively and there are no duplicate rows in R and S. $R \cup S$ therefore consists of all rows that are either in R or S or common in both R and S. As an example, consider two tables, Table 4.15 and Table 4.16, given below obtained by applying projection to the tables *Batting* (Table 3.3) and *Bowling* (Table 3.4). Table 4.15 *Batting3* gives the columns *MatchID* and *PID* from the table *Batting* while the projection on *Bowling* results in *Bowling3* with columns *MatchID* and *PID* in Table 4.16.

Table 4.15 *Batting3* obtained by projection of the Table 3.3 *Batting*

MatchID	*PID*
2755	23001
2755	25001
2755	91001
2755	94002
2755	92002
2689	89001
2689	23001
2689	25001
2689	99002
2689	95001
2689	24001
2689	99001
2689	27001
2755	27001

If we were interested in finding all the players who have batted or bowled or both, we can take a union of the two tables *Batting3* and *Bowling3* (Tables 4.15 and 4.16) and obtain the result in Table 4.17.

There are many duplicates in Table 4.17 and the duplicate pairs have been highlighted in the table and therefore the table does not satisfy the definition of a relation. We have not removed duplicates from the above union to illustrate that duplicates may occur after union. The rows have been sorted to make it easier

Table 4.16 *Bowling3* obtained by projection of the Table 3.4 *Bowling*

MatchID	*PID*
2689	99001
2689	24001
2689	23001
2755	94002
2755	92002
2755	91001
2755	23001

to identify duplicates which must be removed if the result is to satisfy the properties of a relation. There are 14 duplicate rows and therefore seven of them need to be removed.

A projection of the above table, Table 4.17, on attribute *PID* now provides the list of players that have either batted or bowled or both in the matches in the database. The duplicates have been removed and the list in Table 4.18 has been sorted. Table 4.18 therefore has only 11 rows after removing duplicate rows for players 23001, 24001, 99001, 91001, 92002 and 94002.

4.2.9 Set Intersection

The intersection relational operator is the usual set intersection of two tables *R* and *S* which are union-compatible. The result of the intersection is a table consisting of all the rows that are common to both tables *R* and *S*. The number of columns in the intersection $R \cap S$ is the same as that of *R* and *S* and the number of rows in the intersection is equal to the number of rows common to both *R* and *S*.

As an example, let us again consider the earlier example of Tables 4.15 and 4.16, *Batting3* and *Bowling3*. Suppose now that we wish to find those players that bowled as well as batted in the same matches. One way is to formulate the query would be to find the intersection of the two tables. The intersection is formulated in Fig. 4.9.

Table 4.17 Union of Tables 4.15 and 4.16

MatchID	*PID*
2689	**23001**
2689	**23001**
2689	**24001**
2689	**24001**
2689	25001
2689	27001
2689	89001
2689	95001
2689	**99001**
2689	**99001**
2689	99002
2755	**23001**
2755	**23001**
2755	25001
2755	27001
2755	**91001**
2755	**91001**
2755	**92002**
2755	**92002**
2755	**94002**
2755	**94002**

Table 4.18 Projection of Table 4.17 which was a union of two tables

PID
23001
24001
25001
27001
89001
91001
92002
94002
95001
99001
99002

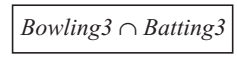

Figure 4.9 Intersection of the tables *Batting3* and *Bowling3*

The result of the intersection is a table with two columns (*MatchID, PlayerID*) for the players that are common to both tables *Bowling3* and *Batting3*, and hence have batted and bowled in the same match. The result of the intersection is given in Table 4.19.

Table 4.19 Result of the intersection operation in Fig. 4.9

MatchID	*PID*
2689	99001
2689	24001
2689	23001
2755	94002
2755	92002
2755	91001
2755	23001

Note that Table 4.19 is the same as Table 4.16 since every player that bowled in that table also batted in Table 4.15.

Furthermore, we should note that we could not take an intersection of the tables *Batting2* and *Bowling2* (Tables 4.7 and 4.8) since the number of columns in those two tables are not the same.

4.2.10 Set Difference

The difference operator when applied to two tables R and S (written as $R - S$) results in a table that consists of all the rows in the first table that are not also in the second table. If the intersection of R and S is null then $R - S = R$, otherwise $R - S = R - R \cap S$, that is, $R - S$ is obtained by removing from R the rows that are also in S. The number of columns in $R - S$ is the same as that of R and S and the number of rows in the intersection is equal to the number of rows of R minus the number of rows in the intersection $R \cap S$.

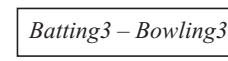

Figure 4.10 Difference of two tables

Table 4.20 Difference of the Tables 4.15 and 4.16, *Batting3* and *Bowling3*

MatchID	*PID*
2689	25001
2689	27001
2689	89001
2689	95001
2689	99002
2755	25001
2755	27001

We again consider the example of the two Tables 4.15 and 4.16. If we want to find the players that batted in a match but did not bowl in it, we could formulate the query by the difference as presented in Fig. 4.10.

The result given in Table 4.20 is obtained when we take the difference of Tables 4.15 and 4.16.

Let us now look at the meaning of the difference *Bowling3—Batting3*. This difference would be a table that consists of all the rows in the first table that are not also in the second table. Therefore it is the list of players that have bawled but not batted in a match. For Tables 4.15 and 4.16, the difference *Bowling3 – Batting3* is null since there are no bowlers in Table 4.16 that have not also batted.

4.2.11 Division[4]

The concept of division is related to the concept of Cartesian product in that $R \times S$ divided by S yields R. Division therefore is essentially an inverse of Cartesian product in the same way as arithmetic division is the inverse of multiplication.

4. Part of this section may be omitted without loss of continuity.

Example

Before discussing the division operator further, we present an example.

Consider a table *R* and a table *S* as given in Tables 4.21 and 4.22.

To simplify our example, we have chosen a very simple table *S* that consists of only two rows as given in Table 4.22.

Table 4.22 The table *S*

PID
99001
27001

The result of dividing relation *R* by relation *S* is given below in Table 4.23.

Table 4.23 gives the match IDs (in this case, only one) in which both players listed in Table 4.22 have played. That is the function of the division operator.

Table 4.21 The table *R*

MatchID	*PID*
2755	23001
2755	25001
2755	91001
2755	94002
2755	92002
2689	89001
2689	23001
2689	25001
2689	99002
2689	95001
2689	24001
2689	99001
2689	27001
2755	27001

We first define division for a simple case when a table *R* has only two columns and the table *S* has only one column. Let the domain of the second column of *R* be the same as of the column of *S*. The division of *R* by *S*, or *R/S* is then a table with only one column that has the same domain as the first column of *R,* such that each row in *R/S* occurs in *R* concatenated with every row of *S*.

Table 4.23 The result of dividing table *R* by table *S*

MatchID
2689

When the tables *R* and *S* are of degree greater than 2, division can be defined similarly. For example, if *R* was $R(a_1, a_2, \ldots a_n, b_1, b_2, \ldots b_n)$ and *S* was $S(b_1, b_2, \ldots b_n)$ then *R/S* would be a table $T(a_1, a_2, \ldots a_n)$ such that *R* had at least all rows that $T \times S$ would have. We may write *R* as having columns *R*[*A, B*] and *S* having column *B* only where *A* and *B* each could consist of more than one column.

Finding those values of attribute *A* that appear with each value of *B* in *S* is not directly possible in relational algebra because relational algebra does not have a direct divide operator. Therefore the division algorithm involves finding values of attribute *A* in *R* that do not appear with all values of *B* in *S*. This is complex but is possible as shown below. *R* divided by *S* is then given by a relation *T* given the relational algebra expression in Fig. 4.11.

$$T = R[A] - ((R[A] \times S) - R)[A]$$

Figure 4.11 Relational algebra expression for division

Let us now explain what the above difficult to understand expression means. Before the explanation, we repeat that *R*[*A,B*]/*S*[*B*] is a table *T*[*A*] such that the relation *R* includes each value of *A* in *T* with every value of *B* in the table *S*. How does the expression in Fig. 4.11 find those values of *A*? We will use a step-by-step description of the expression in Fig. 4.11.

1. We first find a table that has each value of attribute A that exist in R appearing with every value of attribute B that exists in S. This can be done by first finding all values of A in R which is $R[A]$ and then computing $R[A] \times S$, the Cartesian product of $R[A]$ and S. The product has two columns, A and B, same two columns as R does, with each value of column A in R combined with every value of column B in S.
2. We now find the values of attribute A that do not appear with each value of attribute B in S. This can be done by using the product relation found above (the product has each value of column A in R combined with every value of column B in S) and subtract the relation R from it. That means we find $R[A] \times S - R$ which is a table that has only those rows that are in the Cartesian product but not in R. In other words, these are the rows that are not in R because the A value in each of these rows does not appear with each value of B that appears in S.
3. Carry out a projection to find the values of attributes A that do not appear with every value of attributes B in S. This is simply done by taking the projection $(R[A] \times S - R)[A]$ which is a table with only one column, A. It only has those values of column A that are present in R but which do not appear with every value of column B in S.
4. Find the result of the division. To obtain the result we subtract the values of A found above from values of A in R since that gives us the values of A that do appear with every value of B in S. Therefore $R[A] - (R[A] \times S - R)[A]$ are the values of column A that are present in R and each of which appears with every value of B in S. Hence this is the result of dividing R by S.

Let us now again consider the tables R and S given in Tables 4.21 and 4.22 above.

We now illustrate how the formula $T = R[A] - ((R[A] \times S) - R)[A]$ is used in computing the division. $R[A]$ is given in Table 4.24. All the duplicates have been removed. There are only two ODI matches in R.

Table 4.24 The table $R[A]$

MatchID
2755
2689

Now the Cartesian product $R[A] \times S$ is given in Table 4.25. This table shows what R would look like if all the players in S had played in all the matches in R.

Table 4.25 The table $R[A] \times S$

MatchID	*PID*
2755	99001
2689	99001
2755	27001
2689	27001

Now $(R[A] \times S - R)$ is given by Table 4.26. These are rows that are in the product $R[A] \times S$ but not in R. That is, these rows show the matches in which players in S did not play. There is only one such row in this table.

Now we take the projection of Table 4.26 on *MatchID* and subtract it from column A of R. This gives us the result of the division.

The division R/S is given by $R[A] - (R[A] \times S - R)[A]$ which is now given by Table 4.27.

Table 4.26 The table $R[A] \times S - R$

MatchID	*PID*
2755	99001

Table 4.27 The table $R/S = T = R[A] - (R[A] \times S - R)[A]$

MatchID
2689

This is the result of the division. It is a list of all matches that the players in *S* given in Table 4.22 have played together. In this case, for simplicity, Table 4.22 had only two players.

We note that $R/S = T$ does not imply that $S \times T = R$, since *R* could well include other rows. These rows are discarded in the same way as the remainder is discarded in the integer division.

The division of *R* by *S* where the attributes *x* in *R* and *y* in *S* are union-compatible is written as in Fig. 4.12.

$$R\ [x / y]\ S$$

Figure 4.12 Division of *R* by *S* on attributes *x* and *y*

If the division also involves a projection operator as is often necessary, then it may be written as given in Fig. 4.13.

$$R\ [Ax]\ [x / y]\ S[y]$$

Figure 4.13 Division involving projections

This concludes our discussions of relational algebra operators.

The relational algebra described above does not include capabilities like counting and summing. Such capabilities are obviously needed in a practical system. They are included in the commercial language SQL which is discussed in the next chapter.

4.3 RELATIONAL ALGEBRA QUERIES

We now present a number of queries on the cricket database presented in Tables 3.1 to 3.4. The queries range from very simple to quite complex and have been formulated using relational algebra that we have discussed.

4.3.1 Only Projection

- (*A*1) Find the match IDs of matches that do have some bowling.
  This is a simple query that only involves a projection on table *Bowling*. It is formulated in Fig. 4.14.

$$Bowling[MatchID]$$

Figure 4.14 Relational algebra formulation for query *A*6

The result is given in Table 4.28. Note that all duplicates have been removed.

Table 4.28 Result of query *A*1 – Match IDs of matches that do have some bowling

MatchID
2689
2755

4.3.2 Selection combined with Projection

- (*A*2) Find the names of all players who played their first test after 2000.

In relational algebra notation, the query may be written as in Fig. 4.15[5]

$$Player[Fname, LName][FTest > 2000] \text{ or } \pi_{Fname, LName}(\sigma_{FTest>2000}[Player])$$

Figure 4.15 Relational Algebra for query *A*2

Note that the query in Fig. 4.15 involves a projection which projects the result to display the first and last name of the selected players. Before the projection a selection is carried out which selects the players who played their first test after 2000. Clearly the projection cannot be done before the selection because the projection would then have discarded the FTest column.

The result of query in Fig. 4.15 is given in Table 4.29.

Table 4.29 Result of the projection in query *A*2

FName	*LName*
MS	Dhoni
Yuvraj	Singh
Andrew	Symonds
Virender	Sehwag
Ishant	Sharma

- (*A*3) Find the player IDs of all players who batted in the match 2755.

  This query involves a selection and a projection to be applied to the table *Batting*. We carry out the selection first to find all players that have batted in match 2755 and then carry out the projection to get their IDs. The projection could not be carried out before the selection since the *MID* column will then be lost in projection and we could not apply selection.

  The query may be written as given in Fig. 4.16.

$$Batting[MatchID = '2755'][PID]$$

Figure 4.16 Relational algebra formula for query *A*3

The result of relational algebra operation in Fig. 4.16 is given in Table 4.30.

Table 4.30 Result of the selection and projection in query *A*3

PID
23001
25001
91001
94002
92002
27001

4.3.3 Using the Join of Two Tables

- (*A*4) Find the names of all players who batted in match 2755.

  This is a more complex query since it involves information that cannot be accessed without looking at two tables since the table *Batting* has the batting information and the table *Player* has information about players' names. To answer the query, we carry out a natural join of the tables *Player* and *Batting* on the attribute *PlayerID* and then apply selection *MatchID* = 2755. This is followed by a projection on attributes *FName* and *LName.*

  In relational algebra notation, we may write the above query as in Fig. 4.17.

  We are using the symbol $\bowtie$ to indicate the natural join operator. The result of the query posed in Fig. 4.17 is given in Table 4.31.

5. The Greek symbol notation for relational algebra queries are only used for some selected queries.

$$(Player\ [PlayerID = PID]\ Batting)\ [MatchID = \text{'2755'}]\ [Fname, LName]$$
$$or\ \pi_{Fname,\ Lname}\ [\sigma_{MatchID=\text{'2755'}}(Player \bowtie Batting)]$$

Figure 4.17 Relational algebra expression for query *A*4

Table 4.31 Result of query *A*4—Names of all players who batted in match 2755

FName	*LName*
Chaminda	Vaas
Muthiah	Muralitharan
MS	Dhoni
Yuvraj	Singh
Sanath	Jayasuriya
Praveen	Kumar

These are the names of players whose player IDs were in Table 4.30.

4.3.4 Using Joins of Three Tables

- (*A*5) Find the ground names and dates of all matches in which any player with last name Dhoni has batted.

  This query is even more complex since it must involve the three tables *Match*, *Player* and *Batting*. The player name information is in table *Player*, the batting information is in table *Batting* and the ground names are available in table *Match*. One approach to formulating this query is to join *Player* and *Batting* on attribute *PlayerID* and then join the resulting table with table *Match* on the attribute *MatchID*. We then apply a selection that the player name is Dhoni and then take the projection to find names of matches. We assume that the attributes *Ground* and *Date* provide the information required. In relational algebra, the query may be written as in Fig. 4.18.

$$(Player \bowtie Batting) \bowtie Match)\ [LName = \text{'Dhoni'}]\ [Ground, Date]$$

Figure 4.18 Relational algebra formula for query *A*5

The result of query formulated in Fig. 4.18 is given in Table 4.32.

Table 4.32 Result of query *A*5—Ground names and dates of all matches in which Dhoni has batted

Ground	*Date*
Brisbane	4/3/2008
Colombo	27/8/2008

4.3.5 Using Difference

- (*A6*) Find the player IDs of all Indian players who have not batted in any match.
  Since *Batting*[*PID*] is the list of players that have batted in one or more of the matches, the difference between *Player*[*PlayerID*] of Indian players and *Batting*[*PID*] is the list of Indian players that have not batted in any match in the database because the difference consists of all the rows in the first table that are not also in the second table.
  The above query may therefore be formulated as follows in Fig. 4.19.

$$(Player[Country = \text{'India'}])\ [PlayerID] - Batting\ [PID] \text{ or}$$
$$\pi_{PlayerID}[\sigma_{Country=\text{india}}\ Player] - \sigma_{\text{PID}}[Batting)]$$

Figure 4.19 Relational algebra expression for query *A6*

Note that the selection is done first in the query in Fig. 4.19.
We are only considering information in Tables 3.2 and 3.3 which include only part of the information of the two ODI matches. For example, Harbhajan Singh's ID is not in the result table below although he did bat in match 2755.
The result is given in Table 4.33.

Table 4.33 Result of query *A6*—Player IDs of Indian players who have not batted in any match

PlayerID
96001
90001
96003
21001
98002
27002

- (*A7*) Find the player IDs of players who have bowled only in match 2755.
  To find players who have bowled only in match 2755 we must ensure they have not bowled in any other match in our very small database. Therefore, the query involves finding the difference between the set of players who have bowled in match 2755 and those who have bowled in matches other than 2755. Any player who has bowled in 2755 as well as other matches will appear in both sets and will therefore not appear in the difference.

$$Bowling[MatchID = \text{'2755'}]\ [PID] - Bowling[MatchID <> \text{'2755'}]\ [PID]$$

Figure 4.20 Relational algebra expression for query *A7*

The result of query posed in Fig. 4.20 is given in Table 4.34.

Table 4.34 Result of query *A7*—Player IDs of players who have bowled only in match 2755

PID
94002
92002
91001
23001

- (*A8*) Find the match IDs of matches that include all Indian players in our player table that were born after 1985.
  We first find the list of all Indian players born after 1985 by applying a restriction and a projection to the table *Player*. Then we find a list of all player IDs and match IDs for all players that have batted and divide

that list by the list of Indian players born after 1985 in the database. The result of the division will be those matches that had all Indian players batting in them.

$$Batting\ [PID, MatchID] \,/\, (Player[Country = \text{'India'}] \wedge Yborn > 1985]\ [PlayerID])$$
or $\pi_{PID,\ MatchID}[Batting] / \pi_{PlayerID}\ (\sigma_{Country\ =\ \text{'India'}}[Player])$

Figure 4.21 Relational algebra expression for query A8

The result is given in Table 4.35.

Table 4.35 Result of query *A*8—Match IDs of matches that include Indian players born after 1985

MatchID
2689
2755

The above queries would have given the reader a flavour of how relational algebra can be used to answer user queries. We would like to note that the formulations presented above are not the only possible formulations for the queries that we had. A query may often be formulated in more than one way.

A question that often arises at this point is, are there better (that is, more efficient) sequences of operations than those listed above to answer the queries? For example, in query *A*3 above, could we carry out the restriction on the table *Player* before the join? This is clearly an important question but we will not deal with it here. There are usually a number of different sequences that may be used to process a query. Some of these sequences would obviously be more efficient than others. This topic is presented in detail in Chapter 7 on query optimization.

Relational algebra is like the assembler programming language. Just as there are many higher level programming languages like C++, there are other relational query languages as well. We discuss relational calculus next since this was another language proposed by E. F. Codd.

4.4 RELATIONAL CALCULUS[6]

Relational algebra provides a collection of operators and a relational algebra expression explicitly states what operations are to be done first and followed by other operations, and so on. Relational algebra, therefore provides a procedural language that can be used in specifying a sequence of operations that ought to be used in obtaining the result that the user wants. Algebra is thus like a procedural programming language where a program written in the language clearly provides a sequence of instructions that the machine follows to obtain the result. In contrast, relational calculus involves specifying conditions (predicates) that the result must satisfy. Relational calculus, therefore is a nonprocedural language for defining the table of information that the user wants to retrieve. The commercial query language SQL is also a nonprocedural language based on relational calculus.

We will briefly discuss how to define conditions using predicate calculus since it provides the basis for relational calculus. Predicate calculus is a powerful technique for logical reasoning. To introduce it, we need to discuss propositional logic first. The reader need not worry about terms like 'predicate calculus' or 'propositional logic' since they are simple concepts that are explained ahead. The background given in the next few pages can be skipped by readers not interested in the background.

6. The first part of this section may be skipped without any loss of continuity.

4.4.1 Propositions

A proposition is just a statement. It could be 'Tendulkar has batted in match 2755' or 'Shane Warne was born in Melbourne'. These are atomic propositions and may be denoted by symbols like *A* and *B* and their value is either true or false. Compound propositions contain two or more atomic propositions.

A compound proposition with two components may be of the type (*A* AND *B*) or it may be of the type (*A* OR *B*). Such propositions are called *formulas*. Their truth depends on the truth values of their components. When a sequence of propositions is joined by ANDs (e.g., *A* AND *B* AND *C*) we call it a *conjunction* of propositions and the conjunction is true only if all atomic propositions are true. When a sequence of propositions is joined by ORs (e.g., *A* OR *B* OR *C*), we call it a *disjunction* of propositions. The disjunction is true when at least one of the atomic propositions is true and false only when all the propositions are false.

Since we are dealing with propositions that have values true or false, the theorems of Boolean algebra apply. We assume the reader is familiar with, for example, De Morgan's Laws which are given in Fig. 4.22.

NOT (*A* AND *B*) = (NOT *A*) OR (NOT *B*)
NOT (*A* OR *B*) = (NOT *A*) AND (NOT *B*)

Figure 4.22 De Morgan's Laws

Consider the first expression. (*A* AND *B*) is true only if *A* and *B* both are true. It is false if either *A* or *B* is false or both *A* and *B* are false. The opposite of (*A* AND *B*), therefore is false only if (*A* AND *B*) is true which implies that *A* and *B* both are true or both (NOT *A*) as well as (NOT *B*) are false. We can similarly explain the second equation.

4.4.2 The Implies Operator

We now consider formulas that have the form 'if *A* then *B*'. The formula specifies that *B* must be true if *A* is. It does not say anything about the value of *B* if *A* is false. The truth table for 'if *A* then *B*' is

Table 4.36 Truth table for 'if *A* then *B*'

A	*B*	*If A then B*
T	T	T
T	F	F
F	T	T
F	F	T

It can be shown that *if A then B* is equivalent to *(not A) or B*. *If A then B* may be conveniently written as $A \rightarrow B$ (that is, *A* implies *B*). It may be shown that $A \rightarrow B$ is equivalent to (not *B*) $\rightarrow$ (not *A*).

The above notation can be quite useful in logical reasoning but is limited in its power of expression as we shall soon see. We first briefly note how the process of reasoning occurs. Reasoning is based primarily on two rules:

1. If *A* is true and we are given $A \rightarrow B$, then *B* must be true.
2. If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

The first rule is called *modus ponens* and the second rule is called the *chain rule*. Logical reasoning involves proving the truth of a proposition, given a set of propositions that are true. For example, we may be given propositions as follows:

1. If it rains (*A*), it must be cloudy (*B*).
2. If it is cloudy (*B*), I don't play golf (*C*).

3. It is raining (A)
4. Derive – I don't play golf (C)

The above propositions may be written as follows:

- $A \rightarrow B$
- $B \rightarrow C$
- A is true
- B is therefore true given $A \rightarrow B$
- C is therefore true given $B \rightarrow C$

Similarly, we may be given a set of propositions about players and their batting and bowling and we may, for example, wish to prove the truth in 'Dravid has played with Tendulkar'.

Sometimes we may face a situation, such that the proposition that we wish to prove can be shown to be true for all values of atomic propositions in it. Such a proposition is called *tautology*.

As we noted earlier, propositional calculus is useful but limited in power. In the next two sections we present facilities that improve the power of propositional logic. The more powerful logic is called *predicate calculus* or *first order logic*.

4.4.3 Parameterized Propositions

Propositions may be generalized by parameterization. For example, 'Tendulkar played in match 2755' may be written as M2755(Tendulkar) and its value is true if Tendulkar did play in match 2755. We assume M2755 is a predicate meaning 'playing in 2755'. Similarly, M2755(Dravid)) would be false if Dravid did not play in match 2755.

By introducing parameterization we have introduced a more compact notation. Rather than assigning new variables to all propositions regarding players who played in match 2755 we may now use M2755(.). Parameterization gives us a power similar to that provided by a looping facility in a programming language. If a programming language does not have any looping facility, we could still do what we wish but it would require writing similar statements over and over again, which would be very tedious. Parameterization helps us to overcome such tedious repetition.

A parameterized proposition is called a *predicate*.

Once we introduce parameters in propositions, we need a mechanism to specify what values these variables might take. This is done by the *universal* and *existential quantifiers*.

4.4.4 Universal and Existential Quantifiers

The universal quantifier (*for all* or $\forall$) and the existential quantifier (*there exists* or $\exists$) allow use of variables in the predicates enabling very compact representation of propositions. Suppose we want to express the following statement 'If a player is a captain in match 2755 then he is a player in that match', we may write it as in Fig. 4.23.

$$\forall x(\text{C2755}(x) \rightarrow \text{M2755}(x))$$

Figure 4.23 Using the universal quantifier

We assume C2755(x) in Fig. 4.32 is a proposition that x is a captain in match 2755 and M2755(x) states that x has played in match 2755.

Similarly, to express the statement that 'some players in match 2755 are captains', we may write as in Fig. 4.24

$$\exists x(\text{M2755}(x) \text{ and } \text{C2755}(x))$$

Figure 4.24 Using the existential quantifier

The statement in Fig. 4.33 means 'there is at least one person who is a player in match 2755 and is a captain in that match'.

It should be noted that the statement in Fig. 4.25 following is not equivalent to the above statement

The reason is as follows: the statement in Fig. 4.25 means '*there does exist a player for whom M2755 (x)* → *C2755(x) is true*'. This will always be true since one of the players will always be a captain.

$$\exists x(\text{M2755}(x) \rightarrow \text{C2755}(x))$$

Figure 4.25 Using the existential quantifier

This leads us to a rule of thumb in writing predicate calculus statements; a universal quantifier usually goes with an implication while the existential quantifier goes with a conjunction.

We note that expressions containing the universal quantifier may be replaced by equivalent expressions containing existential quantifiers. For example, consider the formula in Fig. 4.26.

$$\forall x(\text{P}(x)) \text{ is equivalent to } \text{Not}(\exists x(\text{Not P}(x))$$

Figure 4.26 Relation between existential and universal quantifiers

Each variable appearing in a formula is either *free* or *bound*. The bound variables are those that have been defined in one of the quantifiers. Others are called free. For example, consider the formula in Fig. 4.27.

$$\forall x(\text{P}(x) \rightarrow \text{R}(x, y))$$

Figure 4.27 Using the universal quantifier

x is a bound variable while y is free. Bound variables are like formal parameters, the variable name being legal only within the scope of the variable and the name of the variable being immaterial. That is, the above formula in Fig. 4.36 may be written as in Fig. 4.28.

$$\forall a(\text{P}(a) \rightarrow \text{R}(a, y))$$

Figure 4.28 Using the universal quantifier

The formula in Fig. 4.28 has the same meaning as in Fig. 4.27. A formula that has no free variables is called a *sentence*.

4.4.5 Relational Calculus

Relational calculus is closely related to predicate calculus since tables in a relational database may be thought of as predicates. For example, given a table

Player(*PlayerID, LName, FName, Country, YBorn, BPlace, FTest*)

the expression *Player*(*a,b,c,d,e,f,g*) may be thought of as a predicate which is true if a row with values *a,b,c,d,e,f,g* exists in the table *Player*.

As noted in the introduction of this chapter, there are two variants of relational calculus; tuple relational calculus (TRC) and domain relational calculus (DRC). These are described now.

The difference between TRC and DRC may be described as follows. TRC, the original relational calculus presented by E.F. Codd, specifies queries in terms of variables that range over rows of the relations in the database while DRC specifies queries in terms of variables that range over columns (or domains) of the relations in the database. Commercial language SQL is closely related to TRC while DRC is related to a graphical language that is used in Microsoft Access. It is called Query by Example (QBE) and was designed at IBM Yorktown Height research centere in the 1970s. QBE is quite different than SQL since it uses a graphical interface that shows tables in which a user can specify queries by providing input which essentially provides an example of the output that user wants. It is easy to translate most SQL queries into TRC and most QBE queries into DRC although TRC and DRC do not include functions.

4.4.6 Tuple Relational Calculus

We now describe the concept of a tuple (or row) variable that is basic to the calculus defined by Codd in 1972. A tuple variable *r* is essentially a row. It is a variable that is said to range over some table *R* if the only values that *r* can take are the rows in table *R*. An indexed row has the form *r.A* where *r* is a tuple variable and *A* is an attribute. To retrieve rows that meet a given condition, the format of queries in calculus is $(r \mid P(r))$, that is, retrieve those values of tuple variable *r* that satisfy the predicate *P(r)* (or for which *P(r)* is true). *r* of course could be a list *(x,y,z)* and is called the *target list*, *P* is sometimes called the *qualification expression*. For example, a query in tuple relational calculus might be written as in Fig. 4.29.

The first expression in Fig. 4.29 specifies that we are interested in column *LName* of rows *s* such that the rows are in table *Player*. We therefore get a list of last names of all players that are in the table. The second expression specifies that we are interested in column *PlayerID* of rows *s* such that *s* is in table *Player* and the value of the column *Country* is 'Australia'. Essentially these expressions are the same queries as we posed in relational algebra but the primary difference is that this formulation does not specify the order of operations as the queries in relational algebra.

$\{s.Lname \mid s \in Player\}$
or $\{s.PlayerID\ s \in Player \wedge s.Country = \text{'Australia'}\}$

Figure 4.29 A query in tuple relational calculus

As above, we will use small case letters *r, x, y,* etc., to denote row variables and *r.LName, x.PID,* etc., to identify a column of the row variable. As noted above, the format of queries in relational calculus is of the form $\{(x,y,z) \mid P(x,y,z)\}$ where *x,y,z* are tuple variables or indexed tuple variables such that the condition *P* is satisfied.

We may sometimes omit the brackets, { }, since this does not lead to any confusion. To illustrate relational calculus, we again consider the database in Fig. 4.30.

Match (MatchID, Tram1, Tram2, Ground, Date, Winner)
Player (PlayerID, LName, FName Country, Yborn,Bplace,FTest)
Batting (MatchID, PID, Order, HOut, FOW, NRuns Mts, NBalls, Fours, Sixes)
Bowling (MatchID, PID, NOvers, Maidens, NRuns, NWickets)

Figure 4.30 Schema for the ODI cricket database

We now present a number of queries and formulate them in relational calculus.

- (*C*1) Find the names and IDs of all players.
  The relational calculus formulation of query *C*1 is given in Fig. 4.31.

  {s.Fname, s.Lname, s.PlayerID | s∈ Player}

  Figure 4.31 Tuple relational calculus expression for query *C*1

  The above calculus expression specifies that we are interested in the values of the columns *FName, LName* and *PlayerID* of all rows *s,* such that *s* is in table *Player* (that is, all players). The result is presented in Table 4.37.

Table 4.37 Names and IDs of all players

FName	*LName*	*PlayerID*
Sachin	Tendulkar	89001
Brian	Lara	90001
Ricky	Ponting	95001
Rahul	Dravid	96001
Herschelle	Gibbs	96002
Shane	Warne	92001
Shaun	Pollock	95002
Michael	Vaughan	99003
Inzamam	Ul-Huq	92003
Stephen	Fleming	94004
Heath	Streak	93002
Anil	Kumble	90002
Gary	Kirsten	93003
Jacques	Kallis	95003
Chaminda	Vaas	94002
Muthiah	Muralitharan	92002
Daniel	Vettori	97004
MS	Dhoni	25001
Yuvraj	Singh	23001
Saurav	Ganguly	96003
Adam	Gilchrist	99002
Andrew	Symonds	24001
Brett	Lee	99001
Sanath	Jayasuriya	91001
Virender	Sehwag	21001
Shahid	Afridi	98001
Harbhajan	Singh	98002
Praveen	Kumar	27001
Ishant	Sharma	27002

- (*C*2) Find IDs of all players from India.
  The tuple relational calculus formulation of query *C*2 is given in Fig. 4.32.

$$\{e.PlayerID \mid e \in Player \wedge (e.Country = \text{'India'})\}$$

Figure 4.32 Tuple relational calculus expression for query *C*2

The above tuple relational calculus expression specifies that we wish to retrieve values of the column *PlayerID* of all rows *e,* such that *e* is a member of the table *Player* and the value of the column *Country* of *e* is India.
The result of the query in Fig. 4.32 is presented in Table 4.38.

Table 4.38 Result of query *C*2 posed in Fig. 4.32

PlayerID
89001
96001
90002
25001
23001
96003
21001
98002
27001
27002

- (*C*3) Find the names of all players from India who have batted in a match.
  The tuple relational calculus formulation of query *C*3 is given in Fig. 4.33.

$$\{s.FName, s.LName \mid s \in Player \wedge \exists y(y \in Batting \wedge s.PlayerID = y.PID \wedge s.Country = \text{'India'})\}$$

Figure 4.33 Tuple relational calculus formulation for query *C*3

The above expression specifies the columns that we wish to retrieve. The columns that we are interested in are *FName* and *LName* and we want all rows *s,* such that *s* is a member of table *Player* and there exists a row *y* that is in table *Batting* such that the value of the column named *PlayerID* in *s* which is in *Player* and *PID* in *y* which is in *Batting* is the same and the value of the column *Country* of row *s* is India. In other words, *s* is retrieved only if it names a player from India and there is an entry for this player in *Batting.*
The result is presented in Table 4.39.

Table 4.39 Result of query *C*3 posed in Fig. 4.33

FName	*LName*
Sachin	Tendulkar
Rahul	Dravid
Anil	Kumble
MS	Dhoni
Yuvraj	Singh
Saurav	Ganguly
Virender	Sehwag
Harbhajan	Singh
Praveen	Kumar
Ishant	Sharma

- (*C*4) Find player IDs of players who have batted in either match 2755 or 2689.
  The tuple relational calculus formulation of this query is given in Fig. 4.34.

$$\{e.PlayerID \mid e \in Player \wedge \exists j(j \in Batting \wedge e.PlayerID = j.PID \wedge j.MatchID = \text{'2755'} \vee j.MatchID = \text{'2689'})\}$$

Figure 4.34 Tuple relational calculus expression for query *C*4

We are now looking for values of the column *PlayerID* from rows *e,* such that *e* is a row in the table *Player* and there exists a row in table *Batting* for that player, such that the Match ID is either 2755 or 2689. Note that the above formulation will not retrieve any duplicates of *PlayerID* even if the player has batted in both matches 2755 and 2689 (why not?). This query could also have been formulated by specifying only the table *Batting,* such that PIDs could be retrieved from that table for rows that met the given condition.

The result of query *C*4 as formulated in tuple relational calculus in Fig. 4.34 is presented in Table 4.40. It should be noted that the result has no duplicates. We would obtain duplicates if we directly retrieve all the PIDs from the table *Batting* since some players have played in both matches 2755 and 2689.

Table 4.40 Result of query *C*4 as posed in Fig. 4.34

PID
23001
25001
91001
94002
92002
89001
99002
95001
24001
99001
27001

- (*C*5) Find player IDs of players batting in both matches 2755 and 2689.
  This query is little bit different than the queries we have seen so far and may be formulated as in Fig. 4.35.

$$\{e.\ PlayerID \mid e \in Player \wedge \exists j, k(j \in Batting \wedge k \in Batting \wedge e.PlayerID = j.PID \wedge e.PlayerID = k.PID \wedge j.MatchID = \text{'2755'} \wedge k.MatchID = \text{'2689'})\}$$

Figure 4.35 Tuple relational calculus expression for query *C*5

In this query formulation, we are looking for values of the column *PlayerID* from rows *e,* such that *e* is a row in the table *Player* and there exist two rows *j* and *k* in table *Batting* for that player (the formulation ensures that player ID in all three rows *e*, *j* and *k* from the two tables are identical), such that one of them corresponds to match 2755 and the other to match 2689.

The result of query *C*5 is presented in Table 4.41.

Table 4.41 Result of query *C*5 as posed in Fig. 4.35

PID
23001
25001
27001

- (*C*6) Find player IDs of players bowling in 2755 but not in 2689.

This query is also different than the two earlier queries and may be formulated as in Fig. 4.36.

$$\{e.\ PlayerID \mid e \in Player \wedge \exists j \wedge \text{not } (\exists k)\ (j, k \in Bowling \wedge e.PlayerID = j\ PID \wedge e.\ PlayerID = k.\ PID \wedge j.\ MatchID = \text{'2755'} \wedge k.MatchID = \text{'2689'}\}$$

Figure 4.36 Tuple relational calculus expression for query *C*6

In this query formulation, we are looking for values of the column *PlayerID* from rows *e,* such that it is a row in table *Player* and there exists a row *j* in table *Bowling* for that player, such that the Match ID is 2755 and, in addition, there does not exist a row *k* in *Bowling* for him, such that the Match ID is 2689. Note the construct *not (∃ k)* which means that a row *k* meeting the condition specified does not exist.

The result of query *C*6 is presented in Table 4.42.

Table 4.42 Result of query *C*6 as formulated in Fig. 4.36

PlayerID
91001
92002
94002

- (*C* 7) Find the names of the cricket grounds that Dhoni has batted on.

This query is somewhat more complex because player names are in the table *Player*, batting information is in the table *Batting* and ground information is in the table *Match*. The query may be formulated in tuple relational calculus as presented in Fig. 4.37.

$$\{s.Ground \mid s \in Match \wedge \exists y, z\ (y \in Player \wedge z \in Batting \wedge z.\ PID = y.\ PlayerID \wedge z.MatchID = s.MatchID \wedge\ y.LName = \text{'Dhoni'})\}$$

Figure 4.37 Tuple relational calculus formula for query *C*7

The above tuple relational calculus formula specifies that we wish to retrieve values of the attribute *Ground* from rows *s,* such that *s* is in table *Match* and there exist rows *y* and *z,* such that *y* is in table *Player* and *z* is in table *Batting* meeting the conditions specified. The conditions are that the row *y* is the row in table *Player* for a player whose last name is Dhoni and *z* is a row in *Batting* for the player *y*

(that is, it is for the player with player ID of Dhoni) in match *s*. Note that we have not checked the first name of Dhoni assuming that there is only one player by that last name.

The result of query *C*7 is presented in Table 4.43.

Table 4.43 Result of Query *C*7

Ground
Brisbane
Colombo

- (*C*8) Find the names of players who have not batted in any match in the ODI cricket database given in Tables 3.1 to 3.4.

This query can be formulated in a simple way because it only requires checking that a player has not batted.

$$\{s.\ FName, LName \mid s \in Player \wedge \text{Not } \exists e\ (e \in Batting \wedge e.\ PID = s.\ PlayerID)\}$$

Figure 4.38 Tuple relational calculus formula for query *C*8

The above tuple relational calculus expression specifies that we wish to retrieve values of the columns *FName* and *LName* from rows *s,* such that row *s* is in the table *Player* and a row *e* does not exist in the table *Batting,* such that this row *e* is for the player *s*.

The result of query *C*8 is given in Table 4.44.

Table 4.44 Result of query *C*8 as formulated in Fig. 4.38

FName	*LName*
Brian	Lara
Rahul	Dravid
Herschelle	Gibbs
Shane	Warne
Shaun	Pollock
Michael	Vaughan
Inzamam	Ul-Huq
Stephen	Fleming
Heath	Streak
Anil	Kumble
Gary	Kirsten
Jacques	Kallis
Daniel	Vettori
Saurav	Ganguly
Virender	Sehwag

- (*C*9) Find the match IDs of matches that have batting records.

This is a simple query using only one table. It can be formulated as given in Fig. 4.39.

The formula in Fig. 4.39 simply finds the values of the column *MatchID* of each row *s* such that the row *s* is in table *Batting*.

There is however a weakness in the above formulation. It will lead to many duplicate tuples since a match has many batting records. If we wish to eliminate the duplicates, a better formulation of the query *C*9 would be as in Fig. 4.40.

$$\{s.MatchID \mid s \in Batting\}$$

Figure 4.39 Tuple relational calculus expression for query *C*9

$$\{s.\ MatchID \mid s \in\ Match \wedge \exists e\ (e \in\ Batting \wedge e.\ MatchID = s.\ MatchID\}$$

Figure 4.40 A better tuple relational calculus expression for query *C*9

This formulation retrieves rows from table *Match* (not from *Batting* which is likely to have duplicate values of *MatchID*) and this ensures that there will be only one row for each match in the table *Match*.

The result of the tuple relational calculus formulation in Fig. 4.40 is given in Table 4.45.

Table 4.45 Result of query *C*9 as posed in Fig. 4.40

MatchID
2689
2755

- (*C*10) Find the player IDs of players who have only bowled in match 2755.

In this query, we need to check that a player has bowled in match 2755 but has not bowled in any other match in the database. The tuple relational calculus formulation for this query is given in Fig. 4.41.

$$\{e.\ PlayerID \mid e \in Player \mid \exists j \wedge not(\exists k)\ (j, k \in Batting \wedge e.\ PlayerID = j.\ PID \wedge e.PlayerID = k.PID \wedge k.MatchID = \text{'2755'} \wedge k.MatchID <> \text{'2755'})\}$$

Figure 4.41 Tuple relational calculus formulation for query *C*10

In the formulation in Fig. 4.50, we look for values of the attribute *PlayerID* of each row *e,* such that it is in the table *Player* and there exists a row *j* in table *Batting* for the same player as in *e* that is for batting in match 2755 and, in addition, there does not exist a row *k* in *Batting,* such that the batting is for the same player as in *e* in a match other than 2755.

The result is given in Table 4.46.

Table 4.46 Result of query *C*10 as posed in Fig. 4.41

PID
99001
24001
23001
94002
92002
91001
23001

- (*C*11) Find the player IDs of players who have bowled in all the matches played in Australia.

We found in the discussion on relational algebra that queries involving 'all' usually involve the relational algebra operator division which was somewhat difficult to understand. It is easier to formulate such queries in tuple relational calculus.

The query may be formulated as in Fig. 4.42.

$$\{k.PlayerID \mid k \in Player \wedge \forall e(e \in Match \wedge e.Team1 = \text{'Australia'} \wedge \exists\ s(s \in Bowling \wedge s.PID = k.PlayerID \wedge s(MatchID) = e(MatchID)\}$$

Figure 4.42 Tuple relational calculus expression for query *C*11

In this query, the tuple relational calculus expression specifies that the values of the attribute *PlayerID* are to be retrieved from each row *k,* such that *k* is in the table *Player* and for each match *e* that is played in Australia there exists a row *s* in the table *Bowling,* such that it is a bowling performance for the player *k* in the match *e*. The ∀*e* (for all *e*) expression makes solving the problems which were more difficult to solve in relational algebra since there was no operation similar to the operation for all in tuple relational calculus. Unfortunately SQL also does not have an operator similar to for all and therefore problems that involve such operations in SQL are rather painful.

The result is given in Table 4.47.

Table 4.47 Result of query *C*11 posed in Fig. 4.42

MatchID	*PID*
2689	99001
2689	24001
2689	23001

4.4.7 Domain Relational Calculus

As noted earlier, domain relational calculus (DRC) is very similar to the tuple relational calculus that we have presented in the last section except that rather than using variables. ranging over tuples in TRC, the variables range over single domains of attributes in DRC. Therefore to specify a relation with *n* attributes it is necessary to specify *n* domain variables one for each attribute.

The basis of DRC essentially is that a relational database consists of the domains that represent the sets of objects in the database. A query language based on DRC therefore consists of variables that range on a single domain of the relational database. It is claimed that DRC therefore manipulates the basic database objects since domain values interact more directly with the semantics expressed by the relations and produces simpler and more English-like languages.

The following query shows the general form of a DRC query

{<*PlayerID, LName, FName, Country, YBorn, BPlace, FTest, F*(*PlayerID, LName, FName, Country, YBorn, BPlace, FTest*) }

In this query *PlayerID, LName, FName, Country, YBorn, BPlace, FTest* represent domain variables that the user wishes to retrieve and *F*(.) is a formula that specifies the condition. The formula *F* may be very simple, for example, stating that all these domains together form the relation *Player*. The formula *F*(.) is shown below.

<*PlayerID, LName, FName, Country, YBorn, BPlace, FTest*> ∈ *Player*

We now present a number of queries and formulate them in domain relational calculus.

- (*D*1) Find the names and IDs of all players.
  The domain relational calculus formulation of query *D*1 is given in Fig. 4.43.

{<Fname, Lname, PlayerID> | ∃ *Country, YBorn, Bplace, FTest* (<*PlayerID, LName, FName, Country, YBorn, BPlace, FTest*>) ∈ Player}

Figure 4.43 Domain relational calculus expression for query *D*1

- (*D*2) Find IDs of all players from India.
  The domain relational calculus formulation of query *D*2 is given in Fig. 4.44.

{<PlayerID> | ∃ *LName, FName, Country, YBorn, BPlace, FTest* (<*PlayerID, LName, FName, Country, YBorn, BPlace, FTest*>) ∈ *Player* ∧ Country = 'India'}

Figure 4.44 Domain relational calculus expression for query *D*2

The above domain relational calculus expression specifies that we wish to retrieve values of the domain *PlayerID* of all rows from *Player* such that they have the value of the domain *Country* India.

- (*D*3) Find the names of all players from India who have batted in a match.
  The Domain relational calculus formulation of query *D*3 is given in Fig. 4.45.

{<FName, LName> | ∃ (*PlayerID, Country, YBorn, BPlace, FTest* (<*PlayerID, LName, FName, Country, YBorn, BPlace, FTest*>) ∈ *Player* ∧ ∃ MatchID, PID, Order, HOut, FOW, NRuns, Mts, NBalls, Fours, Sixes (<MatchID, PID, Order, HOut, FOW, NRuns, Mts, NBalls, Fours, Sixes> ∈ *Batting* ∧ Country = 'India' ∧ PlayerID = PID)}

Figure 4.45 Domain relational calculus formulation for query *D*3

The above expression looks complex because of the number of attributes in the two relations *Player* and *Batting*. Essentially the domain relational calculus specifies the columns that we wish to retrieve from those tuples from *Player* for whom the domain *Country* is India and there exists a tuple in *Batting* for the same player. In other words we retrieve only the names for a player from India if there is an entry for this player in *Batting.*

- (*D*4) Find player IDs of players who have batted in either match 2755 or 2689.
  The tuple relational calculus formulation of this query is given in Fig. 4.46.

{<PlayerID> | ∃ *LName, FNane, Country, YBorn, BPlace, FTest* (<*PlayerID, LName, FName, Country, YBorn, BPlace, FTest*>) ∈ *Player* ∧ ∃ MatchID, PID, Order, HOut, FOW, NRuns, Mts, NBalls, Fours, Sixes (<MatchID, PID, Order, HOut, FOW, NRuns, Mts, NBalls, Fours, Sixes> ∈ *Batting* ∧ MatchID = '2755' ∨ MatchID = '2689')}

Figure 4.46 Domain relational calculus expression for query *D*4

{<PlayerID> | ∃ *LName, FNane, Country, YBorn, BPlace, FTest* (<*PlayerID, LName, FName, Country, YBorn, BPlace, FTest*>) ∈ *Player* ∧ ∃ MatchID, PID, Order, HOut, FOW, NRuns, Mts, NBalls, Fours, Sixes (<MatchID, PID, Order, HOut, FOW, NRuns, Mts, NBalls, Fours, Sixes> ∈ *Batting* ∧ ∃ MatchID1, PID1, Order1, HOut1, FOW1, NRuns1, Mts1, NBalls1, Fours1, Sixes1 (< MatchID1, PID1, Order1, HOut1, FOW1, NRuns1, Mst1, NBalls1, Fours1, Sixes1> ∈ *Batting* ∧ MatchID = '2755' ∧ MatchID = '2689'

Figure 4.47 Domain relational calculus expression for query *D*5

We are now looking for values of the domain *PlayerID* from the relation *Player* such that there exists a row in table *Batting* for that player where the *Match ID* is either 2755 or 2689. Note that the above formulation will not retrieve any duplicates of *PlayerID* even if the player has batted in both matches 2755 and 2689 (why not?). This query could also have been formulated by specifying only the table *Batting* such that *PIDs* could be retrieved from that table for rows that met the given condition.

- (*D*5) Find player IDs of players batting in both matches 2755 and 2689.
  This query is little bit different than the queries we have seen so far and may be formulated as in Fig. 4.47.
  In this query formulation, we are looking for values of the domain *PlayerID* from rows the table *Player* for which there exist two rows in the table *Batting* for that player, such that one of them corresponds to match 2755 and the other to match 2689.

4.4.8 Well-Formed Formula (WFF)

The queries formulated in tuple relational calculus and domain relational calculus above should have given the reader some idea about the features of the two variants of relational calculus. We have however not specified what expressions in calculus are acceptable. This is elaborated ahead.

We have already discussed the concept of free and bound variables. Also we have indicated earlier, that the general form of relational calculus expressions is $\{(x,y,z) \mid P(x, y, z)\}$ where P is a formula in which the variables *x, y, z* are free. An acceptable formula is called a *well-formed formula* or WFF. WFFs may be defined recursively as in Fig. 4.48.

1. Any term is a WFF (a term is either an indexed tuple or a comparison of two indexed tuples).
2. If F and a WFF, so is (not F).
3. If F and G are WFFs, so are $(F \wedge G)$ and $(F \vee G)$.
4. If F is a WFF in which x appears as a free variable, then $\exists x(F)$ and $\forall x(F)$ are WFFs.
5. No other formulas are WFFs.

Figure 4.48 Definition of well-formed formulas in tuple relational calculus

4.5 RELATIONAL ALGEBRA VS RELATIONAL CALCULUS

It is useful to discuss the selective power of relational algebra and relational calculus. Any language that has the power of relational calculus is called *relationally complete*. Codd in his paper in 1970 proposed that any language that was not relationally complete should be considered inadequate. He showed that relational algebra is equivalent to relational calculus and is therefore relationally complete. It is in fact possible to design algorithms that will translate a relational algebra expression into an equivalent relational calculus expression and vice versa. These algorithms are called *reduction algorithms*. We will not discuss them here, but the reader can refer to Codd (1970) or Ullman (1982).

Given that relational algebra and relational calculus have equivalent selective power, one may now ask which one of these two languages should be preferred. Codd, in his 1971 paper, claimed that relational calculus was superior to algebra although the algebra did not require use of quantifiers for all (∀) and there exists (∃). He gave four reasons for preferring relational calculus:

1. *Ease of Augmentation*—Since the algebra and the calculus provided only basic selective power, it is necessary to augment that power by providing a facility to invoke a number of library functions (for example, average, sum, maximum). This augmentation is more easily done in relational calculus than in algebra.
2. *Scope for Search Optimization*—Since the calculus only specifies what is to be retrieved and not how it is to be retrieved, it is easier for a database system to optimize database search to answer queries that are posed in calculus. Queries posed in relational algebra can also be optimized but such optimization is likely to be only local since it would be difficult for a system to derive the properties of the data that needs to be retrieved given the operators that the user has specified.
3. *Authorization Capability*—It is likely to be easier to specify authorization by using relational calculus since authorization is likely to be based on the properties of the data that a user may be authorized to access or update.
4. *Closeness to Natural Language*—It is much more natural for users in relational calculus to specify what data they wish to retrieve than how the data is to be retrieved. Thus a language based on relational calculus is likely to be more user-friendly and easier to use for users with limited computing background.

As noted above, the relational model requires that the language available to the user should at least provide the power of relational algebra without making use of iterations or recursion statements. Relational calculus does that. So does the commercial language, SQL, which we will discuss in detail in Chapter 5.

SUMMARY

This chapter has covered the following:

- Codd defined two formal languages, viz., relational algebra and relational calculus.
- Relational algebra is a procedural language that includes the operators selection, projection, Cartesian product, equi-join, natural join, outer join, union, intersection, difference and division.
- Relational algebra notation has been presented. For example, a selection may be written as Players [Country = 'India'] and the notation has been used to formulate a variety of queries.
- Selection and projection are unary operators, operating on a single table.
- Cartesian product and join are binary operators, operating on two tables. Join is used to combine two related tables on a join condition, for example, the table *Player* and *Batting*.
- Outer join is a join operator in which matching rows from two tables as well as rows that do not match are shown. An outer join may be left outer join or a right outer join or a full outer join.
- It is possible to use set operations in relational algebra including union, intersection and difference.
- Division operator may be applied to two tables and is related to the Cartesian product operator, it may be considered the opposite of the product.

- Background of predicate calculus and relational calculus is presented including meaning of propositions, the implied operator and parameterized propositions.
- Meaning of the existential and universal quantifiers is explained and use of calculus to formulate a variety of queries is illustrated using a number of examples.
- Two notations of relational calculus, viz., the tuple relational calculus (TRC) and domain relational calculus (DRC) are presented.
- TRC uses tuple variables (rows of a table) while DRC uses variables that range over domains to formulate queries.
- Well-formed formulas (WFF) are the acceptable expressions in calculus; WFF is defined.
- Relative merits of algebra and calculus and their relative power to express queries are discussed. The calculus may be preferred for ease of augmentation and for better scope for optimization.

REVIEW QUESTIONS

1. What is the functoin of operations restriction, projection, Cartesian product, equi-join, natural join, outer join, union, intersection, difference, and division? (Section 4.2)
2. Describe the relational algebra notation for restriction, projection and natural join. (Section 4.2)
3. Give examples to explain a database query? Show how to use relational algebra to formulate a variety of queries. (Section 4.2)
4. What is the function of division operation? Can it be expressed in terms of other relational algebra operations?
5. Explain the relational calculus notation and well-formed formulas. (Section 4.3)
6. Explain the notation used in expressing queries in tuple relational calculus. (Section 4.3)
7. Give examples to explain the meaning of the existential quantifier and universal quantifier. (Section 4.4)
8. Describe how to use tuple relational calculus to formulate a variety of queries. (Section 4.4)
9. Give a number of examples of using domain relational calculus to formulate a variety of queries. (Section 4.4)
10. Select two queries and formulae them using tuple relational calculus and the domain relational calculus. (Section 4.4)
11. Explain the relative merits of algebra and calculus and their relative power to express queries. (Section 4.5)

SHORT ANSWER QUESTIONS

1. What is relational algebra?
2. What is relational calculus?
3. What is relational algebra operator selection?
4. Give an example of using selection in Table 4.5.
5. What is relational algebra operator projection?

6. Give an example of applying projection to Table 4.13.
7. What is relational algebra operator Cartesian product?
8. What is the Cartesian product of Tables 4.11 and 4.12?
9. How is the equi-join different from natural join?
10. Define the properties of two relations to be union-compatible.
11. Are Figs 4.11 and 4.12 union-compatible?
12. Find the natural join from Cartesian product of Tables 4.11 and 4.12.
13. What is the left outer join? What does the operation do?
14. What is the right outer join?
15. Find the full outer-join of Tables 4.10 and 4.11. What does the result tell us?
16. What is the relational operator division used for?
17. Give an example of two small relations and divide them to illustrate division.
18. Give examples of universal and existential quantifiers.
19. Express the query 'find all the information about players from India' using TRC and DRC.

MULTIPLE CHOICE QUESTIONS

1. Which one of the following is **not** correct?
   (a) A restriction selects one or more rows of a relation.
   (b) A projection selects one or more columns of a relation.
   (c) A join glues each row of one relation with all the rows of the other.
   (d) A difference gives all the rows in the first relation that are not in the second.
2. Which one of the following is correct?
   (a) A Cartesian product of two relations is the same as their union.
   (b) A join of two relations is a selection on their Cartesian product.
   (c) A join is always an equi-join or a natural join.
   (d) The degree of the equi-join of two relations is the same as the degree of the natural join of the same relations.
3. Consider the following query:

   (((P WHERE COLOUR = 'RED') [P#] JOIN SP) [S#] JOIN S) [SNAME]

   Which one of the following set of operations does the above query involve?
   (a) 1 selection, 2 joins, 3 projections
   (b) 1 projection, 2 joins, 3 selections
   (c) 2 selections, 2 joins, 3 projections
   (d) 1 selection, 2 joins, 3 intersections
4. Which one of the following does not always have the same list of attributes that the operands have?
   (a) Project (b) Select (c) Union (d) Difference
5. Which one of the following always results in fewer rows than the relation *R*? Assume neither of the two relations is empty.
   (a) $R \rhd\lhd S$ (b) $R - S$ (c) R/S (d) *R* intersection *S*
   (e) None of the above

6. Which one of the following always results in at least as many rows as in the relation R? Assume neither of the two relations is empty.
   (a) $R \rhd\lhd S$ (b) R union S (c) $R - S$ (d) R/S
   (e) R intersection S
7. Which two of the following operators require elimination of duplicates?
   (a) Projection (b) Intersection (c) Difference (d) Union
   (e) Join
8. Which one of the following is a complete set of algebra operations?
   (a) selection, projection, union, difference and Cartesian product
   (b) selection, projection and join
   (c) selection, projection and Cartesian product
   (d) selection, projection, division and Cartesian product
9. Consider the following binary operators. Which one of the operator does not require the two operands to be union-compatible (that is, having exactly the same number and types of attributes)?
   (a) Intersection (b) Difference (c) Union (d) Division
10. Relational division has operands and results named as given below. Which of the following is not a relation?
   (a) Dividend (b) Divisor (c) Quotient (d) Remainder
   (e) None of the above
11. $R[A$ op $B]S$ is a join of the relations R and S. Which one of the following is **not** correct?
   (a) $R[A = B]S$ may be empty even when R and S are not
   (b) $R[A = B]S = S[B = A]R$
   (c) $R[A < B]S = S[B > A]R$
   (d) $R[A < B]S = R \times S - R[A > B]S - R[A = B]S$
   (e) $R[A < B]\mathrm{S} = R \times S - S[B < A]R$
12. Which of the following are correct?
   (a) $R \rhd\lhd S = S \rhd\lhd R$ (b) $R - S = S - R$
   (c) $R/S = S/R$ (d) $S \cup T = (S - T) \cup (S$ intersect $R) \cup (T - S)$
13. Consider the join of two relations R and S that are of degree x and y respectively.
   Which one of the following is correct?
   (a) The degree of an equi-join of relations R and S is $x + y$.
   (b) Degree of a natural-join of relations R and S is always $x + y - 1$.
   (c) If the cardinality of R is a and of S is b, then the cardinality of the division R/S is a/b.
   (d) A projection of relation R has the same cardinality as R but the degree of the projected relation is equal to the number of attributes that the projection specifies.
14. Which one of the following is correct? Note that $\cup$ represents union while $\cap$ represents intersection.
   (a) $(R \cup S) \cap T = (R \cap T) \cup S$ (b) $R - S = S - R$
   (c) $(R/S) \times S = R$ (d) $S \cup T = (S - T) \cup (S \cap R) \cup (T - S)$
15. Let $T = R/S$ where R is an enrolment relation and S is a subject relation with degrees 2 and 1 and cardinalities m and n respectively. The attributes of R are student_id and subject_id and the attribute of S is subject_id. Which of the following are correct? (Note: $T \times S$ is the Cartesian product of T and S).

(a) T is of degree 1.
(b) T has cardinality no larger than m/n.
(c) $(R - T \times S)$[student_id] does not include any student who is in T.
(d) T are all the students doing all the subjects in S.
(e) All of the above

16. Let R be an enrolment relation and S be a subject relation. The relation R has two attributes student_id (call it A) and subject_id (call it B) while the relation S consists of only one attribute subject_id (call it also B).
Which one of the following is correct?
(a) $((R[A] \times S) - R)[A]$ is the list of student_id values (attribute A) in R.
(b) $R[A] - ((R[A] \times S) - R)[A]$ is a null relation.
(c) $(R - R[A] \times S)[A]$ is the list of all the student IDs for those doing all the subjects in S.
(d) The expression $R[A] - ((R[A] \times S) - R)[A]$ is the list of all the student IDs for students not doing all the subjects in S.

17. Which one of the following is **not** correct?
(a) Relational algebra provides a collection of operations that can actually be used to build some desired relation; the calculus provides a notation for formulating the definition of that desired relation.
(b) In relational algebra, one specifies what operations must be executed while in relational calculus, the specification of operations is left to the system.
(c) Relational algebra is procedural while relational calculus is nonprocedural.
(d) Using relational calculus one may specify more complex queries than using relational algebra.

EXERCISES

1. Consider the following student database:
Student(ID, Name, Address, City, Tutor)
Enrolment(ID, Code, Marks)
Course(Code, Title, Department)
Formulate the following queries in relational algebra using the student database above. Assume that the attribute *Tutor* is the ID of the tutor.
(a) List all the student names.
(b) List all school and course names.
(c) Find student names of students that have no tutor.
(d) Find student names of students that are doing computer science courses.
(e) Find student names of students that have Suresh Chandra as tutor.

2. The following questions deal with Cartesian product and join:
(a) Consider the relations in the student database exercise above. How many rows will there be in the Cartesian product of the three tables?
(b) How many attributes does the natural join of the two tables *Student* and *Enrolment* have? Explain.

3. Express
   (a) Division R/S in terms of projection, Cartesian product and difference.
   (b) Natural join in terms of projection, Cartesian product and selection.
   (c) Intersection in terms of projection, union and difference.
4. Consider the following two queries for the cricket database:
   (a) Find the youngest player in Table 3.1
   (b) Find the oldest player in Table 3.1

   Can the above two queries be formulated using relational calculus? If yes, formulate the two queries.
5. Explain the motivation for TRC and DRC and discuss the differences between the two approaches.
6. Show how a TRC query may be written as SQL and SQL query may be written as a TRC query.
7. Study the basics of QBE and use QBE in Microsoft Access to understand it and relate it to DRC. Can a DRC be translated to QBE? Show by using an example.
8. Write a query that involves a division and show how it may be expressed in TRC.

PROJECTS

1. Write algorithms and then implement them in the programming language of your choice for all the relational algebra operations. This could be a group project.
2. Using the Web, search for open source software for relational calculus (TRC or DRC). Install the software and test it by posing some queries to it.

LAB EXERCISES

1. Formulate the following queries for the ODI database using relational algebra:
   (a) Find the player's last name that was the youngest amongst the players in the database when he played in his first test match.
   (b) Find the ID of the player that hit the largest number of sixes in an ODI match in the example database.
   (c) Find the list of names of players that have scored a century in the example ODI database.
   (d) Find the average strike rate of players that have scored a century in the example ODI database.
   (e) Find the names of players that have taken four wickets or more in the example ODI database.
   (f) What is the maximum number of wickets that a bowler has taken in an ODI match in the example database? Find the ID of that bowler.
   (g) Find the name of the youngest player in the example database.
   (h) Find the IDs of players that have never scored a four or a six in all matches they have batted in.
2. Formulate the above queries in Tuple Relational Calculus.

BIBLIOGRAPHY

For Students

A number of books listed below are easy to read for students.

For Instructors

Codd's papers and his out-of-print book (which may be downloaded from the Web) are the authoritative documents on the relational model. C. J. Date's books are also a good source of information.

Codd, E. F., "A relational model of data for large shared data banks", *Communications of the ACM*, Vol. 13, No. 6, 1970, pp 377-387.

Codd, E. F., "Relational database: a practical foundation for productivity", *Communications of the ACM*, Vol. 25, No. 2, 1982, pp. 109-117.

Codd, E. F., *The Relational Model for Database Management: Version 2*, Addison-Wesley Publishing Company, Inc, 1990. (This book is out of print now but may be downloaded from http://portal.acm.org/citation.cfm?id=77708.)

Date, C. J., "The Outer Join", In *Relational Database: Selected Writings*, Addison-Wesley, 1986.

Date, C. J., *Database in Depth: Relational Theory for Practitioners*, O'Reilly Media, 2005.

Garcia-Molina, H., J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, Prentice Hall, 2002.

Garcia-Molina, H., J. D. Ullman, and J. Widom, *Database System Implementation*, Prentice Hall, 2000.

Kifer, M., Bernstein, A., and P. M. Lewis, *Database Systems: An Application-Oriented Approach*, Second Edition, Addison-Wesley, 2006.

Ullman, J. D., *Principles of Database Systems*, Computer Science Press, 1982. (May be downloaded from http://www.filestube.com/d/database+systems+ullman)

CHAPTER 5

Structured Query Language (SQL)

OBJECTIVES

- To explain how SQL is used for data definition.
- To describe how SQL is used for retrieving information from a database.
- To explain how SQL is used for inserting, deleting and modifying information in a database.
- To explain the use of SQL in implementing correlated and complex queries including outer joins.
- To discuss embedded SQL and dynamic SQL.
- To discuss how data integrity is maintained in a relational database using SQL.

KEYWORDS

Queries, SQL, CREATE TABLE, CREATE INDEX, CREATE VIEW, CREATE DOMAIN, DDL, DROP TABLE, DROP INDEX, DROP VIEW, DROP DOMAIN, SELECT, INSERT, WHERE, GROUP BY, DELETE, UPDATE, VIEW, grouping, aggregation, functions, AVG, COUNT, MIN, MAX, SUM, nested queries, subqueries, outer join, integrity constraints, primary key, secondary key, domain, NULL, embedded SQL, dynamic SQL, complex queries, correlated queries, existential integrity, referential integrity, UNIQUE, triggers, active databases.

An eye for an eye makes the whole world blind.

Mahatma Gandhi

5.1 INTRODUCTION

We have discussed the relational model in Chapter 3, which is a model for implementing a database that has been modeled using Entity-Relationship modeling or a similar modeling methodology. The relational model is widely used because of its simplicity. It is a conceptual view of the data as well as a high level view of the implementation of the database on the computer. The model does not concern itself with the detailed bits and bytes view of the data.

In the last chapter, we studied two formal data manipulation languages in detail, viz., relational algebra and relational calculus. It is not appropriate to use either of them in a commercial database system that would be used by users with a computing background as well as by those that have little computing knowledge because calculus and algebra are not user-friendly languages and they do not provide facilities for counting or for computing averages, or maximum and minimum values. In this chapter we discuss another query language called the Structured Query Language (SQL) that is much easier for users. It is a commercial query language based on relational calculus.

The query language SQL is used in almost all commercial relational DBMS. Although other query languages exist, SQL has become a de facto standard for relational database systems.

SQL is a nonprocedural language that originated at IBM during the building of the well-known experimental relational DBMS called System R. Originally the user interface to System R was called SEQUEL designed by Donald Chamberlain and Raymond Bryce in 1974. The language was later modified in 1976–77 and its name changed to SEQUEL2. These languages have undergone significant changes over time and the current language was named the Structured Query Language (SQL) in the early 1980s.

Just as for other programming languages, the American National Standards Institute (ANSI) published a standard definition of SQL in 1986 which was subsequently adopted by the International Organization for Standardization (ISO) in 1987. Since then there have been a number of SQL standards, the most recent being SQL : 2003. We will give more details about the SQL standards at the end of this chapter. The focus of this chapter is on the basic SQL constructs. We will not discuss some of the newer features included in the more recent standards.

This chapter is organized as follows. After introducing SQL next the data definition features of SQL are described in Section 5.3. Section 5.4 covers data retrieval features of SQL. This section includes some simple examples of SQL that involve only one table followed by examples that involve more than one table using joins and subqueries, examples using built-in functions followed by explanation of correlated and complex SQL queries. GROUPING and HAVING are presented next. SQL update commands and additional data definition commands are then discussed. Views are described in Section 5.5 and Section 5.6 discusses embedded SQL and dynamic SQL. Section 5.7 explains how SQL may be used for enforcing the final component of the relational model and data integrity. Existential integrity, referential integrity, domains, and nulls have also been discussed here. Triggers are briefly described in Section 5.8. Authentication and authorization

is discussed briefly in Section 5.9. Additional data definition commands are described in Section 5.10. A summary of SQL commands is presented in Section 5.11 and SQL standards are discussed in Section 5.12.

5.2 SQL

As noted above, SQL is designed to support data definition, data manipulation, and data control in relational database systems.

A user of a DBMS using SQL may use the query language interactively or through a host language like C++. An interactive web page could also be the front end of the SQL query in which the user supplies some parameters before the query is executed by the server. We will mainly discuss the interactive use of SQL although SQL commands embedded in a procedural language will also be discussed briefly.

SQL consists of the following facilities:

- *Data Definition Language (DDL)*—Before a database can be used, it must be created. Data definition language is used for defining the database to the computer system. The DDL includes facilities to create and destroy databases and database objects, for example, for creating a new table or a view, deleting a table or a view, altering or expanding the definition of a table (by entering a new column) and creating an index on a table, as well as commands for dropping an index. DDL also allows a number of integrity constraints to be defined either at the time of creating a table or later.
- *Data Manipulation Language (DML)*—These DML facilities include commands for retrieving information from one or more tables and updating information in tables including inserting and deleting rows. The commands SELECT, UPDATE, INSERT and DELETE are available for these operations. The SELECT command forms the basis for retrieving data from the database. The command UPDATE forms the basis for modifying information that is already in the database. INSERT is used for inserting new information in the database while DELETE is used for removing information from the database.
- *Data Control Language (DCL)*—The DCL facilities include database security control including privileges and revoke privileges. Therefore DCL is responsible for managing database security including management of which users have access to which tables and what data.
- *New Features*—Each new standard has introduced a variety of new features in the language. For example, SQL:2003 includes XML. Unfortunately we will not be able to cover these features in this book.

It should be noted that SQL does not require any particular appearance of SQL queries and is not case sensitive except for object names. Although it is much easier to read queries that are well structured and indented, SQL does not require it. In fact, the whole query could be typed in one long line if that is what the user wishes to do. Sometimes though different DBMS may use somewhat different conventions. We will use the convention of writing SQL commands as upper case letters like we did in the last paragraph and names of tables and their columns in lower case. In our view this makes it easier to read the queries.

We first study some of the data definition features of the language in the next section.

We will continue to use the ODI cricket database to illustrate various features of the query language in this section. As a reminder, the schemas of the four tables *Match*, *Player, Batting* and *Bowling* are reproduced

in Fig. 5.1 and the tables are reproduced in Tables 5.1, 5.2, 5.3 and 5.4. Additional data for these tables is presented at the end of this chapter. For example, Appendix 5.1 includes information on another 50 players. Appendix 5.2 includes another 35 rows for the table *Batting* and Appendix 5.3 includes another 35 rows for the table *Bowling*. These may be used for exercises that require more data.

Match (MatchID, Team1, Team2, Ground, Date, Winner)
Player (PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
Batting (MachID, PID, Order, HOut, FOW, NRuns, Mts, Nballs, Fours, Sixes)
Bowling (MatchID, PID, NOvers, Maidens, NRuns, NWickets)

Figure 5.1 Database schema for the ODI cricket database

We now reproduce the sample tables for the database that were presented in Chapter 3. The first table, Table 5.1, is *Match*.

Table 5.1 The relation *Match*

MatchId	*Team1*	*Team2*	*Ground*	*Date*	*Winner*
2324	Pakistan	India	Peshawar	6/2/2006	Team1
2327	Pakistan	India	Rawalpindi	11/2/2006	Team2
2357	India	England	Delhi	28/3/2006	Team1
2377	West Indies	India	Kingston	18/5/2006	Team2
2404a	Sri Lanka	India	Colombo	16/8/2006	Abandoned
2440	India	Australia	Mohali	29/10/2006	Team2
2449	South Africa	India	Cape Town	26/11/2006	Team1
2480	India	West Indies	Nagpur	21/1/2007	Team1
2493	India	West Indies	Vadodara	31/1/2007	Team1
2520	India	Sri Lanka	Rajkot	11/2/2007	Team2
2611	England	India	Southampton	21/8/2007	Team1
2632	India	Australia	Mumbai	17/10/2007	Team1
2643	India	Pakistan	Guwahati	5/11/2007	Team1
2675	Australia	India	Melbourne	10/2/2008	Team2
2681	India	Sri Lanka	Adelaide	19/2/2008	Team1
2688	Australia	India	Sydney	2/3/2008	Team2
2689	Australia	India	Brisbane	4/3/2008	Team2
2717	Pakistan	India	Karachi	26/6/2008	Team2
2750	Sri Lanka	India	Colombo	24/8/2008	Team2
2755	Sri Lanka	India	Colombo	27/8/2008	Team2

The second table, Table 5.2, is *Player*.

Table 5.2 The relation *Player*

PlayerID	*LName*	*FName*	*Country*	*YBorn*	*BPlace*	*FTest*
89001	Tendulkar	Sachin	India	1973	Mumbai	1989
90001	Lara	Brian	West Indies	1969	Santa Cruz	1990
95001	Ponting	Ricky	Australia	1974	Launceston	1995
96001	Dravid	Rahul	India	1973	Indore	1996
96002	Gibbs	Herschelle	South Africa	1974	Cape Town	1996
92001	Warne	Shane	Australia	1969	Melbourne	1992
95002	Pollock	Shaun	South Africa	1973	Port Elizabeth	1995
99003	Vaughan	Michael	England	1974	Manchester	1999
92003	Ul-Huq	Inzamam	Pakistan	1970	Multan	1992
94004	Fleming	Stephen	New Zealand	1973	Christchurch	1994
93002	Streak	Heath	Zimbabwe	1974	Bulawayo	1993
90002	Kumble	Anil	India	1970	Bangalore	1990
93003	Kirsten	Gary	South Africa	1967	Cape Town	1993
95003	Kallis	Jacques	SouthAfrica	1975	Cape Town	1995
94002	Vaas	Chaminda	Sri Lanka	1974	Mattumagala	1994
92002	Muralitharan	Muthiah	Sri Lanka	1972	Kandy	1992
97004	Vettori	Daniel	NewZealand	1979	Auckland	1997
25001	Dhoni	MS	India	1981	Ranchi	2005
23001	Singh	Yuvraj	India	1981	Chandigarh	2003
96003	Ganguly	Saurav	India	1972	Calcutta	1996
99002	Gilchrist	Adam	Australia	1971	Bellingen	1999
24001	Symonds	Andrew	Australia	1975	Birmingham	2004
99001	Lee	Brett	Australia	1976	Wollongong	1999
91001	Jayasuriya	Sanath	Sri Lanka	1969	Matara	1991
21001	Sehwag	Virender	India	1978	Delhi	2001
98001	Afridi	Shahid	Pakistan	1980	Khyber Agency	1998
98002	Singh	Harbhajan	India	1980	Jalandhar	1998
27001	Kumar	Praveen	India	1986	Meerut	NULL
27002	Sharma	Ishant	India	1988	Delhi	2007

The third table is Table 5.3 *Batting*.

Table 5.3 The relation *Batting*

MatchID	*PID*	*Order*	*HOut*	*FOW*	*NRuns*	*Mts*	*Nballs*	*Fours*	*Six*
2755	23001	3	C	51	0	12	6	0	0
2755	25001	5	C	232	71	104	80	4	0
2755	91001	1	C	74	60	85	52	8	2
2755	94002	7	LBW	157	17	23	29	1	0
2755	92002	11	NO	NO	1	11	7	0	0
2689	89001	2	C	205	91	176	121	7	0
2689	23001	4	C	175	38	27	38	2	2
2689	25001	5	C	240	36	52	37	2	1
2689	99002	1	C	2	2	3	3	0	0
2689	95001	3	C	8	1	9	7	0	0
2689	24001	5	C	123	42	81	56	2	1
2689	99001	8	B	228	7	20	12	0	0
2689	27001	9	C	255	7	7	7	1	0
2755	27001	9	B	257	2	7	6	0	0
2689	98002	8	LBW	249	3	7	3	0	0
2755	98002	8	RO	253	2	7	4	0	0

The fourth and the last table, Table 5.4, is *Bowling*.

Table 5.4 The relation *Bowling*

MatchID	*PID*	*NOvers*	*Maidens*	*NRuns*	*NWickets*
2689	99001	10	0	58	1
2689	24001	3	0	27	1
2689	23001	3	0	15	0
2755	94002	9	1	40	1
2755	92002	10	0	56	1
2755	91001	4	0	29	0
2755	23001	10	0	53	2
2755	98002	10	0	40	3
2689	98002	10	0	44	1

We will now discuss the various features of SQL.

5.3 DATA DEFINITION–CREATE, ALTER AND DROP COMMANDS

As noted in the last section, DDL includes commands to create and destroy databases and database objects. The basic DDL commands available in SQL are discussed below. Additional DDL commands are presented later in this chapter in Section 5.4.12.

5.3.1 CREATE Command

CREATE is the most commonly used DDL command. It may be used to create a database, a schema, a new table, a new index, a new view, a new assertion or a new trigger.

We only look at creating an empty table. To create the tables *Match, Player, Batting* and *Bowling*, we use the CREATE TABLE command. Before formulating a CREATE TABLE command, it is necessary that each column type be looked at and likely values considered so that its data type be defined. In Fig. 5.2, we have arbitrarily assumed that *Team*1 and *Team*2 are 15 characters long but that will not be sufficient if we wanted to write one of the teams as Republic of South Africa. Therefore, care must be taken in defining data types for each column. Similarly, the data type of *Ground*, CHAR(20), is not able to represent names like Melbourne Cricket Ground or M.A. Chidambaram Stadium in Chennai or Vidarbha Cricket Association Stadium in Nagpur. CHAR(60) might be a better data type if full ground names are to be used for the column *Ground*. In addition to data type for each column, it is necessary to define the primary key and any foreign keys as well as any columns that are NOT NULL or UNIQUE.

```
CREATE TABLE Match
(MatchID INTEGER,
Team1 CHAR(15),
Team2 CHAR(15),
Ground CHAR(20),
Date CHAR(10),
Result CHSR(10),
PRIMARY KEY (MatchID))
```

Figure 5.2 SQL for Data Definition including specification of primary key

The basic form of the CREATE TABLE command is given in Fig. 5.2.

The CREATE TABLE command as shown in Fig. 5.2 is used to create a new table *Match*. It includes a specification of the names of the table as well as name of all its columns, separated by comma, and their data types. For each column, a name and data type must be specified. SQL data types are discussed in Section 5.3.4. Integrity constraints may also be specified in this command. In the above data definition, we have specified that the column *MatchID* is the primary key of the table and as noted earlier it will not allow a row with a NULL primary key.

Each column name is unique starting with a letter and made up of letters, numbers or an underscore (_) but no SQL reserve words.

The CREATE command also allows default values to be assigned to the columns. If no default value is specified, the system will usually provide a NULL as the default. It is also possible to specify constraints in the CREATE TABLE command. We will discuss constraints later in this chapter.

There are several other CREATE commands, for example, CREATE DOMAIN, CREATE INDEX, CREATE VIEW. We will discuss them later in this chapter in Section 5.6.

5.3.2 ALTER Command

The ALTER command may be used to modify an existing table. ALTER may be used for adding or dropping a column, changing the column definition as well as adding or dropping a table constraint. For example, Fig. 5.3 shows how to add a column to the table *Batting* to include information on how many times a batsman was not out.

```
ALTER TABLE Batting
ADD COLUMN Not_Out INTEGER
```

Figure 5.3 Adding a Column to a Table

The new column appears as the rightmost column in the table. Once the table definition has been changed, the values of the column would need to be inserted. The default value is NULL.

A column in a table may be dropped if required as illustrated in Fig. 5.4.

```
ALTER TABLE Batting
DROP COLUMN Not_Out
```

Figure 5.4 Dropping a Column in a Table

If several columns need to be dropped then several ALTER commands are required. More examples of ALTER are given in Section 5.7.

5.3.3 Destroying a Table using the DROP Command

A table may be destroyed by using the DROP TABLE command as given in Fig. 5.5.

Dropping a table may have side effects. An index or a view may have been defined on the table and one or more columns in the table may be foreign keys in other tables. We may specify CASCADE like in Fig. 5.6 if we want all references to the table to be dropped as well. It is also possible to use RESTRICT which will not allow the DROP command to be executed if there are dependencies on the table.

```
DROP TABLE Batting
```

Figure 5.5 Dropping a Table

There are several other DROP commands, for example, DROP DOMAIN, DROP INDEX, DROP VIEW. We will discuss them later in this chapter in Section 5.7.1.

```
DROP TABLE Batting CASCADE
```

Figure 5.6 Dropping a Table using CASCADE

5.3.4 SQL Data Types

As shown in Fig. 5.2, it is required that the data type of each column be specified when a table is defined. Data types deal with numbers, text, date, time and binary objects. There are a large number of data types defined in SQL3. We list only the most commonly used data types. We have divided them in three categories.

Numerical Types

There are a variety of numerical data types available in SQL. We list some of them in Fig. 5.7.

Data Type	Description
SMALLINT	Integer, precision 5
INTEGER	Integer precision 10
REAL	Mantissa about 7
FLOAT(p)	Mantissa about 16
DOUBLE PRECISION	Same as FLOAT

Figure 5.7 SQL numerical data types

Several other numerical types are available including INTEGER(p), DECIMAL(p, s), NUMERIC(p, s) and FLOAT (p) all of which allow the user to specify precision.

Character Strings and Binary Data Types

A list of some character strings and binary data types is given in Fig. 5.8.

Data Type	Abbreviation	Description	Comments
CHARACTER(n)	CHAR(n)	Fixed length character string of length n (max n = 255 bytes). Padded on right.	CHAR(5) may be 'India' but not 'Kerala'
CHARACTER VARYING(n)	VARCHAR(n)	Variable length character string up to length n (max n =2000).	VARCHAR(10) may be 'India' or 'Kerala'
CHARACTER LARGE OBJECT(n)	CLOB(n)	Variable length character string (usually large)	May be a book. n specified in Kbytes, Mbytes or Gbytes.
BINARY VARYING (n)	VARBINARY(n)	Variable length character string up to length n	n can be between 1 and 8000.
BINARY LARGE OBJECT(n)	BLOB(n)	Variable length binary string (usually large)	May be a photo. n specified in Kbits, Mbits or Gbits.

Figure 5.8 SQL string and binary data types

As shown above, quotes must be used when dealing with character strings. Comparison of character strings in SQL queries is explained later.

BLOBs may be used for large multimedia objects, for example, photos. CLOBs may be used for objects like a book. BLOBs as well as CLOBs may not be compared with others and may not be part of a relation's primary key and may not be used in DISTINCT, GROUP BY and ORDER BY clauses.

Date and Time Data Types

We now list the date and time data types in Fig. 5.9. In addition, data types TIMESTAMP and INTERVAL are also available.

Data Type	Description	Example	Comments
Date	YEAR, MONTH and DAY in the format YYYY-MM-DD. Less than comparison may be used.	2010-05-25 or 25-May-2010	Year varies from 0001 to 9999.
Time	HOUR, MINUTE and SECOND in the format HH:MM:SS. Less than comparison may be used.	22:14:37 (fraction of a second may also be represented)	Hour varies from 00 to 23.
TIMESTAP(n)	Used to specify time of an event in the format YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND	The format is YYYY-MM-DD-HH:MM-SS[.s.]	The length is 26.
INTERVAL	Used to represent time intervals like 9 days or 20 hours and 37 minutes.	Can be YEAR-MONTH or DAY-TIME.	

Figure 5.9 SQL date and time data types

Other complex data types have also been defined by SQL. These include user defined types (UDTs) and structures.

5.4 DATA MANIPULATION–SQL DATA RETRIEVAL

As noted above, the SELECT command is used for retrieving information from a database. The basic structure of the SELECT command is as given in Fig. 5.10.

```
SELECT something _of _interst
FROM table(s)
WHERE condition holds
```

Figure 5.10 The Basic SELECT Command in SQL

The SELECT clause specifies what column values are to be retrieved and is therefore equivalent to specifying a projection by listing one or more column names. There can often be confusion between the SELECT clause and the relational algebra operator called selection. The selection operator selects a number of rows from a relation given a condition that the rows satisfy while the SQL SELECT clause is similar to the relational algebra projection operator as it specifies the columns that need to be retrieved. A SELECT clause is mandatory in all SQL queries.

The FROM clause includes a list of one or more table names that are needed to answer the query. If more than one table is specified, a join operation must be specified. Having more than one table without specifying a join is rarely useful. A FROM clause is mandatory in all SQL queries.

The condition in the WHERE clause in an SQL query is optional. The clause provides qualification for the rows that are to be selected. If a WHERE clause is not specified, there is no qualification and the whole table is selected. Normally the WHERE clause is the heart of an SQL query. It specifies the condition(s) that identify the rows the user wishes to retrieve. It is therefore like the condition in the relational algebra selection

operator. The condition specified in the WHERE clause is either true or false for each row in the table. The rows for which the condition is true are selected and the rows for which the condition is false are rejected. A number of other optional clauses may follow the WHERE clause. These may be used to virtually group the rows in the table and to specify group conditions and to order the table retrieved if necessary. These are discussed later in this chapter.

The result of each SQL query is a table and may therefore be used like any other table. There is however one major difference between a base table and a table that is retrieved from an SQL query. The table retrieved by an SQL query may have duplicates while the rows in a base table are unique.

SQL is case insensitive although we will continue to use upper case for the SQL reserve words like SELECT, FROM and WHERE which could well be written as select, from and where. Also, it should be noted that all table names and column names in a query are one word and do not have spaces within them. The names must be unique. Usually each query ends with a semicolon but the user should check the documentation of the DBMS that is being used to ensure correct notation.

5.4.1 SQL Aliases

A table or a column may be given another name in a SQL query by using an alias. This can be helpful if there are long or complex table or column names which may be converted to short names using aliases, in particular in complex queries using several tables. Also, for some reasons, we may want to change the columns' names in the output from the names used in the base tables. SQL aliases may then be used. A column alias in the column list of the SELECT statement may be specified by using the AS keyword.

The syntax of table name aliases is as shown in Fig. 5.11.

```
SELECT column_name(s)
FROM table_name AS alias_name
```

Figure 5.11 Alias used in the FROM clause in SQL

The syntax for column names is as shown in Fig. 5.12.

```
SELECT column_name AS alias_name
FROM table _name
```

Figure 5.12 Alias used in the SELECT command in SQL

SQL column aliases are used to make the output of an SQL query easy to read and more meaningful. An example is shown in Fig. 5.13.

```
SELECT Teaml AS HomeTeam
FROM Match
```

Figure 5.13 Alias used in the SELECT command in SQL

A query like the one above may be used to change the output of the query. It will now have the heading *HomeTeam* instead of *Team*1.

We now present a number of examples to illustrate different forms of the SELECT command.

5.4.2 Simple Queries—Selecting Columns and Rows

We will first present some simple queries that involve only one table in the database. The simplest of such queries includes only a SELECT clause and a FROM clause.

Retrieving the Whole Table using*

• (*Q*1) Print the *Player* table.

Figure 5.14 shows how this query can be formulated in SQL.

When the WHERE clause is omitted, all the rows of the table are selected.

```
SELECT *
FROM Player
```

Figure 5.14 SQL for Query *Q*1

The asterisk following SELECT indicates that all columns of the table specified in the FROM clause are to be retrieved. We could have written all the column names but the wild card * provides a convenient way to print them all. If all columns are not required then we can formulate a query like the next one.

The result of this query is very similar to the table *Player* given in Table 5.2 at the beginning of this chapter although the ordering of the rows cannot be predicted. As noted in the last chapter, the relational model does not define an ordering of the rows and the ordering in a result is system dependent.

Using Projection to Retrieve Some Columns of All Rows from One Table

• (*Q*2) Find the IDs and first and last names of all players.

Figure 5.15 shows how this query can be formulated in SQL.

The above query is equivalent to specifying a projection operator in relational algebra to the table *Player* such that only columns *PlayerID, FName,* and *LName* are retrieved.

```
SELECT PlayerID, FName, LName
FROM Player
```

Figure 5.15 SQL for query *Q*2

The result of this query is presented in Table 5.5.

Retrieving All Information about Certain Rows from One Table

Often we may not want information about all the rows. Instead we need information only about some rows based on specific criteria. The criteria may be specified by using the WHERE clause.

• (*Q*3) Find all the information from table *Player* about players from India.

Figure 5.16 shows how this query can be formulated in SQL. Since we are only interested in players from India, the WHERE clause is used to specify the condition.

Table 5.5 Result of query *Q*2

PlayerID	*FName*	*LName*
89001	Sachin	Tendulkar
90001	Brian	Lara
95001	Ricky	Ponting
96001	Rahul	Dravid
96002	Herschelle	Gibbs
92001	Shane	Warne
95002	Shaun	Pollock
99003	Michael	Vaughan
92003	Inzamam	Ul-Huq
94004	Stephen	Fleming
93002	Heath	Streak
90002	Anil	Kumble
93003	Gary	Kirsten
95003	Jacques	Kallis
94002	Chaminda	Vaas
92002	Muthiah	Muralitharan
97004	Daniel	Vettori
25001	MS	Dhoni
23001	Yuvraj	Singh
96003	Saurav	Ganguly
99002	Adam	Gilchrist
24001	Andrew	Symonds
99001	Brett	Lee
91001	Sanath	Jayasuriya
21001	Virender	Sehwag
98001	Shahid	Afridi
98002	Harbhajan	Singh
27001	Praveen	Kumar
27002	Ishant	Sharma

Only those rows that have the *Country* column value equal to India are selected. All columns of these rows are presented to the user. Note that when an attribute value is text, the value in the WHERE clause is surrounded by single quotes.

```
SELECT *
FROM Player
WHERE Country = 'India'
```

Figure 5.16 SQL for query *Q*3

The result of this query is all the details of Indian players and is the relation given in Table 5.6.

Table 5.6 Result of query *Q*3

PlayerID	*LName*	*FName*	*Country*	*YBorn*	*BPlace*	*FTest*
89001	Tendulkar	Sachin	India	1973	Mumbai	1989
96001	Dravid	Rahul	India	1973	Indore	1996
90002	Kumble	Anil	India	1970	Bangalore	1990
25001	Dhoni	MS	India	1981	Ranchi	2005
23001	Singh	Yuvraj	India	1981	Chandigarh	2003
96003	Ganguly	Saurav	India	1972	Calcutta	1996
21001	Sehwag	Virender	India	1978	Delhi	2001
98002	Singh	Harbhajan	India	1980	Jalandhar	1998
27001	Kumar	Praveen	India	1986	Meerut	NULL
27002	Sharma	Ishant	India	1988	Delhi	2007

In the query posed in Fig. 5.16, it is possible to use "WHERE Country <> 'India'" if all players who are not from India were to be retrieved. Many other conditions may be used in the WHERE clause as discussed later.

• (*Q*4) Find all the information from table *Player* about players from India who were born after 1980.

Figure 5.17 shows how this query may be formulated in SQL.

In this query we are not only interested in players from India but also those born after 1980. The WHERE clause is used to specify both these conditions.

```
SELECT *
FROM Player
WHERE Country = 'India'
AND YBORN > 1980
```

Figure 5.17 SQL for query *Q*4

Only those rows that have the *Country* column value equal to India and players born after 1980 are selected.

The result of this query is all the details of Indian players who were born after 1980 and is the relation given in Table 5.7.

Table 5.7 Result of query *Q*4

PlayerID	*LName*	*FName*	*Country*	*YBorn*	*BPlace*	*FTest*
25001	Dhoni	MS	India	1981	Ranchi	2005
23001	Singh	Yuvraj	India	1981	Chandigarh	2003
27001	Kumar	Praveen	India	1986	Meerut	NULL
27002	Sharma	Ishant	India	1988	Delhi	2007

Retrieving Certain Columns of Certain Rows from One Table

• (*Q*5) Find details of matches that have been played in Australia.

```
SELECT Team1, Team2, Ground, Date
FROM Match
WHERE Team1 = 'Australia'
```

Figure 5.18 SQL for query *Q*5

First we note a weakness of our example data model. There is no way to obtain sufficient detail about each ODI match as we do not include a variety of useful information, for example, which ODI matches were played as part of the "2007-8 ODI Series between Australia and India". Also in the last few years many triangular ODI series have been held and some matches have been held in neutral venues like Dubai, Belfast or Kuala Lumpur. It is thus difficult to identify where each ODI match was played but we do so for the example database by assuming that the *Team*1 name is also the name of the country where the match is being played.

Figure 5.18 shows how this query may be formulated in SQL.

Again, a WHERE clause specifies the condition that *Team*1 is Australia.

One way of looking at the processing of the above query is to think of a pointer going down the table *Match* and at each row asking the question posed in the WHERE clause. If the answer is yes, the row being pointed at is selected, otherwise rejected. Since we have not included the name of the country where a match is being played in this example database, as noted above, we have to assume that if *Team*1 is Australia then the match is being played in Australia.

The result of this query providing details of matches played in Australia is given below in Table 5.8.

Table 5.8 Result of query *Q*5

Team1	*Team 2*	*Ground*	*Date*
Australia	India	Melbourne	10/2/2008
Australia	India	Sydney	2/3/2008
Australia	India	Brisbane	4/3/2008

• (*Q*6) List matches played in which India or Australia was *Team*1.

```
SELECT Team1, Team2, Ground, Date
FROM Match
WHERE Team1 = IN {'Australia','India'}
```

Figure 5.19 SQL for query *Q*6

The condition in this query is slightly more complex. Figure 5.19 shows how this condition can be formulated in SQL.

In this query also, match information is retrieved when *Team*1 column value is either Australia or India. Any number of values may be included in the list that follows IN.

The result of query *Q*6 above is given in Table 5.9.

Constants like 'Australia' or 'India' are often called a *literal tuple*. A literal tuple of course could have a number of values. For example, it could be

WHERE {2675, Australia, India, Melbourne, 10/2/2008, Team1} IN Match

Table 5.9 Result of query *Q6*

Team1	*Team2*	*Ground*	*Date*
India	England	Delhi	28/3/2006
India	Australia	Mohali	29/10/2006
India	West Indies	Nagpur	21/1/2007
India	West Indies	Vadodara	31/1/2007
India	Sri Lanka	Rajkot	11/2/2007
India	Australia	Mumbai	17/10/2007
India	Pakistan	Guwahati	5/11/2007
Australia	India	Melbourne	10/2/2008
India	Sri Lanka	Adelaide	19/2/2008
Australia	India	Sydney	2/3/2008
Australia	India	Brisbane	4/3/2008

The set of columns in the list before IN must match the list of columns in the list that follows IN for the set membership test carried out by IN to be legal (the list in the example above being a table).

```
SELECT Team1, Team2, Ground, Date
FROM Match
WHERE Team1 = 'Australia'
      OR Team1 = 'India'
```

Figure 5.20 SQL for query *Q6*

The above query in Fig. 5.19 is equivalent to the following SQL query shown in Fig. 5.20.

SQL allows some queries to be formulated in a number of ways. We will see further instances where a query can be formulated in two or more different ways.

The above query could use aliases if the headings in Table 5.9 are to be changed. For example,

```
SELECT Team1 AS HTeam, Team2 AS Visitors, Ground, Date
FROM Match
WHERE Team1 = 'Australia'
      OR Team1 = 'India'
```

Figure 5.21 SQL for query *Q6* using aliases for column names

Removing Duplicates from the Information Retrieved

• (*Q7*) Find IDs of all players that have bowled in an ODI match in the database.

Figure 5.22 shows how this query may be formulated in SQL. A WHERE clause is not required since we want IDs of all players in the table *Bowling*.

```
SELECT DISTINCT (PID)
FROM Bowling
```

Figure 5.22 SQL for query *Q7*

The above query is a projection of the table *Bowling*. Only the *PID* column has been selected and DISTINCT is used to remove duplicate values.

The result of query *Q*7 is given in Table 5.10.

Table 5.10 Result of query *Q*7

PID
99001
24001
23001
94002
92002
91001
23001
98002

DISTINCT may also be used when the query retrieves more than one column. The next query presents an example illustrating this fact.

- (*Q*8) Find names of teams and grounds where India has played an ODI match outside India.

We need to make a number of assumptions in formulating this query. We assume that if *Team*2 is India then India is playing outside India. We also assume that India could have played the same country at the same ground more than once (for example, Australia at Melbourne) and therefore we should remove the duplicates.

The query may now be formulated as in Fig. 5.23 in SQL.

```
SELECT DISTINCT (Team1, Ground)
FROM Match
WHERE Team2 = 'India'
```

Figure 5.23 SQL for query *Q*8

Obviously the list obtained by specifying DISTINCT *(Team1, Ground))* would be smaller than that obtained by specifying *(Team1, Ground)* without the DISTINCT clause although our database is so small that it had only one duplicate, playing Sri Lanka in Colombo.

The result of query *Q*8 is the teams and grounds on which India has played ODI matches outside India as given in Table 5.11.

Table 5.11 Result of query *Q*8 formulated in Fig. 5.23

Team1	*Ground*
Pakistan	Peshawar
Pakistan	Rawalpindi
West Indies	Kingston
Sri Lanka	Colombo
South Africa	Cape Town
England	Southampton
Australia	Melbourne
Australia	Sydney
Australia	Brisbane
Pakistan	Karachi

• (*Q*9) Find IDs of all players that batted in match 2755.

To formulate this query, we require a WHERE clause that specifies the condition for rows in the table *Batting* such that the rows are for match 2755. Figure 5.24 shows how this query may be formulated in SQL.

```
SELECT PID
FROM Batting
WHERE MatchID = '2755'
```

Figure 5.24 SQL for query *Q*9

This query also involves a selection as well as a projection. Note that the *MatchID* attribute is being treated as a string.

The result of this query is given in Table 5.12. Our small sample database has only seven batting records for match 2755.

Table 5.12 Result of query *Q*9

PID
23001
25001
91001
94002
92002
27001
98002

Sorting the Information Retrieved

- (*Q*10) Display a sorted list of ground names where Australia has played as *Team*1.

This is a query that uses a WHERE clause to specify the condition that Australia is *Team*1 and using ORDER BY to sort the result.

Fig. 5.25 shows how this query can be formulated in SQL.

```
SELECT Ground
FROM Match
WHERE Team1 = 'Australia'
ORDER BY Ground
```

Figure 5.25 SQL for query *Q*10

The list returned by the query in Fig. 5.25 would include duplicate names if more than one match had the same ground name (our example database is rather small and does not contain information on many ODI series played in a country). DISTINCT could be used to remove duplicates if required.

The above query produces Table 5.13 as the result:

Table 5.13 Result of query *Q*10

Ground
Adelaide
Brisbane
Melbourne
Sydney

The ORDER BY clause used in the query above returns results in ascending order (that is the default). Descending order may be specified by using

ORDER BY *Team1* DESC

The order ascending may be specified explicitly. When the ORDER BY clause is not present, the order of the result depends on the DBMS implementation as discussed earlier.

The ORDER BY clause may be used to order the result by more than one column. For example, if the ORDER BY clause in a query was ORDER BY *Ground, Date*, then the result will be ordered by *Ground* and the rows with the same *Ground* will be ordered by *Date*. The ordering may be ascending or descending and may be different for each of the fields on which the result is being sorted.

String Matching

SQL provides string pattern matching facilities using the LIKE command. LIKE is a powerful string matching operator that uses two special characters (called *wildcards*). These characters are underscore (_) and percent (%). They have special meanings; underscore represents any single character while percent represents any sequence of *n* characters including a sequence of no characters. For example, LIKE 'Del%' matches any string that starts with Del followed by any number of character or no character. LIKE 'Gup_' matches a four character string starting with Gup at the beginning. Other possibilities are LIKE '%ing%' (any string with at least three characters with ing in it) or LIKE '_ing_' (a five character string with 'ing' in the middle, for example Singh) or LIKE '%mes_' (a string of at least four characters with 'mes' followed by another character at the end, matching Ramesh or any word ending in, for example 'mesh').

It should be noted that LIKE is not always the same as equality of two strings. The test of equality first pads the smaller of the two strings if they are of unequal length by blanks and then compares them. LIKE does no padding.

There are a number of string operations available in SQL. These include the operations given in Fig. 5.26.

Operation	Meaning
UPPER (string)	Converts the string into uppercase
LOWER (string)	Converts the string into lowercse
INITCAP (string)	Converts the intial letter to uppercase
LENGTH (string)	Finds the string length
SUBSTR (string, n, m)	Get the substring starting at position n

Figure 5.26 Some SQL string operations

We now consider a simple query to illustrate the use of LIKE.

• (*Q*11) Find the names of all players whose last name starts with Sing.

In this query we need to use the LIKE command. The expression that follows LIKE must be a character string. The character percent (%) represents any sequence of characters including a sequence of no characters.

```
SELECT FName, LName
FROM Player
WHERE LName LIKE 'Sing%'
```

Figure 5.27 SQL for query *Q*21

Figure 5.27 shows how this query can be formulated in SQL.

This query produces Table 5.14 as result.

Table 5.14 Result of query *Q*11

FName	*LName*
Yuvraj	Singh
Harbhajan	Singh

In Fig. 5.27, the last statement may be changed to WHERE *LName* NOT LIKE 'Sing%' to find players whose name does not start with Sing. WHERE *LName* LIKE 'Sing_' would find only those names that have one character following 'Sing'.

Also LIKE '%ing%' would find any string that has the substring 'ing' in it.

SQL also provides more complex pattern matching facilities using SIMILAR but we do not discuss that SQL command here.

5.4.3 Different Forms of the WHERE Clause

So far we have only used the equality comparison in the WHERE clause. Other relationships are available. These include greater than (>), less than (<), less than equal (≤), greater than equal (≥) and not equal (<>).

We show below a number of different forms of the WHERE clause.

Some of the common forms are as follows:

- WHERE *C*1 AND *C*2—Each selected row must satisfy both the conditions *C*1 and *C*2.

- WHERE *C*1 OR *C*2—Each selected row must satisfy either the condition *C*1 or the condition *C*2 or both conditions *C*1 and *C*2.
  AND and OR operators may be combined, for example, in a condition like *A* AND *B* OR *C*. In such compound conditions the AND operation is done first. Parentheses could be used to ensure that the intent is clear, for example, (*A* AND *B*) OR (*C*). A more complex example may help: *A* AND *B* OR *C* AND *D* OR *E* AND *F* would be processed as (*A* AND *B*) OR (*C* AND *D*) OR (*E* AND *F*) and will be true if any one or more of the conditions *A* AND *B*, *C* AND *D*, *E* AND *F* is true.
- WHERE NOT *C*1 AND *C*2—Each selected row must satisfy condition *C*2 but must not satisfy the condition *C*1.
- WHERE *A* IN—Each selected row must have the value of *A* in the list that follows IN. NOT IN may be used to select all that are not in the list that follows NOT IN.
- WHERE *A* operator ANY—Each selected row must satisfy the condition for any of the list that follows ANY. Any of the following operators may be used in such a clause: greater than (>), less than (<), less than equal (≤), greater than equal (≥) and not equal (<>).
- WHERE *A* operator ALL—Each selected row now must satisfy the condition for each of the list that follows ALL. Any of the following operators may be used in such a clause: greater than (>), less than (<), less than equal (≤), greater than equal (≥) and not equal (<>).
- WHERE *A* BETWEEN *x* AND *y*—Each selected row must have value of *A* between *x* and *y* including values *x* and *y*. NOT BETWEEN may be used to find rows that are outside the range (*x*, *y*).
- WHERE *A* LIKE *x*—Each selected row must have a value of *A* that satisfies the string matching condition specified in *x*. The expression that follows LIKE must be a character string and enclosed in apostrophes, for example 'Delhi'.
- WHERE *A* IS NULL—Each selected row must satisfy the condition that *A* is NULL, that is, a value for *A* has not been specified.
- WHERE EXISTS—Each selected row be such that the subquery following EXISTS does return something, that is, the subquery does not return a NULL result.
- WHERE NOT EXISTS—Each selected row be such that the subquery following NOT EXISTS does not return anything, that is, the subquery returns a NULL result.

5.4.4 Queries Involving More than One Table—Using Joins and Subqueries

Many relational database queries require more than one table because each table in the database is about a single entity. For example, player information including player name is in table *Player* while batting and bowling performances are provided in tables *Batting* and *Bowling*. Therefore any time we wish to retrieve information about the batting or bowling performance of a player by using the player's name (and not the *Player ID*) we have to use two tables as shown in the examples that follow.

There are two possible approaches to using more than one table. One approach is to use a join. The other approach is to use one or more subqueries as shown in the next few examples. When using a join, most queries below only consider an inner equi-join.

Subqueries

A subquery is a query that is a part of another query. Subqueries may also include subqueries and this may continue to any number of levels. The queries presented below use subqueries in the WHERE clause but subqueries may also be used in the FROM clause instead of a table name as we will show later.

When a subquery is used in the WHERE clause, it normally tests for two types of conditions. The first type of condition usually tests a condition on the result from the subquery. For example, it may use formulation like "PID IN <subquery>" or "PID NOT IN <subquery>". The second type of condition is like "EXISTS <subquery>" or "NOT EXISTS <subquery>" to test that a non-null result is returned by the subquery or a null result is returned. We illustrate these conditions now.

• (*Q*12) Find Match IDs of all matches in the database in which Tendulkar batted.

As noted above, this query needs to use two tables as the player name is given only in the table *Player* and the batting performance is given only in the table *Batting*. We will use a subquery to obtain the *PID* of Tendulkar and use that information to obtain *Match IDs* of matches that he has played in.

This query is a simple use of a subquery. The subquery only returns a single constant which is compared with another value in the WHERE clause. If the comparison's result is true then the row in the outside query is selected otherwise it is not.

In a query like this in which the subquery returns only one value, *PID* IN could be replaced by *PID* = but if we are not certain that the result of the subquery will be a constant then it is best to use the IN operator.

Figure 5.28 shows how this query may be formulated in SQL.

```
SELECT MatchID
FROM Batting
WHERE PID IN
      (SELECT PlayerID
      FROM Player
      WHERE Lname = 'Tendulker')
```

Figure 5.28 SQL for query *Q*12

The above formulation of the query has a subquery (or a *nested query*) associated with it. The subquery feature is very useful and as noted earlier, a subquery may itself have a subquery within it and queries may be nested to any level.

The result of the above query is given in Table 5.15. Due to very limited data in our database only one ODI is found.

Table 5.15 Result of query *Q*12

MatchID
2689

It may be appropriate to briefly discuss how such a query is processed. The subquery is processed first resulting in a new table (in the present case a single column table with only one row). We have of course assumed that the last name Tendulkar is unique in the database. This is unlikely to be true if the last name of the player was Singh or some other common name. This resulting table is then used in the outer query in which the WHERE clause becomes true if the *PID* value of the player in the *Batting* table is in the set of *PIDs* that is returned by the subquery. Of course, in this case there is only one member in the set returned by the subquery. The IN operation in the WHERE clause tests for set membership. The clause may also use construct NOT IN instead of IN when appropriate.

Another format for the IN condition is illustrated in the following query.

• (*Q*13) Find the match information of all matches in which Dhoni has batted.

Now we show how a subquery may itself use a subquery in its WHERE clause.

This query must involve the three tables *Match*, *Batting* and *Player* as the match information is in table *Match*, the player names are in table *Player* and batting information is available only in table *Batting*. In this query we use a subquery that itself uses a subquery within it. The innermost subquery obtains the *Player ID* of Dhoni, then the next level subquery is used to obtain the *Match ID* of matches he has batted in and then use those *Match IDs* in the outermost query to obtain information about the matches.

```
SELECT MatchID, Team1, Team2, Ground, Date
FROM Match
WHERE MatchID IN
        (SELECT MatchID
        FROM Batting
        WHERE PID IN
                (SELECT PlayerID
                FROM Player
                WHERE LName = 'Dhoni'))
```

Figure 5.29 SQL for query *Q*13

Figure 5.29 shows how this query may be formulated in SQL.

This query results in the last subquery (or innermost subquery) being processed first. This subquery involves a selection of table *Player* which results in a table with only one row that is the *Player ID* of Dhoni. This result is then used to find a list of *Match IDs* from table *Batting* in which Dhoni has batted. These *Match IDs* are then used to find match information from the table *Match*.

Note that each subquery is executed only once and its result is used as a constant or a relation in the next outer query.

The result of the query is only two matches in our example database. The results are presented in Table 5.16.

Table 5.16 Result of query *Q*13

MatchId	*Team1*	*Team2*	*Ground*	*Date*
2689	Australia	India	Brisbane	4/3/2008
2755	Sri Lanka	India	Colombo	27/8/2008

We now look at how these queries can be formulated using a join. Additional queries that use subqueries are presented following the discussion of join.

Join Queries

It is possible to combine rows from two tables (or any number of tables) listed in the FROM clause by specifying the join condition(s) in the WHERE clause. We illustrate join by reformulating the last two queries.

```
SELECT MatchID
FROM Batting, Player
WHERE PID = PlayerID
AND LName = 'Tendulkar'
```

Figure 5.30 SQL query using a join for query *Q*12

The two queries, *Q*12 and *Q*13, involving subqueries may be reformulated using the join operator as given in Figs 5.30 and 5.31 respectively.

Reformulation of query *Q*13 is given in Fig. 5.31.

It is worth discussing how the join operator is used in the above queries. When the FROM clause specifies more than one table name, the query processor (conceptually) forms a Cartesian product of those tables and

applies the join condition specified in the WHERE clause. Any other conditions in the WHERE clause are then also applied. In practice, the algorithm used is not that simple (and inefficient). Fast algorithms, to be discussed in Chapter 8, are used for computing joins.

```
SELECT Team1, Team2, Ground, Date
FROM Match, Batting, Player
WHERE Match.MatchID = Batting.MatchID
AND PlayerID = PID
AND LName = 'Dhoni'
```

Figure 5.31 SQL query using a join for query *Q*13

In both joins in Figs 5.30 and 5.31, the join condition is an equality condition and therefore equi-join is used in both SQL formulations.

We should also note that, in queries like those in Figs 5.30 and 5.31, it may sometimes be necessary to qualify some of the column names if the same name is used in more than one table in the FROM clause. For example, if the column *MatchID* was called by the same name in *Match* and *Batting* then one of the equalities in the WHERE clause will have to be *Match.MatchID = Batting.MatchID*. Such qualifications are needed only when there is a chance of ambiguity.

As noted earlier, it is possible to give aliases to some table names (for example, *Batting* may be given an alias *b1*) when names are long or more than one instance of the same table is used in a query.

• (*Q*14) Find the player ID of players who have made a century in each of the ODI matches 2755 and 2689.

The table *Batting* has a separate row for every batsman in each innings. We can therefore either use a subquery or use a join. We use a join of table *Batting* with itself.

Figure 5.32 shows how this query may be formulated in SQL.

There are no players that have scored a century in our database and therefore the result is null.

```
SELECT PID
FROM Batting b1, Batting b2
WHERE b1. PID = b2. PID
AND b1. MatchID = 2755
AND b2. MatchID = 2689
AND b1. NRuns > 99
AND b2. NRuns > 99
```

Figure 5.32 SQL for query *Q*14

This query is presented to illustrate that sometimes we may need to join a table to itself. In this query we have joined the table *Batting* to itself such that the join column is *PID.* In addition we have ensured that the first score is from the Match 2755 and the second score is from Match 2689 and both scores are 100 or more. We then retrieve the *Player IDs*.

The resulting join table in the query above would have rows such that each row has two scores of a player from the two ODI matches and both scores are 100 or more.

The query formulation in Fig. 5.32 also shows, as discussed earlier, that a table may be given an alias if necessary. In this case one instance of table *Batting* is given the name *b1* while the other is given the name *b2*.

More Subqueries

• (*Q*15) Find IDs of players that have both bowled and batted in the ODI match 2689.

This query needs to access both the tables *Batting* and *Bowling*. We will use a subquery formulation in which the subquery finds player IDs of those players that bowled in match 2689. The outer subquery then finds player IDs of those players that batted in match 2689 if they are also in the list of players that bowled in that match. Note that the inner subquery is executed only once.

Figure 5.33 shows how this query may be formulated in SQL.

```
SELECT PID
FROM Batting
WHERE MatchID = '2689'
AND PID IN
        (SELECT PID
        FROM Bowling
        WHERE MatchID = '2689')
```

Figure 5.33 SQL for query *Q*15

As noted above, the subquery finds the player IDs of those players that have bowled in Match 2689 while the outside subquery finds from this list of bowler player IDs those players that have also batted.

The result of this query is given in Table 5.17.

Table 5.17 Result of query *Q*15

PID
99001
24001
23001
98002

This query could also be formulated as an intersection of the two queries, one that finds the player IDs of all the bowlers and the other that finds the player IDs of all the batsmen. The intersection then gives the player IDs of those players that are in both player ID lists.

• (*Q*16) Find IDs of players that have either bowled or batted (or did both) in the ODI match 2689.

This query is similar to the last query, again requiring both tables *Batting* and *Bowling*. In this case, a subquery formulation is presented in which the subquery finds player IDs of all those players that bowled in match 2689. The outer query then finds player IDs of all those players that batted in match 2689 or if they are on the list of players that bowled in that match. Note that the inner subquery is executed only once.

Figure 5.34 shows how this query may be formulated in SQL.

```
SELECT PID
FROM Batting
WHERE MatchID = '2689'
OR PID IN
        (SELECT PID
        FROM Bowling
        WHERE MatchID = '2689')
```

Figure 5.34 Incorrect SQL for query *Q*16

Using UNION, INTERSECT and MINUS (or EXCEPT)

The query *Q*16 may also be written using the UNION operator as given in Fig. 5.35.

```
SELECT PID
FROM Batting
WHERE MatchID = '2689'

UNION

(SELECT PID
FROM Bowling
WHERE MatchID = '2689')
```

Figure 5.35 SQL for query *Q*16

The result of this query is given in Table 5.18.

Table 5.18 Result of query *Q*16

PID
89001
98002
23001
25001
99002
95001
24001
99001
27001

Figure 5.35 SQL formulation of the query would produce duplicates in the result because some players would appear in both the lists, the list of bowlers for match 2689 and the list of batsmen for match 2689. We should note that when UNION is used the data type of the columns in the first SELECT command must match the data type of the corresponding column in the second SELECT command. In the above query there is no difficulty as we are retrieving only one column which obviously matches.

Can this query be formulated in another way; perhaps using the table match in addition to tables *Batting* and *Bowling?*

What would the query become if UNION was replaced by INTERSECT or MINUS in Fig. 5.35?

Let us first consider replacing UNION with INTERSECT as shown in Fig. 5.36.

```
SELECT PID
FROM Batting
WHERE MatchID = '2689'

INTERSECT

(SELECT PID
FROM Bowling
WHERE MatchID = '2689')
```

Figure 5.36 SQL query using INTERSECT

Set intersection of any two sets results in those members of the two sets that are common to both sets. Therefore the query in Fig. 5.36 results in giving us a list of *PIDs* for players who have batted as well as bowled in Match 2689. The result therefore is the same as Table 5.17.

Let us now use EXCEPT, which is the same as MINUS, in Fig. 5.37.

```
SELECT PID
FROM Batting
WHERE MatchID = '2689'

EXCECT

SELECT PID
FROM Bowling
WHERE MatchID = '2689'
```

Figure 5.37 SQL query using EXCEPT

Set difference of any two sets, results in those members of the first set that are not in the second set. Therefore the query in Fig. 5.37 results in giving us a list of *PIDs* for players who batted but did not bowl in Match 2689. This is reformulated in the next query and its result is the same as in Table 5.19.

• (*Q*17) Find IDs of players that have batted in match 2689 but have not bowled.

Figure 5.38 shows another way this query may be formulated in SQL. Given the difficulty in formulating the query *Q*16, we must formulate this one carefully.

This query is equivalent to the query in Fig. 5.37 using EXCEPT to select those players that are selected by the outer query (i.e., those that have batted in the match) as long as they are not in the list of players that are returned by the subquery, that is, the list of players that bowled in that match. This has been done by using the "*PID* NOT IN" in the WHERE clause. Note that the subquery is only executed once.

```
SELECT PID
FROM Batting
WHERE MatchID = '2689'
AND PID NOT IN
      (SELECT PID
      FROM Bowling
      WHERE MatchID = '2689')
```

Figure 5.38 SQL for query *Q*17

The result of this query is given in Table 5.19.

Table 5.19 Result of query *Q*17

PID
89001
25001
99002
95001
27001

5.4.5 Using Subquery in the FROM Clause

A subquery may also be used in the FROM clause. We illustrate this use by an example given in Fig. 5.39.

• (*Q*18) Find the *Match IDs* of matches in which Sachin Tendulkar has played.

This subquery in Fig. 5.39 uses a subquery in the FROM clause which returns a table as its result and is therefore treated like a virtual table. In this particular case, the table returned has only one element which is the *PlayerID* of Sachin Tendulkar. This subquery table must be given a name which is ST in this query. We can now use this name in the WHERE clause.

```
SELECT MatchID
FROM Batting, (SELECT PlayerID
                FROM Player
                WHERE LName = 'Tendulkar') ST
WHERE PID = ST.PlayerID
```

Figure 5.39 Using a subquery in the FROM Clause

This use of subquery in the FROM clause is helpful in answering a variety of queries that use multiple aggregate functions as we will show later in this chapter.

We are now ready to illustrate the use of different forms of the WHERE clause in example queries that follow.

• (*Q*19) Find Match IDs of those matches in which player 27001 bats and makes more runs than he made at every match he played at Brisbane.

To formulate this query we first find all the scores of the player 27001 when he batted in Brisbane. This requires two subqueries, the innermost subquery and the middle subquery, since batting performance is given in table *Batting* and the name of the ground is available only in the table *Match*. Once the Brisbane scores are available, the outermost query finds Match IDs of matches where he scored more.

Figure 5.40 shows how this query may be formulated in SQL.

As noted earlier, when ALL is used in the WHERE clause as in Fig. 5.40 above, the value in the outer query must satisfy the specified condition with all the rows that are returned by the subquery. In this particular query the outermost query condition *NRuns* > ALL would be true only if the value of attribute *NRuns* is larger than the largest value of *NRuns* returned by the middle subquery.

It should also be noted that the outermost query will scan the whole table *Batting* including the matches played in Brisbane but none of the Brisbane scores would meet the condition.

```
SELECT MatchID
FROM Batting
WHERE PID = '27001'
AND NRuns > ALL
        (SELECT NRuns
        FROM Batting
        WHERE PID = '27001'
        AND MatchID IN
                (SELECT MatchID
                FROM Match
                WHERE Ground = 'Brisbane'))
```

Figure 5.40 SQL for query *Q*19

The result of this query is given in Table 5.20.

• (*Q*20) Find IDs and scores of players who scored less than 75 but more than 50 in Colombo.

Once again we need to use a subquery because the information about the grounds is given in table *Match* while the batting information is in *Batting*. We will use the clause WHERE something BETWEEN *x* AND *y* although we could also use two WHERE conditions such as WHERE something ≥ *x* AND something ≤ *y*.

Table 5.20 Result of query *Q*19

MatchID
2755

Figure 5.41 shows how this query may be formulated in SQL.

```
SELECT PID, NRuns
FORM Batting
WHERE NRuns BETWEEN 51 AND 74
AND MatchID IN
        (SELECT MatchID
        FROM Match
        WHERE Ground = 'Colombo')
```

Figure 5.41 SQL for query *Q*20

Note that we are looking for scores between 51 and 74 and not between 50 and 75. This is because (A BETWEEN x AND y) has been defined to mean $A \geq x$ and $A \leq y$.

The result of this query is given in Table 5.21.

Table 5.21 Result of query *Q*19

PID	*NRuns*
25001	71
91001	60

• (*Q*21) Find the player IDs of those players whose date of first test match (*FTest*) is not given in the database.

As noted earlier, SQL provides a special command for checking if some information is missing as shown in Fig. 5.42 below.

```
SELECT PlayerID
FROM Player
WHERE FTest IS NULL
```

Figure 5.42 SQL for query *Q*21

It should be noted that the column value being NULL is very different than it being zero. The value is NULL only if it has been defined to be NULL.

The result of this query is given in Table 5.22.

Table 5.22 Result of query *Q*21

PlayerID
27001

5.4.6 Correlated Subqueries and Complex Queries

A number of queries above have used subqueries. The subqueries used were needed to be evaluated only once. Such subqueries are called *noncorrelated subqueries*. There is another type of subqueries that are more complex and are called *correlated subqueries.* In these subqueries the value retrieved by the subquery depends on a variable (or variables) that the subquery receives from the outer query. A correlated subquery thus cannot be evaluated once and for all since the outer query and the subquery are related. It must be evaluated repeatedly; once for each value of the variable received from the outer query.

The correlated subqueries are most often used to check for existence or absence of matching rows in the parent table and the related table in the subquery. A correlated subquery therefore always refers to one or more columns from a table in the outer query. We illustrate such queries in examples ahead.

• (*Q*22) Find the names of players who batted in match 2689.

The query *Q*8 required finding the player IDs of players who batted in a match. This query is a little bit different since in this query we wish to find the names (not player IDs) of players who have batted. We now need to use the two tables *Player* and *Batting* because player names are available only in the table *Player* while batting information is given in table *Batting*. In this query we look at each player in the table *Player* and then check, using a subquery, if there is a batting record for him in the table *Batting*.

```
SELECT FName, LName
FORM Player
WHERE EXISTS
        (SELECT *
        FROM Batting
        WHERE PlayerID = PID
         AND MatchID = '2689')
```

Figure 5.43 SQL for query *Q*22

Figure 5.43 shows how this query may be formulated in SQL.

The WHERE EXISTS clause returns true if the subquery following the EXISTS returns a non-null result. Similarly a WHERE NOT EXISTS clause returns true only if the subquery following the NOT EXISTS returns nothing, that is, a null table.

This query produces Table 5.23 as result.

Table 5.23 Result of query *Q*22

FName	*LName*
Sachin	Tendulkar
M. S.	Dhoni
Yuvraj	Singh
Adam	Gilchrist
Andrew	Symonds
Brett	Lee
Praveen	Kumar
Ricky	Ponting
Harbhajan	Singh

Again, the above query can be formulated in other ways. One of these formulations uses a join. Another uses a different subquery. Using the join, we obtain the SQL query given in Fig. 5.44.

The join table would have rows only for players that have batted in match 2689 since a selection has been applied after the join.

```
SELECT FName, LName
FORM Player, Batting
WHERE PlayerID = PID
AND MatchID = '2689'
```

Figure 5.44 SQL for query *Q*22

• (*Q*23) From the example database, find the player IDs of players who have scored more than 30 in every ODI match that they have batted.

SQL provides no direct way to check that for a given player all his batting or bowling performances satisfy some condition like all scores being above 30 but we can obtain the same information by reformulating the question. Instead of asking for players that have scored more than 30 every time they have batted, we ask for players that have batted but never scored less than 31.

Reformulating the question makes the query somewhat complex. The SQL query below formulates the query as "find the *Player IDs* of players that have no score of less than 31 in any ODI innings that they have batted"!

```
SELECT PID AS PlayerID
FORM Batting bl
WHERE NOT EXISTS
        (SELECT *
        FROM Batting b2
        WHERE b1.PID = b2.PID
        AND NRuns < 31)
```

Figure 5.45 SQL for query *Q*23

Figure 5.45 shows how this query may be formulated in SQL.

Note that the subquery in Fig. 5.45 cannot be executed only once since it is looking for rows in batting that satisfy the condition *NRuns* < 31 for each player in the table *Batting* whose *PID* has been supplied to the subquery by the outside query. The subquery therefore must be evaluated repeatedly; once for each value of the variable *PID* received by the subquery. The query is therefore using a correlated subquery.

Note that the condition in the WHERE clause will be true only if the subquery returns a null result and the subquery will return a null result only if the player has no score which is less than 31.

Note that we have used aliases in Fig. 5.28 to label two instances of *Batting* as *b1* and *b2*. Also the heading of the query output has been changed to *PlayerID* by using an alias in the SELECT clause.

The result of this query in presented in Table 5.24.

Table 5.24 Result of query Q23

PlayerID
25001
80001

It is worth considering why we used the table *Batting* in the outside query. Should we have used the table *Player* in it? The reason for using the table *Batting* in the outside query is that we want to first find a player that has batted and then check if the player has ever scored less than 31. We do not need to check a player that has never batted according to our example database.

Some further explanation might be helpful. The outer query looks at each row of the table *Batting* and then checks if the player qualifies. The checking is done in the subquery which checks if the players in Table *b1* have ever scored below 31.

We also wish to note that the above query formulation has a weakness that an alert reader would have already identified. The above query would retrieve *PID* of a player that qualifies for each value of *NRuns* that is greater than 30 that the player has. Therefore if a player has batted five times and has scored more than 30 every time, his ID will be retrieved five times in the result. DISTINCT may be used to remove such duplicates. Another approach would be to use the table *Player* instead of table *Batting b1* in the outside query but this leads to another problem that we discussed earlier.

• (*Q*24) Find the names of all players that have batted in all the ODI matches in Melbourne that are in the database.

This is a complex query that uses a correlated subquery because of the nature of SQL as it does not include the universal quantifier (*forall*) that is available in relational calculus. Once again we need to reformulate the question we are asking and this time it becomes quite ugly with a double negative. The reformulated question is "Find the names of players for whom there is no ODI match played in Melbourne that they have not batted "! The SQL query that solves it is shown in Fig. 5.46.

```
SELECT FName, LName
FORM Player
WHERE NOT EXISTS
        (SELECT *
        FROM Match
        WHERE Ground = 'Melbourne'
        AND NOT  EXISTS
                (SELECT *
                FROM Batting
                WHERE PID = PlayerID
                 AND (Batting.MatchID = Match.MatchID))
```

Figure 5.46 SQL for query *Q*24

This nested query consists of three components: the outermost query, the middle subquery and the innermost (or the last) subquery. Conceptually, the outermost part of the query starts by looking at each row in the table *Player* and for that player under consideration evaluates whether the NOT EXISTS clause is true. The NOT EXISTS clause will be true only if nothing is returned by the middle subquery. The middle subquery will return a null table only if for each of the matches played in Melbourne, the NOT EXISTS clause in it is false. Of course, the NOT EXISTS clause in it will be false only if the innermost subquery returns a non-null result. That is, there is no row in *Batting* for the player who has played in Melbourne that the middle subquery is considering for the player of interest to the outermost query.

Although the middle subquery does not use any variable from the outermost query, the last two lines in the innermost subquery use variables from both the outer queries. It uses *PlayerID* from the *Player* table from the outermost part of the query and *MatchID* from the *Match* table in the middle subquery. The query therefore includes a correlated subquery and the query is quite complex.

The scoping rules of the names used in a nested query needs to be understood. For example, in Fig. 5.44, the innermost query uses the table *Batting* and therefore all attribute names of that table may be used within that subquery. Attribute names from the outer subqueries may only be used if those names are unique or they are qualified by the name of the table.

SQL does not have a universal quantifier (*forall*) but does have an existential quantifier (*there exists*). As considered in the last chapter, the universal quantifier may be expressed in terms of the existential quantifier. The query formulation above implements a universal quantifier using the existential quantifier. Effectively, for each player, the query finds out if there exists a match in Melbourne in which the player has not batted. If such a match exists then the player is not selected for inclusion in the result.

Since our example database is very small, the result of this query is null. A larger database is presented in the Appendix at the end of this chapter.

More correlated and complex queries are presented after we have discussed the aggregate functions and the GROUP BY and HAVING commands.

5.4.7 Using SQL Aggregate Functions

SQL provides a number of built-in functions to carry out computations or summarizations. Some arithmetic can be carried out in the SELECT clause of an SQL query as well but it is rather limited and is not be discussed here.

The built-in functions are presented in Fig. 5.47.

As the names imply, the AVG function is for computing the average for a column that is of numeric type ignoring the NULL values, COUNT counts the occurrences of rows, MIN finds the smallest value ignoring the NULL values, MAX finds the largest value ignoring the NULL values, and SUM computes the sum. AVG, SUM and COUNT may also be used as COUNT (DISTINCT *A*) to count only unique values, AVG (DISTINCT *A*) to compute average of only unique values and finally SUM (DISTINCT *A*) to sum only unique values.

AVG
COUNT
MIN
MAC
SUM

Figure 5.47 SQL aggregate functions

Some systems allow additional functions, for example, STDDEV for computing standard deviation and VARIANCE for computing variance. These functions also ignore rows with NULL values.

We consider a number of queries that use these functions.

• (*Q*25) Find the number of players that bowled in the ODI match 2689.

```
SELECT COUNT (*) AS NBowlers
FROM Bowling
WHERE MatchID = '2689'
```

Figure 5.48 SQL for query *Q*25

Figure 5.48 shows how this query may be formulated in SQL.

The above query uses the function COUNT to count the number of rows that satisfy the condition in the WHERE clause. We have again used an alias for the heading of the query output.

Table 5.25 Result of query *Q*25

NBowlers
4

The result of the query is presented in Table 5.25.

• (*Q*26) Find the average batting score of all the players that batted in the ODI match 2689.

Figure 5.49 shows how this query may be formulated in SQL.

```
SELECT AVG(NRuns) AS AveRuns_2689
FROM Batting
WHERE MatchID = '2689'
```

Figure 5.49 SQL for query Q26

This query uses the function AVG to compute the average of the values of the column *NRuns* for all rows that satisfy the condition specified.

Table 5.26 Result of query *Q*26

AveRuns_2689
25.2

The result of the query is presented in Table 5.26.

• *Q*(27) Find the youngest player in the database.

We do not have the player's age as an attribute in our *Player* table. The only attribute related to age is the player's year of birth (*YBorn*) in the table *Player*. To find the youngest player we will find the player who was born last according to *YBorn*. To find the player born last, we need to find the player, whose year of birth is the largest, assuming this attribute is an integer.

```
SELECT LName AS Youngest_Player
FROM Player
WHERE YBorn =
      (SELECT MAX(YBorn)
      FROM Player)
```

Figure 5.50 SQL for query *Q*27

Figure 5.50 shows how this query may be formulated in SQL.

This is not a correlated query since the subquery does not use a variable from the outer query and is executed only once.

The result of the query *Q*27 is presented in Table 5.27.

Table 5.27 Result of query *Q*27

Youngest_Player
Sharma

Ishant Sharma is the youngest player having been born in 1988.

We note that we have used the equality comparison "YBorn =" just outside the subquery. Technically this is comparing an atomic attribute value of *YBorn* with the result of a subquery which only returns a table as the result, like all queries. In this query however it is clear that the result of the subquery is an aggregation MAX and therefore an atomic value. Even if more than one row has the same MAX value, only a single value will be returned which may be compared with an attribute value.

When we need to find the row with an attribute value that is the largest or smallest, the functions MAX and MIN are used but SQL does not allow us to formulate the query in a straightforward manner as that given in Fig. 5.51.

```
SELECT LName, MAX(YBorn)
FROM Player
```

Figure 5.51 An illegal formulation of query *Q*27

This SQL query is illegal although some systems will produce an output that will include all values of *LName* combined with the value of MAX(*YBorn*). The reason it is illegal is that when an aggregate operation is used in the SELECT clause then only aggregation operations may be used except when the GROUP BY clause is used. When the system finds the value of MAX(YBorn), it only looks at the *YBorn* column and values of other attributes corresponding to rows with the MAX value are then not available. We therefore must first find what the MAX(*YBorn*) value is and then find the row which has the attribute value equal to that MAX value.

Further queries using the built-in functions are presented after we have considered a mechanism for grouping a number of rows having the same value of one or more specified column(s).

5.4.8 Using the GROUP BY and HAVING Clauses

So far we have considered relatively simple queries that use aggregation functions like MAX, MIN, AVG and SUM. These functions are much more useful when used with GROUP BY and HAVING clauses which divide a table into virtual tables and apply qualifications on those virtual tables.

GROUP BY and HAVING clauses allow us to consider groups of records together that have some common characteristics (for example, players from the same country) and compare the groups in some way or extract information by aggregating data from each group. The group comparisons also are usually based on using an aggregate function.

```
SELECT something_of_interest
FROM table(s)
WHERE condition_holds
GROUP BY column_list
HAVING group_condition
```

Figure 5.52 The basic SELECT command in using GROUP BY

The general form of a GROUP BY query is as shown in Fig. 5.52.

The SELECT clause must include column names that are in the GROUP BY column list (other column names are not allowed in the SELECT clause) and aggregate functions applied to those columns. These results of aggregate functions should normally be given a column name for output. The HAVING clause must

be such that the condition can be easily checked, for example, AVG(*NRuns*) > 50. Note that AVG(*NRUNS*) will always have one value for each group and therefore the comparison is straightforward. We cannot use HAVING *NRuns* > 50 because each group has many values of *NRuns*.

The presence of the WHERE clause in the above query should be noted. This clause applies to the table before GROUP BY and HAVING and therefore results in a pre-selection of rows.

As noted earlier, GROUP BY allows a table to be divided into several virtual tables followed by use of any one of the following common aggregations:

- Count rows in each virtual group.
- Obtain the sum of values from a single column in each virtual group.
- Calculate average values from a column by choosing all the rows from each virtual group.
- Find maximum and minimum values from a column for each virtual group.

The procedure in executing a GROUP BY and HAVING clause works as follows:

- *All the rows that satisfy the WHERE clause (if given) are selected.*
- *All the rows are now grouped virtually according to the GROUP BY criterion.*
- *The HAVING clause including aggregation is applied to each virtual group.*
- *Groups satisfying the HAVING clause are selected.*
- *Values for the columns in the SELECT clause and aggregations are now retrieved.*

By default all the rows in a table are treated as one group if a GROUP BY is not specified. As noted earlier, WHERE may be used to pre-exclude rows before dividing the table and using the GROUP BY and HAVING clauses. GROUP BY must have one or more column names.

• (*Q*28) Find the number of players in the database from each country.

To find the number of players from each country, we need to partition the *Player* table to create virtual subtables for players from each country. Once the table has been so partitioned, we can count them.

```
SELECT Country, COUNT(*) AS NPlayers
FROM Player
GROUP BY Country
```

Figure 5.53 SQL for query *Q*28

Figure 5.53 shows how this query may be formulated in SQL.

The GROUP BY clause groups the rows by the specified column *Country* and the function COUNT then counts the number of rows in each virtual group. Note that the SELECT statement includes the column name *Country* which is the column which is used in the GROUP BY.

The result of the query is given in Table 5.28.

Table 5.28 Result of query *Q*28

Country	*NPlayers*
South Africa	4
Australia	5
England	1
Pakistan	2
New Zealand	2
Zimbabwe	1
India	10
Sri Lanka	3
West Indies	1

It should be clear that the above table could not be obtained without the GROUP BY clause which has many uses in formulating SQL queries. If the GROUP BY command was not available, we would need to find the number of players from each country by formulating a separate query for each country.

In the output we used an alias '*NPlayers*' for Count(*) because aggregate functions in the SELECT list do not have column names.

• (*Q*29) Find the batting average of each player in the database. Present the *PID* of each player.

To find the batting average of each player, we need to partition the table to create virtual subtables for each player. Once the table has been so partitioned, we can find the average of each player.

```
SELECT PID, AVE(NRuns) AS Ave
FROM Batting
GROUP BY PID
```

Figure 5.54 SQL for query Q29

Figure 5.54 shows how this query may be formulated in SQL.

The GROUP BY clause groups the rows by the specified column *PID* and the function AVE then computes the average of *NRuns* in each virtual group. Note that the SELECT statement includes the column name *PID* which is the column which is used in the GROUP BY.

The result of the above query is given in Table 5.29.

Table 5.29 Result of query Q29

PID	*Ave*
23001	19
24001	42
25001	54
27001	5
89001	91
91001	60
92002	1
94002	17
95001	1
98002	3
99001	7
99002	2

• (*Q*30) Find the batting average of each player from India in the database. Present the first name of each player.

To find the batting average of each Indian player, we need to join the tables *Player* and *Batting* and then select the rows for Indian players. The resulting table is now partitioned to create virtual subtables for each player. Once the table has been so partitioned, we can find the average of each player.

Figure 5.55 shows how this query may be formulated in SQL.

Before the GROUP BY the query removes all the rows for which *Country* is not India. Now the GROUP BY clause groups the rows by *LName* and the function AVE then computes the average runs for each virtual group.

```
SELECT FName, AVE(NRuns) AS AveRuns
FROM Player, Batting
WHERE PlayerID = PID
AND Country = 'India'
GROUP BY FName
```

Figure 5.55 SQL for query Q30

The result of the above query *Q*30 is given in Table 5.30.

It should be noted that we could not have used *FName* in the SELECT clause if we had used GROUP BY *PlayerID* because the system has no way of knowing that for each *PlayerID* there is a unique *FName*.

Table 5.30 Result of query Q30

FName	*AveRuns*
Yuvraj	38
M.S.	36
Kumar	7
Sachin	91
Harbhajan	3

It should be noted that if we used GROUP BY *LName* then the average for singh would include runs for both Yuvraj Singh and Harbhajan Singh.

• (*Q*31) Find the average score for each player when playing in Australia.

Figure 5.56 shows how this query may be formulated in SQL.

```
SELECT PID AS PlayerID, AVG(NRuns) AS AveRuns
FROM Batting
WHERE MatchID IN
        (SELECT MatchID
        FROM Match
        WHERE Team1= 'Australia')
GROUP BY PID
```

Figure 5.56 SQL for query Q31

This query first finds all rows in *Batting* that are for matches played in Australia by using the subquery. Then the rows are grouped by *PID* and the average score for each player is computed. Unfortunately our example database has batting information from only one ODI that was played in Australia (although *Match* table has information on matches 2675 and 2688 also) and therefore the average scores are the same as the score in that match.

The result of the query is given in Table 5.31.

Table 5.31 Result of query *Q*31

PlayerID	*AveRuns*
89001	91
23001	38
25001	36
99002	2
95001	1
24001	42
99001	7
27001	7
98002	3

• (*Q*32) Find the ID of players that had a higher average score than the average score for all players when they played in Sri Lanka.

Figure 5.57 shows how this query may be formulated in SQL.

```
SELECT PID AS PlayerID, AVG(NRuns) AS AveRuns
FROM Batting
GROUP BY PID
HAVING AVG(NRuns) >
        (SELECT AVG(NRuns)
        FROM Batting
        WHERE MatchID IN
            (SELECT MatchID
            FROM Match
            WHERE Team1= 'Sri Lanka')
        GROUP BY PID)
```

Figure 5.57 SQL for query *Q*32

This query first finds the average score for all players that have batted in Sri Lanka and then the outermost query selects those players that have average scores that are greater than that score.

The result of query *Q*32 is given in Table 5.32.

Table 5.32 Result of query *Q*32

PlayerID	*AveRuns*
25001	53.5
91001	60.0
89001	91.0
24001	42.0

There was only one match in Sri Lanka so the average in the middle query are equal to the number of runs scored in that match.

Multiple Aggregate Functions

In some situations, more than one function may be required. Let us illustrate it by an example.

• (*Q*33) In the query *Q*28, we found the number of players in the database from each country. Now we find the average number of players from this list in Table 5.28.

```
SELECT AVG(COUNT(*)) AS NAvePlayers
FROM Player
GROUP BY Country
```

Figure 5.58 An illegal SQL query for *Q*33

Once we have counted the number of players from each country, as in query *Q*28, the result is a table to which one may be tempted to apply another function to compute the average number of players.

Figure 5.58 shows how one might formulate this query. The formulation is however incorrect.

How do we then formulate this query?

One approach is to use a subquery in the FROM clause. This subquery then produces the table given in Table 5.28. Once we have that table, we may compute the average as shown in Fig. 5.59.

```
SELECT AVG(NPlayers) AS NAvePlayers
FROM
        (SELECT Country, COUNT(*) AS NPlayers
        FROM Player
        GROUP BY Country) NP
```

Figure 5.59 Legal SQL for query Q31

In the subquery, the table *Player* is divided into groups, each group with the same *Country* value. The number of rows in each group is counted resulting in the table *NP* in the FROM clause. Now we can find the average of the *NPlayers* column of this table.

The result of the above query *Q*33 is given in Table 5.33.

Table 5.33 Result of query *Q*33

NAvePlayers
3.2

Later in the chapter we will show how this query may be formulated using a view.

5.4.9 Outer Join

The concept of outer join was discussed in the last chapter. Further details are now presented including some examples in SQL to illustrate the concept.

We will use two new tables. Figure 5.60 lists the top 10 batsmen in ODI matches and Fig. 5.61 lists the top 10 batsmen in Test matches (both based on the total number of runs scored and current on 1st January 2010). The cricket database we have been using does not illustrate the outer join as well as these tables.

The best ODI batsmen in the world on 1st January 2010 are given in Fig. 5.60.

The best test batsmen in the world on 1st January 2010 are given in Fig. 5.61.

As noted earlier the natural join command finds matching rows from the two tables that are being joined and rejects rows that do not match. For example, if we needed to find the names, the total number of runs scored and the number of centuries scored by each player in ODI matches and Test matches we would join *bestODIbatsmen* and *bestTestbatsmen* and then carry out a projection. The SQL query is given in Fig. 5.62.

Player	Span	Matches	Innings	Runs	Ave	Strike Rate	100s
SR Tendulkar	1989-2010	440	429	17394	44.71	85.90	45
ST Jayasuriya	1989-2010	444	432	13428	32.43	91.22	28
RT Ponting	1995-2010	330	321	12311	43.19	80.50	28
Inzamam-ul-Haq	1991-2007	378	350	11739	39.52	74.24	10
SC Ganguly	1992-2007	311	300	11363	41.02	73.70	22
R Dravid	1996-2010	339	313	10765	39.43	71.17	12
BC Lara	1990-2007	299	289	10405	40.48	79.51	19
JH Kallis	1996-2010	295	281	10409	45.25	72.01	16
AC Gilchrist	1996-2008	287	279	9619	35.89	96.94	16
Mohammed Yousuf	1998-2009	276	261	9495	42.96	75.30	15

Figure 5.60 The table *bestODIbatsmen*

Player	Span	Matches	Innings	TRuns	Ave	T100s
SR Tendulkar	1989-2010	162	265	12970	54.72	43
BC Lara	1990-2006	131	232	11953	52.88	34
RT Ponting	1995-2010	140	236	11550	55.26	38
AR Border	1978-1994	156	265	11174	50.56	27
SR Waugh	1985-2004	168	260	10927	51.06	32
R Dravid	1996-2010	137	237	11256	53.60	28
JH Kallis	1995-2010	133	225	10479	54.57	32
SM Gavaskar	1971-1987	125	214	10122	51.12	34
GA Gooch	1975-1995	118	215	8900	42.58	20
Javed Miandad	1976-1993	124	189	8832	52.57	23

Figure 5.61 The table *bestTestbatsmen*

```
SELECT ORuns, 100s, Player, TRuns, T100s
From bestODIbatsmen AS b1, bestTestbatsmen AS b2
WHERE b1.Player = b2.Player
```

Figure 5.62 SQL query involving a join

The result of this query is given in Table 5.34.

This result of course only gives information about players that are in both the tables given in Figs 5.60 and 5.61. No information is provided about players that are in one table but not the other. The outer join provides information about rows that match in the two tables and also about rows that do not match.

Table 5.34 Result of the inner join query

ORuns	*100s*	*Player*	*TRuns*	*T100s*
17394	45	SR Tendulkar	12970	43
12311	28	RT Ponting	11550	38
10405	19	BC Lara	11953	34
10765	12	R Dravid	11256	28
10409	16	JH Kallis	10479	32

Consider the Fig. 5.63 which explains the concept of inner and outer joins. The figure shows the distribution of join attribute values for two relations *R* and *S*. The rows for which the join attribute values match are shown in the middle and these rows form the natural join of the two tables. The remaining rows of the two relations are rows that did not find a matching row from the other relation. There are therefore three groups of rows from the two relations:

- Rows from both relations that have matching join attribute values and are therefore combined to form the (inner) natural join of *R* and *S*.
- Rows from relation *R* that did not match rows from *S*.
- Rows from relation *S* that did not match rows from *R*.

These three regions are shown in Fig. 5.63. It shows that the left outer join of tables *R* and *S* has all the rows of relation *R* including those that are in the natural join as well as those that are not. The right outer join has all the rows of relation *S* including those that are in the natural join as well as those that are not. Finally, the full outer join of the join has all the rows of *R* and *S* as well as those that form the natural join. We now illustrate these concepts using the example.

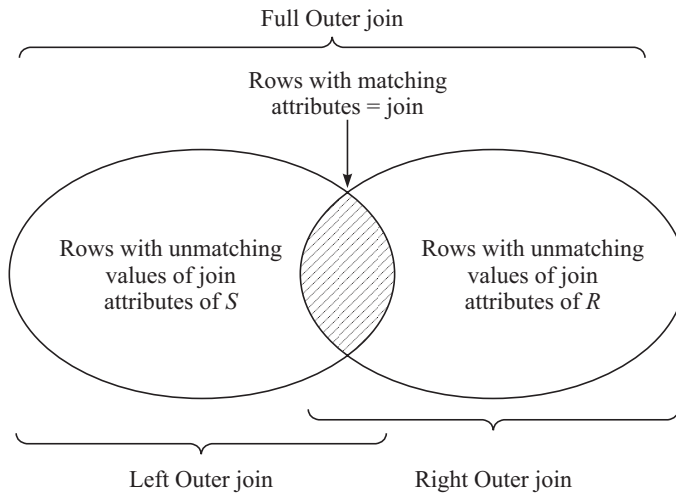

Figure 5.63 Inner join and outer joins

• (*Q*34) Find the left outer join of the tables in Figs 5.60 and 5.61 and find the total number of runs scored, total number of centuries, and the strike rate in the ODIs for each player.

Consider the left outer join between the two tables *bestODIbatsmen* and *bestTestbatsmen*. It is formulated in SQL using the FROM clause as given in Fig. 5.64. We find the total number of runs scored in both types of the game and the total number of centuries scored in the query in Fig. 5.64.

```
SELECT ORuns+TRuns AS TR, 100s+T100s AS TC, SR, Player
From bestODIbatsmen AS b1 LEFT OUTER JOIN bestTestbatsmen AS b2
ON b1.Player = b2.Player
```

Figure 5.64 SQL query involving a left outer join

The result of this query *Q*34 is given in Table 5.35 which gives the total number of runs (TR), total number of centuries (TC) and strike rate for players that have made it to the list of top ten batsmen in both ODI matches and test matches but it also shows those players that appear in the list of top 10 ODI batsmen but do not in the list of top 10 test batsmen. Since they do not appear in both tables, some of the information for them is given as NULL. These entries did not appear in the result of a natural join in Table 5.34.

Table 5.35 Result of the left outer join query in query *Q*34

TR	*TC*	*SR*	*Player*
30364	88	85.90	SR Tendulkar
23861	66	80.50	RT Ponting
22358	53	79.51	BC Lara
22021	40	71.17	R Dravid
20888	48	72.01	JH Kallis
NULL	NULL	91.22	ST Jayasuriya
NULL	NULL	74.24	Inzamam-ul-Haq
NULL	NULL	73.70	SC Ganguly
NULL	NULL	96.94	AC Gilchrist
NULL	NULL	75.30	Mohammed Yusuf

Similarly we can carry out a right outer join of the two tables which will give us a list of players who are in both tables as well as those that are in the test batsmen table but not in the ODI table. We leave it as an exercise for the reader to find the right outer join.

• (*Q*35) Find the full outer join of the tables in Figs 5.60 and 5.61 and find the number of runs scored in ODIs and tests, total number of centuries in ODIs and tests, and the strike rate in the ODIs for each player.

The full outer join is illustrated in the query in Fig. 5.65 which is formulated once again using the FROM clause.

```
SELECT ORuns, 100s, SR, Player, TRuns, T100s
From bestODIbatsmen AS b1 FULL OUTER JOIN bestTestbatsmen AS b2
ON b1.Player = b2.Player
```

Figure 5.65 SQL query involving the full outer join

The result of this query is given in Table 5.36 which gives the number of ODI runs, number of ODI centuries and strike rate as well as number of test runs and test centuries. Table 5.36 shows players who are in both lists (which is what we got in the natural join) as well as players who are in either of the lists but not in both. Some information about the top 10 test players that are not in top 10 ODI batsmen in the left three columns is NULL while batsmen who are in the ODI list but not the test list have two rightmost columns as NULL. These players would not appear in the result of a natural join which will only have players that are in both lists.

The result of a full join is quite useful since it allows us to easily identify players who appear in the list *bestODIbatsmen* and not *bestTestbatsmen* and also the other way around.

Table 5.36 Result of the full outer join query formulated in Figure 5.63

ORuns	*100s*	*SR*	*Player*	*TRuns*	*T100s*
17394	45	85.90	SR Tendulkar	12970	43
12311	28	80.50	RT Ponting	11550	38
10405	19	79.51	BC Lara	11953	34
10765	12	71.17	R Dravid	11256	28
10409	16	72.01	JH Kallis	10479	32
13428	28	91.22	ST Jayasuriya	NULL	NULL
11739	10	74.24	Inzamam-ul-Haq	NULL	NULL
11363	22	73.70	SC Ganguly	NULL	NULL
9619	16	96.94	AC Gilchrist	NULL	NULL
9495	15	75.30	Mohammed Yousuf	NULL	NULL
NULL	NULL	NULL	AR Border	11174	27
NULL	NULL	NULL	SR Waugh	10927	32
NULL	NULL	NULL	SM Gavaskar	10122	34
NULL	NULL	NULL	GA Gooch	8900	20
NULL	NULL	NULL	Javed Miandad	8832	23

Table 5.36 shows that there are a total of 15 batsmen that are either in the top ten list of ODI batsmen or in the top ten list of test batsmen or both. Only five of them are in both the lists. These players are Tendulkar, Ponting, Lara, Dravid and Kallis and they are, in my view, the batting superstars. Since three of these five

players are still playing, this table will probably look a bit different by the time you are reading it. There is very little chance of new players appearing in the two top ten lists since only a small number of players are close to the bottom of these lists at the time of writing. S. Chanderpaul of the West Indies is the only one who is still playing and has 9063 test runs and 8664 ODI runs on 21 February 2011.

5.4.10 Modifying Data in SQL—Update Commands

SQL update commands include the following:

- Adding new data—Adding new rows to tables using the INSERT command.
- Modifying existing data—Modifying column data in existing rows using the UPDATE command.
- Deleting data—Removing rows from existing tables using the DELETE command.

We now illustrate the INSERT, UPDATE and DELETE commands.

Adding New Data

The INSERT command is a very simple way to insert a row into a table. The next query illustrates INSERT.

• (*Q*36) Insert a new player Michael Clarke with player ID 200311 in the table *Player*.

Figure 5.66 shows how this row insertion to the table *Player* may be formulated in SQL.

```
INSERT INTO Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
<200311, 'Clarke', 'Michael', NULL, NULL, NULL, NULL>
```

Figure 5.66 SQL for row insertion Q36

Note that the above insertion procedure is somewhat tedious since it requires all column names to be listed and all corresponding matching values specified. The order of the column names does not need to be the same as given in the CREATE TABLE statement. Furthermore, it is possible to omit some column names from the list and their value is assigned to be NULL. The INSERT statement can be made a little less tedious if the list of column values that are to be inserted are in the same order as specified in the CREATE TABLE definition and all the column values are present. We may then write as given in Fig. 5.67.

```
INSERT INTO Player <200311, "Clarke", 'Michael', NULL, NULL, NULL, NULL>
```

Figure 5.67 SQL for row insertion *Q*36

It should be noted that when one wishes to insert a row, one may not have all the values of the columns. These columns, unless they are part of the primary key of the table, may be given NULL values. If there is relevant data in other tables, that data may be inserted into another table by using results of a query. For example, an INSERT like that given in Fig. 5.68 may be used.

```
INSERT INTO NEWBowling
SELECT PID, MatchID FROM bowling
WHERE NWickets > 4;
```

Figure 5.68 SQL for row insertion using information from another table

Modifying Existing Data

Every database will need modification of data from time to time; the data may have been input incorrectly or may be out-of-date. For example, an employee address or salary may have changed. The UPDATE command is used to carry out modifications.

The basic form of UPDATE command is as given in Fig. 5.69.

Table is the name of the table to be modified. SET Newvalues provides a list of new values for each table column specified while the WHERE condition selects the rows that are to be modified. An UPDATE statement may be used without a WHERE clause in which case the column values are modified for all the rows of the table. This is useful only infrequently.

```
UPDATE Table
      SET Newvalues
      [WHERE condition]
```

Figure 5.69 SQL for UPDATE of rows

• (*Q*37) Modify the number of centuries made by Tendulkar in Tables 5.61 to 51.

Figure 5.70 shows a case of modifying existing data as required in *Q*37.

```
UPDATE bestTestbatsmen
SET T100s = 51
WHERE Player = "SR Tendulkar"
```

Figure 5.70 SQL for UPDATE of row for query *Q*37

• (*Q*38) Modify the number of matches and innings played by Tendulkar to 177 and 290, respectively.

We are modifying some more data in Table 5.61. This is shown in Fig. 5.71.

```
UPDATE bestTestbatsmen
SET Matches = 177,
Innings = 290
WHERE Player = 'SR Tendulkar'
```

Figure 5.71 SQL for UPDATE of more than one colum

Deleting Data

Similar to UPDATE, all or some selected rows of a table may be deleted by a command illustrated in Fig. 5.72.

• (*Q*39) Delete Match 2689 from the table *Match*.

Figure 5.72 shows how this delete operation to delete a row corresponding to match 2689 may be formulated in SQL.

```
DELETE Match
WHERE MatchID = '2689'
```

Figure 5.72 SQL for Delete Q39

The format of the DELETE command is very similar to that of the SELECT command. The DELETE command goes down the table *Match* row by row and deletes rows that meet the condition *MatchID* = '2689'. DELETE may therefore be used to either delete one row or a group of rows that satisfy the given condition.

It should be understood that DELETE commands like that in Fig. 5.72 and the one in Fig. 5.73 below are likely to have side effects if the rows deleted by these commands are being referenced by rows in the other tables.

• (*Q*40) Delete all bowling records of Brian Lara.

Once again the query is similar to some we have seen before. We use a subquery to find the player ID of Brian Lara and then delete all rows for that *Player ID* from the table *Bowling*.

Figure 5.73 shows how this delete operation may be formulated in SQL.

```
DELETE Bowling
WHERE PID =
        (SELECT PlayerID
        FROM Player
        WHERE LName = 'Lara')
```

Figure 5.73 SQL for Delete Q40

We have assumed that the last name Lara is unique.

5.5 VIEWS—USING VIRTUAL TABLES IN SQL

When a database is defined using the CREATE TABLE command in SQL, the user is instructing the DBMS to create tables and store them on disk. These tables can then be retrieved, updated or deleted as the user wishes. These are real tables or base tables that make up the database. The database also, as noted earlier, creates a lot of temporary tables when the database is queried. We have noted earlier that the result of any SQL query is itself a table. In normal query sessions, a query is executed and a table is materialized, displayed; and once the query is completed, discarded. In some situations it may be convenient to store the query definition as a definition of a table that could be used in other queries. Such a table is called a *view* of the database. A view is a virtual table that does not really exist physically as the base tables do, and it has no real existence, as it is defined as a query on the existing base tables. The user is able to treat a view like a base table that would be returned when the view definition is executed and all SELECT-FROM query commands may query views just like they query the base tables. Since a view is a virtual table, it is automatically updated when the base tables are updated. The user may in fact also be able to update the view if desired, and update the base tables used by the view, but, as we shall see later, not all views can be updated.

The facility to define views is useful in many ways. It is useful in controlling access to a database. Users may be permitted to see and manipulate only that data which is visible through some views. It also provides logical independence as the user dealing with the database through a view does not need to be aware of the tables that exist, since a view may be based on one or more base tables. If the structure of the base tables is changed (e.g., a column added or a table split in two), the view definition may need to be changed as well but the user's view will not be affected.

5.5.1 Defining Views

Views may be defined in a very simple way in SQL. As an example of view definition we define *Batsmen* as in Fig. 5.74.

```
CREATE VIEW Batsmen (PID, FName, LName, Country, MID, Score)
AS SELECT PlayerID, FName, LName, Country, MatchID, NRuns
FROM Player, Batting
WHERE PlayerID = PID
```

Figure 5.74 Creating a view in SQL

Another example is the following view definition that provides information about all the bowlers in ODI match 2689. Figure 5.75 shows how this can be created in SQL.

```
CREATE VIEW Bowling2689 (PID, FName, LName, Country, NOvers, NWickets)
AS SELECT PlayerID, FName, LName, Country, NOvers, NWickets
FROM Player, Bowling
WHERE PlayerID = PID
AND MatchID = '2689'
```

Figure 5.75 Creating a view for match 2689

Note that we can use column names for the view that are different than the column names in the base tables.

5.5.2 Querying Views

As noted earlier, views may be used in retrieving information as if they were base tables although views do not exist physically. When a query uses a view instead of a base table, the DBMS retrieves the view definition from meta-data and uses it to compose a new query that would use only the base tables. This query is then processed. Let us consider some examples.

• (*Q*41) Find the names of all players who have scored centuries.

Figure 5.76 shows how this query may be formulated in SQL using a view. In this case the view *Batsmen* defined in Fig. 5.74 is being used.

```
SELECT FName, LName
FROM Batsmen
WHERE Score ≥ 100
```

Figure 5.76 Using the view *Batsmen*

• (*Q*42) Find the names of bowlers that have taken five wickets or more in an ODI match 2689.

Figure 5.77 shows how this query may be formulated in SQL using a view. In this case the view *Bowling2689* defined in Fig. 5.75 is being used.

```
SELECT FName, LName
FROM Bowling2689
WHERE NWickets ≥ 5
```

Figure 5.77 Using the view *Bowling2689*

The above queries on views are transformed by the system by using the view definitions and then processed. The transformation is relatively easy as it only requires the view name to be replaced by the tables' names in the FROM clause of the view definition. A WHERE clause of the view definition is included in addition to the WHERE clause specified in the query on the view.

After transformation, the queries in Figs 5.76 and 5.77 would look like what is given in Figs 5.78 and 5.79.

```
SELECT FName, LName
FROM Player, Batting
WHERE PlayerID = PID;
WHERE NRuns ≥ 100
```

Figure 5.78 Query in Fig. 5.76 after transformation of the view definition

Transformation of the query in Fig. 5.77 is given in Fig. 5.79.

```
SELECT FName, LName
FROM Player, Bowling
WHERE PlayerID = PID
AND MID = '2689'
AND NWickets ≥ 5
```

Figure 5.79 Query in figure 5.77 after transformation of the view definition

Now the queries may be processed as normal SQL queries.

Earlier in Section 5.4.8 we considered how to pose queries in SQL when multiple aggregate functions are required in the query. Query *Q*33 was formulated using a subquery in the FROM clause. The subquery in the FROM clause could in fact be defined as a view and the view used in the FROM clause to obtain exactly the same result.

We formulate another query to illustrate the use of multiple aggregate functions using views.

• (*Q*43) Find the country that has the maximum number of players in the database.

This query requires counting of players in each country and then finding the country that has the maximum number of players. We will first define a view that does the counting of the players and then use that view to find the country with maximum number of players.

Figure 5.80 defines the view *NPlayers* that we require for obtaining the names of countries and the number of players from each of them.

```
CREATE VIEW NPlayers (Country, Count (*) as C)
AS SELECT Country, COUNT (*)
FROM Player
GROUP BY Country
```

Figure 5.80 Creating a view for number of players from each *Country*

NPlayers can now be used as in Fig. 5.81.

```
SELECT Country, COUNT(*)
FROM Player
GROUP BY Country
HAVING COUNT(*) =
        (SELECT MAX(C)
        FROM NPlayers)
```

Figure 5.81 Finding the *Country* that has maximum number of players

Figure 5.81 shows how the two aggregate functions MAX and COUNT may be used together in a query by using a view.

5.5.3 Updating Views

A number of interesting questions arise regarding views since they are virtual tables. For example:

- Can queries use views as well as base tables?

- Can views be updated?
- Can rows be inserted or deleted in a view in spite of views being virtual tables?

The answer to the first question is yes. There is no difficulty in using views and base tables in the same query as the query is going to be transformed into a query that uses base tables anyway.

The answer to the second and third question is maybe! Essentially if the view definition is such that a row in a view can directly identify one or more rows in the base tables then the views may be updated and rows be inserted in the view. It means the view must have a primary key of one or more base tables. The following points are worth noting about modifying data through views:

- A view to be modified must not contain any aggregate functions although a subquery used in the view may.
- The view to be modified must not use the duplicate removal DISTINCT.
- The view must not include calculated columns.
- A view may be deleted in a similar fashion as deleting a table.

Further discussion of this topic is beyond the scope of this book.

5.6 USING SQL IN PROCEDURAL PROGRAMMING LANGUAGES–EMBEDDED SQL AND DYNAMIC SQL

We have so far covered the interactive use of SQL in which the SQL queries are submitted from a terminal, executed and results returned to the terminal (sometime called *Direct SQL*). Although SQL provides powerful features to build programs, SQL is sometimes inadequate for writing application programs. Embedding SQL commands to access and manipulate the database in a conventional programming language (called the *host language*) like C or C++ allows power of SQL to be combined with the facilities of a programming language, for example, looping, to be used together to build application programs that use a database.

5.6.1 Embedded SQL

Use of embedded SQL is common in some application development environments. Embedded programming is useful when a previously written program in a procedural language needs to access a database or because SQL is not providing the facilities, for example, it has no looping facilities like IF..THEN..ELSE. On the other hand using SQL in a procedural programming language like C or C++ also creates some problems because the procedural languages are not equipped to deal with collections of multiple rows as a single operand. To overcome this difficulty, a facility is needed for stepping through a collection of rows returned by a SQL query, one row at a time. This is done by introducing the concept of a *cursor*.

It is possible to use SQL from within a program written in almost any of the conventional programming languages. Since conventional programming languages know nothing about SQL, no database table names may be used in a host language program outside the embedded SQL commands. An SQL query in a host

language can appear anywhere in the host program but it must be identified by beginning it with EXEC SQL command followed by SQL statements or declarations. For example, the embedded SQL in a host language may look like the query in Fig. 5.82 with the semicolon usually terminating the command. This query does not deal with the problem of going through the results step by step. We shall discuss that soon.

```
EXEC SQL
SELECT *
FROM Player
WHERE Country = 'India';
```

Figure 5.82 A simple incomplete SQL query in embedded SQL

SQL statements in a procedural language are handled by the language preprocessor (sometime called the precompiler) for that language before the resulting program may be compiled. A preprocessor is therefore required if one wishes to use SQL in a host language. When necessary, SQL variables may be declared within a SQL declaration section by starting the declaration section by EXEC SQL BEGIN DECLARE SECTION and ending it by the command EXEC SQL END DECLARE SECTION. When a program with a SQL query is preprocessed, the SQL query becomes a procedure call of the host language and the program may then be executed. Such a use of SQL is called *embedded SQL.*

This process is explained in Fig. 5.83 that shows the preprocessor phase, followed by the compilation phase and finally the execution phase of the program.

A number of issues arise when using embedded SQL. Two of them are as follows:

1. There needs to be an error detection mechanism when using SQL in a host language.
2. The host language does not have any working storage allocated for SQL results. Some working storage is required.

We now consider how these problems are resolved in embedded SQL

5.6.2 Error Handling Using SQLCA, SQLCODE and SQLSTATE

As noted earlier a program in a host programming language knows nothing about the database it uses and thus there is considerable data independence between the program and the database. To make this interface between the host program and SQL work, the host program needs some communication working storage. Although implementation may vary from one DBMS to another, a communication area is provided to communicate between the database and the embedded program. This communication area may be accessed in the host program by including the following statement where SQLCA stands for SQL communication area.

```
EXEC SQL INCLUDE SQLCA;
```

Once SQLCA has been declared, two variables SQLCODE and SQLSTATE may now be used to detect error conditions in an embedded program.

All variables used to store database column values that are retrieved must be declared in the host language between the declaration statements as noted earlier.

```
EXEC SQL BEGIN DECLARE SECTION
int PlayerID;
char FName[20], LName[20;
EXEC SQL END DECLARE SECTION
```

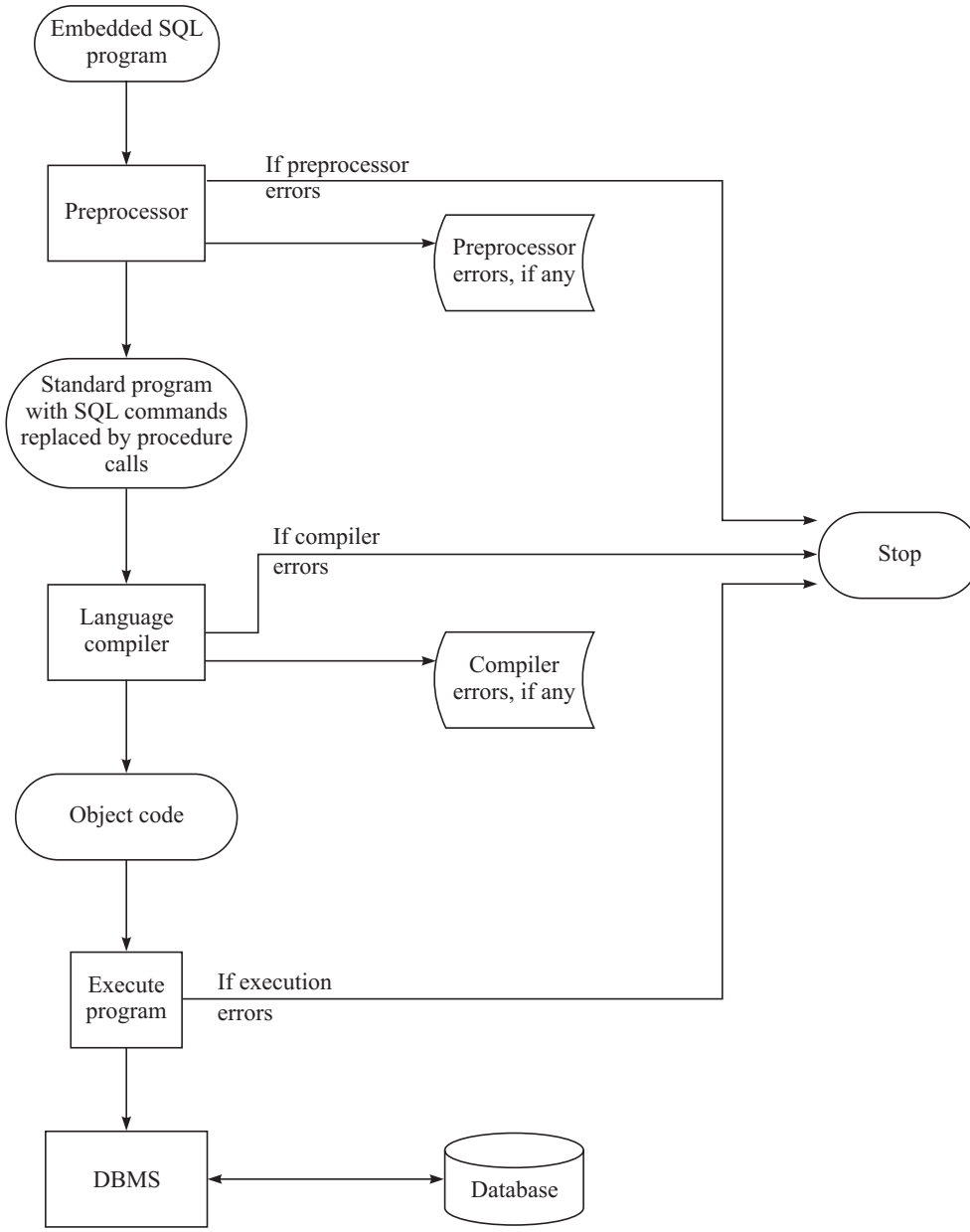

Figure 5.83 Preprocessing and compilation of an embedded SQL program

The host variables declared here are used in the SQL queries embedded in the host language program. Note that these variables are called shared variables and they are prefixed by a colon (:) when used.

5.6.3 Impedance Mismatch and Cursors

As noted above, the host language does not have any working storage for SQL results that are tables, usually with more than one row. Some SQL commands like UPDATE, DELETE and INSERT of course do not require any additional storage in the host program as shown below.

Update

```
EXEC SQL
UPDATE bestTestbatsmen
T100s = T100s + 4
WHERE Player = 'SR Tendulkar'
```

Select

When a number of rows are selected by an embedded SQL query then the concept of *cursor* as defined below must be used. As noted earlier, a cursor is needed since most conventional host programming languages are record-based, while SQL is set-based. This is called *impedance mismatch.* Therefore we need a binding mechanism that makes it possible to retrieve one record at a time as well as map the attribute values to the host language variables. A cursor (it may be thought of as a pointer) allows us to do that by essentially holding the position of the current tuple of the table that is the result of the SQL query while the host language deals with the table one row at a time. The cursor is not associated with any tables in the database, it is only associated with the table that is the result of an embedded SQL query.

Once a cursor is defined, it may be opened to point at just before the first row of the resulting table. There are two other operations available, *fetch* and *move*. Fetch gets the next row of the result while move may be used to move the cursor. Close may be used to close the cursor.

```
EXEC SQL DECLARE cursor1 CURSOR FOR
SELECT PlayerID, FName, LName
FROM Player
WHERE Country = "India";
```

Now we may issue the following commands:

```
OPEN cursor1
FETCH cursor1 INTO :var1, :var2, :var3
MOVE cursor1
CLOSE cursor1
```

Embedded SQL allows further processing of each tuple to be carried out in the host programming language.

Summary of Using Embedded SQL

1. First the declarations:

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
```

```
Int PlayerID
Char LName[20], FName[20]
EXEC SQL END DECLARE SECTION;
```

2. Queries (with multiple rows result)

```
EXEC SQL DECLARE scursor CURSOR FOR
(SELECT PlayerID, FName, LName
FROM Player
WHERE Country = 'India');
EXEC SQL OPEN scursor;
do {
EXEC SQL FETCH scursor INTO <vars>;
} while (SQLCODE == 0);
EXEC SQL CLOSE scursor;
```

5.6.4 Dynamic SQL

Dynamic SQL is a programming technique that enables a program to build SQL statements dynamically at runtime. Thus more general purpose flexible applications may be created.

In some cases when using embedded SQL in a host programming language it becomes necessary to use and build the SQL query dynamically at runtime. Building SQL queries dynamically provides the user a flexible environment because the full details of a SQL query may not be known at the time of writing the program. For example, the name of the table that is to be used in the query may not be known prior to running the program. This can happen, for example, if the application program is accessing daily sales figures which are stored in tables that are named with the date, as Sales20100627 for 27 June 2010. Similar problems may arise in a reporting application, in a data warehouse application where the table name changes every time the reporting program is run. In some situations a query on the Web might allow the user to select the sort order. Instead of coding the query a number of times, with different ORDER BY clauses, a dynamic SQL query including a user specified ORDER BY clause might be the preferred solution.

Dynamic queries are useful where one of the following conditions holds:

- *An application program needs to allow users to input or choose query search or sorting criteria at runtime.*
- *An application program queries a database where the data definitions of tables are changing regularly.*
- *An application program queries a database where new tables are being created frequently and these tables are used in the queries.*

We now describe the difference between static SQL and dynamic SQL. Static SQL statements do not change from execution to execution. The full static SQL query is known when the query is processed, which provides the following benefits:

- Query processing can verify that the SQL query references valid database objects.

- Query processing can verify that the user has necessary privileges to access the database objects.
- Performance of static SQL is generally better than that of dynamic SQL.

Because of these advantages, dynamic SQL should be used only if static SQL will not achieve the query goals or if using static SQL is cumbersome compared to using dynamic SQL. However, as noted earlier, some table names may not be known when the program is written and the program is run frequently.

Dynamic SQL programs can handle changes in data definitions, without the need to recompile. This makes dynamic SQL much more flexible than static SQL. Dynamic SQL allows reusable code to be written because the SQL can be easily adapted for different environments.

5.7 DATA INTEGRITY AND CONSTRAINTS

We noted at the beginning of Chapter 3 that the relational model has three main components: data structure, data manipulation and data integrity. We have so far discussed the data structure, three data manipulation languages (relational algebra and relational calculus in the last chapter and SQL in this chapter) and data integrity. We now discuss how SQL may be used in maintaining integrity by specifying integrity constraints.

The CREATE TABLE command may include specification of integrity constraints which are usually either *column constraints* (associated with a single column) or *table constraints* (usually associated with more than one column). Any column constraint may also be defined as a table constraint.

We will look at the following types of constraints.

- Primary Key
- Foreign Key
- Domains
- Nulls

5.7.1 Primary Key Constraint

We have defined the concepts of candidate key and primary key in Chapter 3. From the definition of a primary key, it is required that each table have one primary key even if it is the combination of all the columns in the table, since all rows in a table are defined to be distinct.

One or more columns of a table may be defined as primary keys. The primary key of course is unique for each row of the table. No two rows can have the same primary key value.

One may consider the primary key definition as a constraint on the table that is being created. The primary key may be defined in the CREATE TABLE simply by using the PRIMARY KEY command as follows:

```
PRIMARY KEY (PlayerID)
```

In enforcing the existential integrity constraint the database system ensures that the primary key values are unique.

5.7.2 UNIQUE Constraint

As discussed above, a primary key has to be unique but columns that are not included in the primary key may also be defined as UNIQUE. For example, names of players may be unique and may be declared as such. A column defined as UNIQUE does not need to be NON NULL; it only needs to have all values that are not null as unique.

5.7.3 Foreign Key Constraint

As discussed in Chapter 3, the concept of a foreign key and referential integrity is important since a relational database consists of tables only (no pointers) and relationships between the tables are implicit, based on references to the primary keys of other tables. These references are called *foreign keys*.

The foreign key constraint applies on the column or columns in the referencing table, which is the table that refers to the primary key of another table. For example, it applies to the *PID* columns in tables *Batting* and *Bowling* since they refer to the *PlayerID* column in the table *Player*. It may be declared in the *Bowling* table definition as in Fig. 5.84.

```
CREATE TABLE Bowling
(MatchID INTEGER NOT NULL,
PID INTEGER NOT NULL,
NOvers INTEGER,
Maidens INTEGER,
NRuns INTEGER,
NWickets INTEGER,
PRIMARY KEY (MID, PID)
FOREIGN KEY (MatchID) REFERENCES Match
FOREIGN KEY (PID) REFERENCES Player)
```

Figure 5.84 SQL for data definition specifying foreign keys

This foreign key constraint will ensure that no row may be inserted in the table *Bowling* if the *PID* does not match with a *PlayerID* that is already in the table *Player* and *MatchID* does not match with the column *MatchID* in the table *Match*.

As mentioned earlier, a foreign key of a table may also refer to the primary key of the same table.

5.7.4 Domains

As noted earlier, the value of an attribute of a table must be drawn form a given set of values or be of specific data type. Domains can be specified as shown in Fig. 5.85. It should be used that although NAME1 and NAME2 in Fig. 5.85 are both declared as CHAR(10). They are different domains.

CREATE DOMAIN NAME1	CHAR(10)
CREATE DOMAIN Player1	INTEGER
CREATE DOMAIN NAME2	CHAR(10)

Figure 5.85 SQL for specifying domains

5.7.5 NULL Values

The issue of null values in a database is extremely important. A value in a database may be null for several reasons. The most obvious is that the value is not known at the present time. For example, a new employee's home telephone number will not be known until a phone is connected to his/her residence. Another reason some values in a database may be null are that the attribute may not be applicable. For example, an employee database may have an attribute spouse which is not applicable for a single employee.

If some attributes are not allowed to be NULL then NOT NULL should be appended to the definition of those attributes in table definition. Certainly the primary keys of tables should not be allowed NULL values.

5.8 TRIGGERS AND ACTIVE DATABASES

Triggers are an event handling mechanism for enforcing some types of data integrity.

Let us first define a trigger. A trigger is a kind of special procedure that is associated with a particular table and it executes automatically when data in the table is modified because some types of data modification may require special attention. A number of triggers may be defined on the same table. It may be that for some table one would like to be warned when a new row is inserted or deleted or updated.

A trigger, that is the procedure, may be fired before an event or after an event. A trigger may also be executed instead of an event.

Triggers provide reactive behaviour which is motivated by the occurrence of some event, typically a database operation. Triggers therefore are a practical way to check for integrity constraints and will be discussed in more detail in Chapter 12. Triggers also have other applications, for example, scheduling and real-time applications. Databases that use triggers are often called active database systems. The term can be defined as

Definition—Active Databases
A database that is designed to react to an internal or external stimulus is called an active database.

Active databases have a variety of applications. For example, an inventory control system could generate an alert when the number of a product in the inventory falls below some specified number.

The chapter on database integrity (Chapter 12) includes more information on triggers.

5.9 DATA CONTROL–DATABASE SECURITY

Databases store a large amount of personal and valuable business information. It is therefore essential that this information be protected from unauthorized access. Security of information is a complex subject and therefore we are able to only briefly introduce the subject here. There are two fundamental concepts of security, viz., authentication and authorization.

Authentication deals with uniquely identifying users, for example, as done by using a user name and password to access a computer or using a card and pin number for accessing an account through an ATM. More recently, it has been suggested that biometric techniques like fingerprints can provide a more secure method of authentication. To establish an authentication procedure there must be a facility to set up an account for each user that is to be authenticated and a password be provided to the user.

Authorization deals with the decision making regarding what parts of a database an authenticated user is allowed to access. For example, anyone can access the library catalogue but to be able to access student files in a university, a user must not only be an authenticated user but also have permission to access student files. To establish an authorization procedure, there must be procedures to allow a user permission to access some part of the database for reading only or for reading and writing. Views may be defined for a user to limit the content that a user can access.

In addition to identification, authentication and authorization, we need additional security facilities to protect a database. SQL provides two major techniques to provide security. These are the view mechanism and the GRANT/REVOKE mechanism.

We now briefly describe the GRANT and REVOKE commands available in SQL for authorization. There are no commands for authentication since it is assumed the DBMS provides facilities for authentication. View is a very useful security mechanism which allows a part of the database to be defined so that a user may be authorized to access only that part. Although the cricket database is not a good example to illustrate security mechanisms, let us assume that for some security reasons a user is only allowed to access information from the cricket database about players from his/her own country. A view may then be defined which will allow such an information to be available. GRANT command may then be used to grant authorization.

The GRANT command in SQL is used to authorize a user to carry out the specified operations (for example, SELECT, CREATE, INSERT, UPDATE and DELETE) on the listed tables or views. A user must have appropriate privileges before whatever operation the user wishes to carry out on some tables in the database.

The basic form of GRANT and REVOKE is shown in Fig. 5.86.

```
GRANT privilege ON resource TO username
REVOKE privilege ON resource TO username
```

Figure 5.86 SQL for granting and revoking privileges

The chapter on database security (Chapter 11) includes more information on these issues.

5.10 FURTHER DATA DEFINITION COMMANDS

In Section 5.3, at the beginning of this chapter, we discussed the most important data definition commands in SQL. These included CREATE TABLE, ALTER TABLE and the DROP commands. We now discuss a number of DDL commands in more detail.

5.10.1 The CREATE Command

As described in Section 5.3.1, the CREATE TABLE command is used to define a new table to the database. The CREATE command may also be used to create a view as shown in Section 5.5.1. It may also be used to create an index, an assertion, a domain, and a trigger.

Although the details of how a query is processed and how a database is stored are discussed in the following chapters, we briefly discuss the CREATE INDEX command that can assist in accessing data in the database faster. Details of how an index works are given in Chapter 7.

Suppose we have a large cricket database with a large number of players and many rows in the tables *Bowling* and *Batting*. Since most queries regarding batting and bowling access the database by using the *Player ID*, it may make sense to improve the efficiency of such queries. This can be achieved by creating an index on column *PID* in both tables.

Figure 5.87 shows how the index may be created in SQL.

```
CREATE INDEX PID1_INDX
ON Bowling (PID)
```

Figure 5.87 SQL for index creation on *Bowling*

```
CREATE INDEX PID2_INDX
ON Batting (PID)
```

Figure 5.88 SQL for index creation on *Batting*

And similarly on *Batting*.

Creation of indexes assists in efficient retrieval since a search is carried out using the index rather than each row of the table.

5.10.2 The DROP Command

Dropping a Table

We described how a table may be dropped or destroyed in Section 5.3.3. It can be done by using the DROP TABLE command. Any data that was in the dropped table is lost. As noted in Section 5.3.3, CASCADE and RESTRICT specifications may follow the DROP TABLE command to deal with the problem of disposition of other objects dependent on the table. These objects may include a view whose specification references the dropped table. Another dependency is a table that references the dropped table in a constraint, a CHECK constraint or REFERENCES constraint.

The RESTRICT clause ensures that the table is not, dropped if any dependencies exist. The CASCADE clause specifies that any dependencies be removed before the table is removed.

We also noted in Section 5.3.3 that there are several other DROP commands, for example, DROP DOMAIN, DROP INDEX, and DROP VIEW. We briefly discuss them now.

Dropping Views

A view may be dropped just as a table is dropped from a database. For example, we can drop view Bowling2689 as in Fig. 5.89.

```
DROP VIEW Bowling2689
```

Figure 5.89 Dropping the view Bowling 2689

Dropping Domain

It may at some stage become necessary to drop one of the domains which can be easily done by the command given in Fig. 5.90.

```
DROP DOMAIN NAME1
```

Figure 5.90 Dropping the Domain NAME1

Dropping Index

It may at some stage become necessary to drop one of the indexes which can be easily done by the command given in Fig. 5.91.

```
DROP INDEX PID1_INDX
```

Figure 5.91 Dropping the Index PID1_INDX

5.10.3 The ALTER Command for Modifying Table and Column Definitions

Earlier in Section 5.3.2 we discussed how the ALTER table command may be used for adding a column to a table or dropping a column. The ALTER command may also be used for changing the column definition as well as adding or dropping a table constraint. For example, Fig. 5.3 shows how to add a column to the table *Batting* to include information on how many times a batsman was not out.

The ALTER table command may also be used to add or drop an integrity constraint for a table. The ADD and DROP commands are similar to the ALTER command in Section 5.3.2 as shown in Fig. 5.92.

```
ALTER TABLE table_name <action>

<action> may include
DROP [Column] column_name
ADD [column] column_name
DROP CONSTRAINT  constraint_name
ADD CONSTRAINT constraint_definition
```

Figure 5.92 ADD or DROP commands for modifying a table

5.11 SUMMARY OF SOME SQL COMMANDS

The following list provides a summary of some of the commonly used SQL commands. SQL : 2003 is a much more complex language that includes many more commands.

```
SELECT [DISTINCT| ALL] <List of Columns, Functions, Constants, etc>
        FROM <List of Tables or Views>
        [WHERE <Condition(s)>]
        [GROUP BY < Grouping Columns>]
        [HAVING <Condition>]
        [ORDER BY <Ordering Column(s)> [ASC|DESC];

UPDATE <Table Name>
SET <Column Name> = <Value>
WHERE <Condition>;
INSERT INTO <Table Name> [(Column List)]
VALUES (<Value List>);
DELETE FROM <Table Name> WHERE <Condition>;
CREATE VIEW <View Name> AS <Query>;
CREATE TABLE <Table Name>
        (<Column Name> <Data Type> [(<Size>)] <Column Constraint>, … Other Columns);
```

There are a number of commands available for dropping a table, altering a table, creating an index, dropping an index and for creating and dropping a view.

CREATE DOMAIN—used for defining the domain for a column.
DROP DOMAIN—used for dropping a domain.
ALTER DOMAIN—used for changing an already defined domain.
DROP TABLE—to drop the whole table.
ALTER TABLE—used to drop or add columns or constraints.
DROP INDEX—to drop an already created index.
DROP TRIGGER—to drop an already created trigger.
DROP VIEW—to drop an already created view.
TRUNCATE TABLE—to remove the contents of a table.

Tables may have a variety of constraints as listed below. These are discussed in detail in Chapter 11 on Integrity.

NULL or NOT NULL
UNIQUE
PRIMARY KEY
CHECK
DEFAULT
FOREIGN KEY
UNIQUE
TRIGGER
ASSERTION

5.12 SQL STANDARDS

As noted at the beginning of this chapter, ANSI published the first standard definition of SQL in 1986 which was subsequently adopted by ISO in 1987. The 1987 standard is sometimes called SQL1. It included what most SQL implementations were providing at that time. In case of disagreement amongst the vendors, the areas were not defined by the SQL1 standard. SQL1 specified facilities for using SQL through host languages Cobol, Fortran, Pascal and PL/I, the most widely used languages at that time. SQL1 was defined to have two levels of compliance, viz., entry level (level 1) and full level (level 2). SQL1 unfortunately did not insist on meeting referential integrity requirements. In 1989, a new standard that provided only minor enhancements to SQL1 was published. This standard provided some additional features and defined support for two additional host languages, Ada and C. The two levels of the standard were retained although referential integrity was added to level 2.

The next standard for the language was published in 1992. It included a significant number of new features, was almost six times as long as the previous SQL1 standard, and was defined at three levels, viz., entry, intermediate and full. Some people have called this standard SQL : 1992 while others prefer to call it SQL2. The standards continue to evolve as some vendors implement additional language features to compete with other vendors and the international user community proposes new features to be included. A new standard was published in 1999 (called SQL3 or SQL : 1999). A number of advanced features including object-table features were included in the 1999 standard. Enhanced constraints including full referential integrity constraints were added to the entry level. The higher levels included assertions, domains, outer join, union and intersect operations.

At the time of writing this chapter, the latest SQL standard is SQL : 2003. It has been claimed that this standard does not include many new features although all parts of the SQL : 1999 standard have been revised to correct mistakes of the previous standard. In addition, a new part (Part 14) SQL/XML defines extensions related to XML documents.

The standards are strongly influenced by the vendors (and others) and once a standard is released the vendors do not necessarily implement all of it. Some vendors continue to include features that are not in the standard but are demanded by the user community.

SUMMARY

In this chapter we have discussed the query language SQL in detail. The following topics have been covered:

- SQL is the de facto standard query language for relational databases.
- SQL includes a data definition language (DDL), a data manipulation language (DML) and a data control language (DCL).
- SQL may be used in interactive mode as well as with a host programming language.
- Simple SQL queries may use only one table and use SELECT something FROM tables WHERE a condition is met. SQL is based on relational calculus.

- Data definition in SQL includes updates, data insertion, deletion, and authorization using commands that include CREATE, UPDATE and DELETE.
- SQL provides a variety of data types including INTEGER, REAL, CHARACTER, BLOB, DATE and TIME.
- WHERE clause may be of many different forms, for example, WHERE C1 and C2, WHERE EXISTS, WHERE A LIKE x.
- SQL queries that require information from more than one table are formulated using a join and/or subquery.
- Subqueries are usually used in the WHERE clause but may also be used in the FROM clause.
- SQL queries may use set processing commands UNION, INTERSECT and MINUS.
- Formulation of SQL queries that use subqueries may use EXISTS and NOT EXISTS.
- Some SQL queries require GROUP BY and HAVING clauses.
- Aggregate function like AVE, SUM, MIN, MAX and COUNT may be used in SQL queries. The functions are particularly useful when GROUP BY and HAVING clauses are used.
- Removing duplicates by using DISTICT, sorting results by using ORDER BY and string matching using LIKE is possible in SQL queries.
- Correlated subqueries and complex queries have been discussed. Correlated subqueries are subqueries that use one or more attribute values from the outer query.
- The important concept of outer join has been presented so that when a join is carried out not only the matching rows are shown but also the rows that do not match are displayed.
- The concept of view which is a virtual table has been explained. There are many benefits of views, the most important being to allow users only that part of the database that they need to use. Some views may be updated but not all.
- Advantages and disadvantages of SQL are described including its simplicity.
- The use of SQL as a host language, use of cursors, dynamic SQL is described. These facilities are required when the interactive SQL is not powerful enough for a given application.
- The role of data integrity in the relational model as implemented in SQL is described including existential integrity (primary key constraints), referential integrity (foreign key constraints), domains and NULLS.
- SQL was standardized in 1986, and 1987, followed by the SQL : 1999 standard. Further revisions include significant work on SQL with Java and XML. This resulted in SQL : 2003. Further revisions have been proposed and draft SQL : 2008 has been released.

REVIEW QUESTIONS

1. Where was SQL developed and when did it become an ISO standard? (Section 5.1)
2. Describe the structure of SQL queries and explain the different parts of the query. (Section 5.2)
3. Explain the data definition commands CREATE, ALTER and DROP. Give an example of each of them. (Section 5.3)
4. How are aliases used in SQL queries? What is the purpose of using an alias? (Section 5.4.1)

5. Give an example of formulating a simple SQL query that only selects some rows and columns from a table. (Section 5..4.2)
6. Give an example illustrating the use of DISTINCT to remove duplicates from the result of an SQL query? (Section 5.4.2)
7. Explain how string pattern matching is done in SQL to find the name starting with *Tend* and finishing with *ar*. (Section 5.4.2)
8. List five different forms used in the WHERE clause. (Section 5.4.3)
9. Write an SQL query which finds the names of players whose place of birth is not specified in the database. (Section 5.4.3)
10. How does an SQL use subquery in querying more than one table? Give an example. (Section 5.4.4)
11. Give an example of using a join in an SQL query. (Section 5.4.4)
12. Explain the use of UNION, INTERSECT and EXCEPT in SQL queries. (Section 5.4.4)
13. Show how a subquery may be used in the SQL FROM clause. (Section 5.4.5)
14. What is a correlated SQL subquery? Give an example. (Section 5.4.6)
15. What aggregate functions are available in SQL? (Section 5.4.7)
16. How can data be virtually grouped in SQL queries? Explain how GROUP BY, HAVING and WHERE can be used in the same SQL query. (Section 5.4.8)
17. Give an example of using multiple aggregate functions in an SQL query. (Section 5.4.8)
18. Explain the difference between inner join and outer join. (Section 5.4.9)
19. What is a left outer join, a right outer join and a full outer join? (Section 5.4.9)
20. What commands are used in SQL for data update, data insertion, and data deletion? (Sections 5.4.10)
21. Explain the motivations for using view? (Section 5.5)
22. Describe how the concept of views may be used for security control. (Section 5.5)
23. Is it possible to update views? (Section 5.5.3)
24. Explain how SQL may be used in a procedural programming language. (Section 5.6)
25. What is impedance mismatch? (Section 5.6)
26. Explain when is a dynamic SQL needed. Give an example. (Section 5.6.2)
27. Describe the basic data integrity requirements: existential integrity (primary keys), referential integrity (foreign keys), domains and nulls. (Section 5.7)
28. How can SQL be used in maintaining the above integrity constraints? (Section 5.7)
29. What is a trigger? What is it used for? (Section 5.8)
30. Explain the role of SQL in authentication and authorization. (Section 5.9)
31. Give a list of all CREATE, ALTER and DROP commands in SQL. (Section 5.10)

SHORT ANSWER QUESTIONS

1. Write an SQL query to retrieve all rows of the table *Player*.
2. Write an SQL query to retrieve only the last name of all players from *Player*.
3. What major data types are available in SQL?
4. Write an SQL query to retrieve only the rows of Indian players from *Player*.
5. Write an SQL query to retrieve only the last name of all Indian players from *Player*.

6. What is DISTINCT?
7. Write an SQL query to retrieve the last name of players born after 1980 from *Player*.
8. Write an SQL query to retrieve the player ID of players who scored a century in a match from *Batting*.
9. How can the result of a query be sorted?
10. Write an SQL query to retrieve the last name of players who have not played a test match from *Player*.
11. Write an SQL query to retrieve the player ID of players who have taken more than five wickets in a match from *Bowling*.
12. Write an SQL query to retrieve the number of times Tendulkar has batted in our database from table *Batting*.
13. Write an SQL query to find the highest score of Dhoni from the information in table *Batting*.
14. What is an uncorrelated subquery? Give an example.
15. What is a corelated subquery? Give an example.
16. What aggregate functions are available in SQL?
17. Use an example to explain left outer join, right outer join and full outer join.
18. Give examples of queries involving UNION and INTERSECT.
19. How does view help in database security? Give an example of its use.
20. What is primary key constraint? Give an example.
21. What is referential constraint? Give an example.
22. What does the CREATE TABLE command do? Give an example.
23. Are there other CREATE commands?
24. What does the DROP command do? Give an example.
25. What does the ALTER command do? Give an example.
26. What is a cursor used for?
27. What is impedance mismatch?
28. What is dynamic SQL useful for?
29. What is inclusion dependency?

MULTIPLE CHOICE QUESTIONS

1. Which of the following is **not** true?
   (a) SQL was called SEQUEL before the name SQL was chosen.
   (b) SQL was first designed by E. F. Codd.
   (c) SQL was developed as part of a relational DBMS prototype at IBM San Jose Research Laboratory.
   (d) SQL was first defined around 1974.
2. Which of the following are correct for SQL?
   (a) SQL may be used for data definition as well as data retrieval.
   (b) SQL may be used for defining base tables as well as view tables.
   (c) SQL may be used for defining primary keys as well as foreign keys for each table.
   (d) SQL may be used for creating as well as dropping indexes.
   (e) All of the above

3. Which two of the following statements are true about SQL?
   (a) It is a language very much like relational calculus.
   (b) It is a language very much like relational algebra.
   (c) It is a nonprocedural language.
   (d) It is a procedural language like C++.
4. Which one of the following data definition commands is **not** an SQL command?
   (a) CREATE TABLE
   (b) DROP TABLE
   (c) MODIFY TABLE
   (d) DROP DOMAIN
   (e) CREATE DOMAIN
5. Which one of the following is true about domains in SQL?
   (a) SQL domains are user-defined data types.
   (b) SQL domains must be used in data definition.
   (c) SQL domains provide strong typing.
   (d) SQL domains are only syntactic shorthand for system-defined data types.
   (e) SQL domains allow users to define allowed operations on each domain.
6. Which of the following are true about SQL?
   (a) SQL tables cannot have duplicate rows inserted in them.
   (b) SQL tables do not have any column orderings.
   (c) SQL tables do not have to have atomic column values.
   (d) SQL tables can have a default value defined for each column.
7. Which one of the following is true about SQL?
   (a) SQL automatically eliminates duplicate rows from results .
   (b) SQL automatically sorts the rows from results.
   (c) SQL does not provide commands to compute all relational algebra operators directly.
   (d) SQL does not allow any join other than an equi-join.
8. Which one of the following cannot be included in the CREATE TABLE command in SQL?
   (a) Names and data types of the table columns.
   (b) Primary key and foreign keys of the table.
   (c) Default values for the table columns.
   (d) Who is authorized to access the table.
9. Which one of the following is allowed in SQL?
   (a) Adding or deleting a column from an existing table.
   (b) Adding or deleting a constraint from an existing table.
   (c) Changing a column definition.
   (d) All of the above
10. Which one of the following is **not** true?
    (a) An SQL query must not have more than one join in one query.
    (b) All column names in an SQL query must be qualified by the table name.
    (c) All table names given in an SQL FROM clause must be distinct.
    (d) An SQL query must not have an aggregation function in the HAVING clause.
    (e) All of the above

11. SQL provides a number of special aggregate functions. Which one of the following is **not** included in SQL?
    (a) COUNT (b) SUM (c) MAX (d) MIN
    (e) MEDIAN
12. When functions are used in SQL statements, which one of the following is **not** correct?
    (a) DISTINCT must be used if the duplicates are to be eliminated before the function is applied.
    (b) All cases of COUNT *must* include DISTINCT.
    (c) Any NULLS in the argument column are ignored when functions like AVG, MAX, and MIN are used.
    (d) If the argument happens to be an empty set, COUNT returns zero while other functions return null.
13. Which of the following is **not** correct?
    (a) GROUP BY logically partitions the table; no physical rearranging is done.
    (b) Each expression in the SELECT clause must be a single value per group when GROUP BY is used.
    (c) It is not possible to use WHERE and then use GROUP BY in the same query.
    (d) HAVING is to GROUP BY as WHERE is to SELECT.
    (e) The WHERE clause selects the rows which satisfy the given condition, the HAVING clause selects a group that satisfies the condition using aggregation.
14. Which of the following is **not** true?
    (a) A query may have as many as six or more different SQL formulations.
    (b) Different versions of the same query are guaranteed to be processed equally efficiently.
    (c) Subqueries may be nested to any depth.
    (d) SQL is not a faithful implementation of the relational model.
15. Which of the following is **not** correct?
    (a) Views are virtual tables, tables that do not exist in their own right.
    (b) The user does not see any difference between a base table and a view.
    (c) Any retrieval operations on views are translated into equivalent operations on the base tables.
    (d) Views may be updated in the same way as base tables.
    (e) None of the above

    The following database will be used in a number of questions that follow.

```
student(student_id, student_name, address)
enrolment(student_id, subject_id, mark)
subject(subject_id, subject_name)
```

16. Consider the following query:

```
SELECT subject_name, Count(*)
FROM subject, enrolment
WHERE subject.subject_id = enrolment.subject_id
AND mark > 40
GROUP BY subject_name
HAVING Count(*) > 5
```

Which one of the following is true about the query above?

(a) The query is illegal.
(b) The query retrieves the subject names and number of students in each subject.
(c) The query retrieves the subject names and number of students in each subject that has more than five students.
(d) The query retrieves the subject names and number of students in each subject that has more than five students, each of them with more than 40 marks.

17. Consider the following query:

```
SELECT subject_name, Count(*)
FROM subject, enrolment
WHERE subject.subject_id = enrolment.subject_id
AND mark > 40
AND subject_id IN
        (SELECT subject_id
        FROM enrolment
        GROUP BY subject_id
        HAVING Count(*) > 5 )
GROUP BY subject_name
```

Which one of the following is true?

(a) The query is illegal.
(b) This query is equivalent to the last query.
(c) The query retrieves the subject names and number of students in each subject that has more than five students.
(d) The query retrieves the subject names and number of students in each subject that has more than five students, each having more than 40 marks.

18. Consider the following query:

```
SELECT subject_id
FROM subject
WHERE subject_id IN
 (SELECT subject_id
        FROM student, enrolment
        WHERE student.student_id = enrolment.student-Id
        AND student_name = 'Ravi Kumar')
OR subject-id IN
        (SELECT subject-id
        FROM subject, enrolment
        WHERE subject.subject_id = enrolment.subject_id
        GROUP BY subject_id
        HAVING count(*) > 100)
```

Which of the following is true?

(a) The above query is illegal.

(b) The query retrieves the subject id of subjects that have a student by the name of Ravi Kumar and have an enrolment of over 100.
(c) The query retrieves the subject id of subjects that have a student by the name of Ravi Kumar or have an enrolment of over 100.
(d) The query retrieves the subject id of subjects that have a student by the name of Ravi Kumar and do not have an enrolment of over 100.

19. Consider the following query:

```
SELECT student_name, avg(mark)
FROM student, enrolment
WHERE student.student_id = enrolment.student_id
```

Which one of the following is correct for the query?
(a) The query is not legal.
(b) The query retrieves student names and their average mark.
(c) The query retrieves student names and the class average mark for each student.
(d) The query retrieves student names and the mark in each subject.

20. Consider the following query:

```
SELECT subject_name
FROM subject, enrolment
WHERE subject.subject_id = enrolment.subject_id
AND AVG(mark) >70
```

Which one of the following is correct for the query?
(a) The query is not legal.
(b) The query retrieves subject names where the average mark is greater than 70.
(c) The query retrieves subject names which have some mark greater than 70.
(d) The query retrieves subject names which have a student with an average mark greater than 70.

21. Consider the following query:

```
SELECT student_name, subject_name, AVG(mark)
FROM subject, enrolment
WHERE subject.subject_id = enrolment.subject_id
GROUP BY subject_id
HAVING AVG(mark) >70
```

Which one of the following is correct for the query?
(a) The query is not legal.
(b) The query retrieves student name, subject name and the subject average mark.
(c) The query retrieves student name, subject name and the student average mark.
(d) The query retrieves student name, subject name and the student average mark only for those subjects where the average mark is greater than 70.

22. Consider the following query:

```
SELECT subject_id
FROM subject, enrolment
WHERE subject.subject_id = enrolment.subject_id
GROUP BY subject_id
HAVING AVG(mark) >70
```

Which one of the following is correct for the query?

(a) The query is not legal.
(b) The query retrieves subject id of all subjects.
(c) The query retrieves subject id of subjects that have an average mark greater than 70.
(d) The query could not be written without a join or a subquery.

23. Consider the following query:

```
SELECT subject_name
FROM subject, enrolment
WHERE subject.subject_id = enrolment.subject_id
GROUP BY subject_name
HAVING mark > 70
```

Which one of the following is correct for the query?

(a) The query is not legal.
(b) The query retrieves subject names of all subjects.
(c) The query retrieves subject name of subjects that have an average mark greater than 70.
(d) The query retrieves subject name of subjects where some mark is greater than 70.
(e) None of the above

24. Consider the following query:

```
SELECT subject_name
FROM subject, enrolment
GROUP BY subject_id
```

Which one of the following is correct for the query?

(a) The query is not legal.
(b) The query retrieves subject names of all subjects.
(c) The query retrieves (once?) subject names of all subjects that have students enrolled in them.
(d) The query could not be formulated without a join or a subquery.
(e) None of the above

25. Consider the following query:

```
SELECT student_name
FROM student, enrolment
WHERE student.student_id = enrolment.student_id
AND mark IS NULL
```

Which one of the following is correct for the query?

(a) The query is not legal.
(b) The query retrieves student name of each of those students who have no mark given for all their subjects.
(c) The query retrieves once, the student name of each of those students who have no mark given for some of their subjects.
(d) The query retrieves student name every time an enrolment in a subject has no mark given.
(e) The above query does not retrieve any duplicate student names.

26. Consider the following query:

```
SELECT subject_name
FROM subject, enrolment
GROUP BY subject_id
HAVING MAX(AVG(mark)) > 60
```

Which one of the following is correct for the query?

(a) The query is not legal.
(b) The query retrieves subject names for subjects where the maximum mark is greater than 60.
(c) The query retrieves subject names where the maximum average mark is greater than 60.
(d) The query retrieves subject names where the average mark is maximum of all subjects and is greater than 60.

27. Consider the following query:

```
SELECT AVG(mark)
FROM enrolment
WHERE subject_id = 'IT302'
```

Which one of the following values is returned by the above query if the mark values in IT302 are 90, 60 and NULL?

(a) 75 (b) 50 (c) NULL (d) Not defined

28. Consider the following query:

```
SELECT student_id
FROM enrolment
WHERE subject_id = 'IT1500'
AND subject_id = 'IT1200'
```

Which one of the following is correct?

(a) The query is not legal.
(b) The query retrieves student IDs of students doing both IT 1500 and IT1200.
(c) The query returns a NULL result.
(d) None of the above.

29. Consider the following query:

```
SELECT student_id
FROM enrolment
```

There are 100 students and 500 enrolments although 5 students have not yet enrolled in any course. The number of rows returned by the above query is:

(a) 100 (b) 500 (c) 95 (d) Unable to determine

30. Consider again the database in the last question and consider the following query that involves a join.

```
SELECT student_id
FROM enrolment, student
WHERE enrolment.student_id = student.student_id))
```

The number of rows obtained by running the above query is?

(a) 100 (b) 500 (c) 95 (d) 50,000

(e) Unable to determine

31. Consider the following query:

```
SELECT subject_name
FROM subject
WHERE subject_id IN
        (SELECT subject_id
        FROM enrolment
        WHERE student_id IN
                (SELECT student_id
                FROM student
                WHERE student_name = 'Prem Kumar'))
```

Which one of the following is correct about the query above?

(a) The query is not legal.

(b) The query retrieves all subject names of subjects in which Prem Kumar is enrolled.

(c) The query retrieves all subject names of subjects in which Prem Kumar is not enrolled.

(d) None of the above

32. Consider the query:

```
SELECT subject_name
FROM enrolment, student, subject
WHERE student.student_id = enrolment.student_id
AND enrolment.subject_id = subject.subject_id
AND student.name = 'Prem Kumar'
```

Which one of the following is true?

(a) The query is not legal.

(b) The query retrieves all subject names of subjects in which Prem Kumar is not enrolled.

(c) The query is equivalent to the last query.

(d) None of the above

33. Consider the query:

```
SELECT student_id
FROM enrolment
WHERE subject_id = 'IT302'
AND student_id IN
        (SELECT student_id
        FROM enrolment
        WHERE subject_id = 'IT304')
```

Which one of the following is true about the query above?

(a) The query is not legal.
(b) The query retrieves student IDs of students doing both IT302 and IT304.
(c) The query retrieves student IDs of students doing IT302 or IT304.
(d) The query returns a NULL result.
(e) None of the above

34. Consider the following query:

```
SELECT student_id
FROM enrolment e1
WHERE NOT EXISTS
        (SELECT *
        FROM enrolment e2
        WHERE e1.student_id = e2.student_id
        AND mark < 50))
```

Which one of the following is true about the query above?

(a) The query is not legal.
(b) The query returns the student ID of students who have a mark of below 50 in each subject that they are doing.
(c) The query returns the student ID of students who have a mark of below 50 in some subject that they are doing.
(d) The query returns the student ID of students who have a mark of 50 or above in each subject they are enrolled.

35. Consider the following query:

```
SELECT student_name
FROM student
WHERE NOT EXISTS
        (SELECT *
        FROM subject
        WHERE department = 'Comp. Science'
        AND NOT EXISTS
                (SELECT *
                FROM enrolment
                WHERE student.student_id = enrolment.student_id
                AND subject.subject_id = enrolment.subject_id))
```

Which one of the following is true about the query above?

(a) The query is not legal.
(b) The query returns the student name of all students who have passed all computer science subjects are doing.
(c) The query returns the student name of each student who is enrolled in every subject that is offered by the department of computer science.
(d) The query returns the student name of each student who is enrolled only in subjects offered by the department of computer science.

EXERCISES

1. Write SQL statements for the following queries on the company database and determine whether it's a correlated or noncorrelated query.
   (a) Retrieve the names of players whose maximum score is higher than the average score of all players.
   (b) Find the names of all employees who have a supervisor with the same birthday as theirs.
   (c) Find the names of all employees who are directly supervised by Franklin Wong.
   (d) Find the country of the player which has the highest maximum score in the database.
2. Formulate the following queries for the ODI database used in this chapter:
   (a) Find the player's last name who was the youngest amongst the players in the database when he played in his first test match.
   (b) Find the ID of the player who has made the highest number of runs in the example database.
   (c) Find the details of the ODI match in the example database where Tendulkar scored a century but India lost the match.
   (d) Find all the IDs of matches where India won the match.
   (e) Find the ID of the player who hit the largest number of sixes in an ODI match in the example database.
   (f) Find the ID of the player who had the highest strike rate in an ODI match in the example database.
   (g) Find the list of names of players who have scored a century in the example ODI database.
   (h) Find the average strike rate of players who have scored a century in the example ODI database.
   (i) Find the names of players who have taken four wickets or more in the example ODI database.
   (j) What is the maximum number of wickets that a bowler has taken in an ODI match in the example database? Find the ID of that bowler.
   (k) Find the name of the youngest player in the example database.
   (l) Find the IDs of players who have never scored a four or a six in all matches they have batted.
   (m) Find the ID of the player who scored the largest percentage of his runs in fours and sixes.
   (n) Find the ID of the player who has the highest average score in the matches he has batted. Ensure that *Not Out* scores are not counted in computing the average.

3. Consider the following two queries. The first query is as follows:

```
SELECT PID
FROM Batting b1
WHERE NOT EXISTS
        (SELECT *
        FROM Batting b2
        WHERE b1.PID = b2.PID
        AND (SCORE < 51))
```

The second query is given below.

```
SELECT PID
FROM Player
WHERE NOT EXISTS
        (SELECT *
        FROM Batting
        WHERE Player.ID = PID
        AND (NRuns < 51))
```

Explain the difference between the two queries.

4. Formulate in SQL the query 'Find IDs of players who have either bowled or batted (or did both) in the ODI match 2689'.
Is the formulation given below correct? Explain the reasons for your answer.

```
SELECT PID
FROM Batting
WHERE MatchID = '2689'
OR PID IN
        (SELECT PID
        FROM Bowling
        WHERE MatchID = '2689')
```

Formulate this query in two other ways and show that those formulations are correct. One of these formulations should use the table *Match* in addition to tables *Batting* and *Bowling?*

5. Consider the query 'Find the player IDs of players who have scored more than 30 in every ODI match that they have batted in the example database'. We now present two different SQL formulations for this query.
The first formulation is given below:

```
SELECT PID
FROM Batting b1
WHERE NOT EXISTS
        (SELECT *
        FROM Batting b2
        WHERE b1.PID = b2.PID
        AND NRuns < 31)
```

The second formulation is given below and it shows how the query may be formulated in another way.

```
SELECT PID
FROM Player p1
WHERE NOT EXISTS
        (SELECT *
        FROM Batting b1
        WHERE p1.PlayerID = b1.PID
        AND NRuns < 31)
```

Is there another way we can formulate this query in SQL? If yes, show the formulation.

Compare and contrast the weaknesses and strengths of the two formulations. Modify each of them to remove the weaknesses of the formulation.

6. Formulate a query in SQL using the left outer join to retrieve names of players that have never batted according to the ODI database.
7. Give an example of two relations that would give a meaningful result of carrying out a full outer join. Formulate the full join using SQL.
8. Explain a situation when embedded SQL would be useful rather than using a direct SQL query. Show how the embedded SQL will be used in that case.
9. Explain a situation when dynamic SQL would be useful. Show how the dynamic SQL will be helpful rather than using a direct SQL query.
10. List as many conditions as you can think of, where a view can be safely updated.

PROJECTS

1. Find out if there are other languages for querying a relational database systems. One possibility is to explore Query-by-Example or QBE that was designed at IBM. A version of it has been implemented in Microsoft Access. Study the language you have chosen and write a number of queries presented in this chapter using that language. If you are using QBE then you may wish to refer to: pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/qbe.pdf
2. Implement all the queries listed in this chapter using a DBMS that you can access. Present a report. List any errors.

LAB EXERCISE

1. Consider the following relational database in which *CName* is a company name:

```
Lives(Name, Street, City)
Works(Name, CName, Salary)
Located-in(CName, City)
manages(Name, Manager-name)
```

Formulate the following queries in SQL.

(a) Find the names of all people who work for Bank of India.
(b) Find the name and city of all people who work for Bank of India.
(c) Find the name, street and city of all people who work for Bank of India and earn more than Rs. 55,000.
(d) Find all people who live in the same city as the company they work for.
(e) Find all people who live in the same city and on the same street as their manager.
(f) Find all people who do not work for Bank of India.
(g) Find all people who earn more than every employee of Small Bank Corporation.
(h) Assume the companies are located in several cities. Find all companies located in every city in which Small Bank Corporation is located.
(i) Find all people who earn more than the average salary of people working in their company.
(j) Find the company employing the most people.
(k) Find the company with the smallest payroll.
(l) Find the companies that pay more, on average, than the average salary at Bank of India.
(m) Modify the database so that Agarwal now lives in Shastri Nagar.
(n) Give all employees of Bank of India a 10 percent raise.
(o) Give all managers a 10 percent raise.
(p) Give all managers a 10 percent raise unless the salary becomes greater than Rs. 58,000. In such cases, give only a 3 percent raise.
(q) Delete all rows in the *Works* table for employees of Bank of Bihar.

BIBLIOGRAPHY

For Students

A number of books listed below are readable and suitable for students. The book by Beaulieu is good for learners. Chapters in database textbooks can also be a good starting point to learn SQL. A fascinating account of people, projects and politics involved in developing SQL is given in the report by Paul McJones.

For Instructors

A number of books listed below are suitable for instructors. The book by Faroult and Robson is excellent. The book by Cumming and Russell and the one by Celko is good for an experienced SQL programmer. There are many other books on SQL, some of them deal with a specific system, for example, SQL Server, ORACLE, MySQL and so on.

Beaulieu, A., *Learning SQL*, O'Reilly Media, 2005, 306 pp.

Celko, J., *SQL for Smarties,* Third Edition, Morgan Kaufman, 2005, 808 pp.

Cumming, A., and G. Russell, *SQL Hacks,* O'Reilly Media, 2006, 410 pp.

Patrick, J. J., *SQL Fundamentals*, Third Edition, Prentice Hall, 2008, 832 pp.

McGrath, M., "SQL in Easy Steps", In *Easy Steps Limited*, Second Edition, 2005, 192 pp.

Bowman, J. S., S. L. Emerson, and M. Darnovsky, *The Practical SQL Handbook: SQL Variants*, Fourth Edition, Addison-Wesley, 2001, 512 pp.

Date, C. J., and H. Darwen, *A Guide to the SQL Standard*, Third Edition, Addition Wesley, 1993, 414 pp.

Earp, R., and S. Bagui, *Learning SQL: A step-by-step guide using Oracle*, Addison-Wesley, 2002, 368 pp.

Eisenberg, A., J. Melton, K. Kulkarni, J. Michels and F. Zemke, "SQL: 2003 has been published", *SIGMOD Record*, Vol. 33, No. 1, March 2004, pp. 119-126.

Faroult, S., and P. Robson, *The Art of SQL*, O'Reilly Media, 2006, 367 pp.

Gennick, J., SQL *Pocket Guide*, O'Reilly, 2003, 154 pp.

Groff, J. R., and Weinberg, P. N., *SQL: The Complete Reference*, Second Edition, McGraw-Hill, 2002, 1080 pp.

Gulutzan, P., and Pelzer, T., *SQL-99 complete, really.,* CMP Books, 1999, 1104 pp.

Kline, K., D. Kline, and B. Hunt, *SQL in a Nutshell: A Desktop Quick Reference*, Second Edition, O'Reilly, 2004, 700 pp.

Kreines, D. C., *Oracle SQL: The Essential Reference*, O'Reilly, 2000, 418 pp.

McJones, P. (Ed.), "The 1995 SQL Reunion: People, Projects, and Politics", SRC Technical Note, 1997-08, Digital Systems Research Center, Palo Alto, 1997. http://www.mcjones.org/System_R/SQL_Reunion_95/sqlr95.html

Tow, D., *SQL Tuning*, O'Reilly, 2003, 314 pp.

Viescas, J. L., and M. J. Hernandez, *SQL Queries for Mere Mortals: A Hands-On Guide to Data Manipulation in SQL*, Second Edition, Addison-Wesley, 2007, 672 pp.

Lans, R. F. van der, *Introduction to SQL: Mastering the Relational Database Language*, Fourth Edition, Addison-Wesley, 2006, 1056 pp.

Appendix 5.1 Player Table

PlayerID	*LName*	*FName*	*Country*	*YBorn*	*BPlace*	*FTest*
28001	Kohli	Virat	India	1988	Delhi	NULL
24007	Gambhir	Gautam	India	1981	Delhi	2004
28003	Raina	Suresh	India	1986	Ghaziabad	NULL
28004	Sharma	Rohit	India	1987	Nagpur	NULL
20007	Khan	Zaheer	India	1978	Shrirampur	2000
26005	Patel	Munaf	India	1983	Ikhar	2006
29020	Nayar	Abhishek	India	1983	Secunderabad	NULL
29022	Pathan	Yusuf	India	1982	Baroda	NULL
25007	Kulasekara	Nuwan	Sri Lanka	1982	Nittambuwa	2005
23007	Thusara	Thilan	Sri Lanka	1981	Balapitiya	2003
28005	Mendis	Ajantha	Sri Lanka	1985	Moratuwa	2008
27007	Warnapura	Malinda	Sri Lanka	1979	Colombo	2007
20005	Sangakkara	Kumar	Sri Lanka	1977	Matale	2000
97005	Jayawardene	Mahela	Sri Lanka	1977	Colombo	1997
26007	Kapugedera	Chamara	Sri Lanka	1987	Kandy	2006
99005	Dilshan	Tilakaratne	Sri Lanka	1976	Kalutara	1999
28007	Uthappa	Robin	India	1985	Coorg	NULL
23009	Pathan	Irfan	India	1984	Baroda	2003
26008	Chawla	Piyush	India	1988	Aligarh	2006
26009	Sreesanth	S	India	1983	Kothmangalam	2006
23005	Bracken	Nathan	Australia	1977	Penrith	2003
26006	Clark	Stuart	Australia	1975	Sutherland	2006
27008	Johnson	Mitchell	Australia	1981	Townsville	2007
28008	Hopes	James	Australia	1978	Townsville	NULL
24005	Clarke	Michael	Australia	1981	Liverpool	2004
94005	Hayden	Mathew	Australia	1971	Kingaroy	1994
25005	Hussey	Michael	Australia	1975	Morley	2005
23010	Butt	Salman	Pakistan	1984	Lahore	2003
21010	Malik	Shoaib	Pakistan	1982	Sialkot	2001
20010	Khan	Younis	Pakistan	1977	Mardan	2000
27010	Sohail	Tanvir	Pakistan	1984	Rawalpindi	2007
23011	Gul	Umar	Pakistan	1984	Peshawar	2003
26010	Anjum	Iftikhar	Pakistan	1980	Khanewal	2006
29010	Alam	Fawad	Pakistan	1985	Karachi	NULL
24010	Amla	Hashim	SouthAfrica	1983	Durban	2004
26010	Botha	Johan	South Africa	1982	Johannesburg	2006
98010	Ntini	Makhaya	South Africa	1977	Mdingi	1998
22010	Prince	Ashwell	South Africa	1977	Port Elizabeth	2002
22011	Smith	Graeme	South Africa	1981	Johannesburg	2002

Contd.

Appendix 5.1 Contd.

PlayerID	*LName*	*FName*	*Country*	*YBorn*	*BPlace*	*FTest*
98011	Flintoff	Andrew	England	1977	Preston	1998
25011	Pietersen	Kevin	England	1980	Pietermaritzburg	2005
24012	Strauss	Andrew	England	1977	Johannesburg	2004
24013	McCallum	Brendon	New Zealand	1981	Dunedin	2004
28010	Ryder	Jesse	New Zealand	1984	Masterton	2008
97010	Vettori	Daniel	New Zealand	1979	Auckland	1997
94010	Chanderpaul	Shivnarine	West Indies	1974	Unity Village	1994
20011	Gayle	Chris	West Indies	1979	Kingston	2000
20012	Sarwan	Ramnaresh	West Indies	1980	Wakenaam Island	2000
24013	Bravo	Dwayne	West Indies	1983	Santa Cruz	2004

Appendix 5.2 Batting Table

MatchID	*PID*	*Order*	*HOut*	*FOW*	*NRuns*	*Mts*	*Nballs*	*Fours*	*Six*
2755	23001	3	C	51	0	12	6	0	0
2755	25001	5	C	232	71	104	80	4	0
2755	91001	1	C	74	60	85	52	8	2
2755	94002	7	LBW	157	17	23	29	1	0
2755	92002	11	NO	NO	1	11	7	0	0
2689	89001	2	C	205	91	176	121	7	0
2689	23001	4	C	175	38	27	38	2	2
2689	25001	5	C	240	36	52	37	2	1
2689	99002	1	C	2	2	3	3	0	0
2689	95001	3	C	8	1	9	7	0	0
2689	24001	5	C	123	42	81	56	2	1
2689	99001	8	B	228	7	20	12	0	0
2689	27001	9	C	255	7	7	7	1	0
2755	27001	9	B	257	2	7	6	0	0
2717	98001	6	C	290	9	17	8	0	0
2717	21001	2	C	231	119	150	95	12	5
2717	23001	4	C	294	48	54	47	1	3
2717	25001	5	NO	NULL	26	38	30	0	1
2619	96001	1	C	150	53	105	60	7	1
2619	89001	2	C	156	94	120	81	16	1
2619	23001	4	C	209	18	28	24	1	0
2619	96001	5	C	216	4	3	5	0	0
2619	25001	6	B	294	35	50	37	3	0
2618	96001	1	C	150	59	119	79	7	2
2618	89001	2	C	116	71	87	59	13	0
2618	23001	4	C	266	72	72	57	10	2
2618	25001	5	C	311	24	60	17	3	0
2618	96001	6	RO	318	24	41	17	3	1

Appendix 5.3 Bowling Table

MatchID	*PID*	*NOvers*	*Maidens*	*NRuns*	*NWickets*
2689	99001	10	0	58	1
2689	24001	3	0	27	1
2689	23001	3	0	15	0
2755	94002	9	1	40	1
2755	92002	10	0	56	1
2755	91001	4	0	29	0
2755	23001	10	0	53	2
2689	23005	9	1	31	3
2689	26006	6	0	32	1
2689	27008	6	0	33	0
2689	28008	6	0	20	0
2689	24005	10	0	52	3
2689	27001	10	2	46	4
2689	26009	9	0	43	2
2689	23009	8.4	0	54	2
2689	98002	10	0	44	1
2689	26008	9	0	45	0
2755	25007	9	1	40	1
2755	23007	8.4	0	47	5
2755	28005	10	1	43	1
2755	27001	4	0	25	0
2755	20007	9.3	3	27	1
2755	26005	9	0	48	2
2755	98002	10	0	40	3
2755	28004	4	1	13	0

CHAPTER 6

Normalization

OBJECTIVES

- Discuss the problems that can occur in a database system due to poor database design.
- Describe the anomalies that can arise if there is information redundancy.
- Explain the concept of functional dependencies, closure and minimal cover.
- Discuss the concept of Normal Forms, in particular the Third Normal Form (3NF) and the Boyce-Codd Normal Form (BCNF) and their purpose.
- Present techniques for normalizing relations.
- Discuss the desirable properties of decompositions.
- Describe the concepts of the Fourth Normal Form (4NF) and the Fifth Normal Form (5NF).
- Explain the concept of inclusion dependency.

KEYWORDS

Redundancy, anomalies, insertion anomaly, deletion anomaly, update anomaly, functional dependency, normal forms, first normal form, 1NF, second normal form, 2NF, third normal form, 3NF, Boyce-Codd normal form, BCNF, multivalued dependency, join dependency, fourth normal form, 4NF, fifth normal form, 5NF, decomposition, lossless, dependency preservation, inclusion dependency.

A mind all logic is like a knife all blade. It makes the hand bleed that uses it.

Rabindra Nath Tagore

6.1 INTRODUCTION

In Chapter 3, we considered the primary features of the relational database model. We noted that relations that form a database must satisfy some properties, for example, relations have no duplicate rows, rows have no ordering associated with them, and each element in the relation is atomic. We have also so far assumed that each relation has a group of attributes that belong together. Relations that satisfy these basic requirements may still have some undesirable properties, for example, data redundancy and update anomalies. We illustrate these undesirable properties and study how relations may be transformed or decomposed (or *normalized*) to eliminate them. Most of these undesirable properties do not arise if database modeling has been carried out carefully using a technique like the Entity-Relationship model. It is still important to understand the techniques in this chapter to check the model that has been obtained is good and ensure that no mistakes have been made in modeling. The basis of this checking is called *functional dependency*, a formal mathematical constraint between the attributes of a relation. We will discuss the functional dependency theory in this chapter.

Of course, mistakes can be made in database modeling especially when the database is large and complex or one may, for some reason, carry out database schema design using techniques other than a modeling technique. For example, one could collect all the information that an enterprise possesses and build one giant table (often called the *universal relation*) to hold it. This bottom-up approach is likely to lead to a table that suffers from problems we have noted above. For example, the table is highly likely to have redundant information and update, deletion and insertion anomalies. Normalization of such large tables will then be essential to avoid (or at least minimize) these problems.

To summarize, normalization plays only a limited role in the database schema design if a top-down modeling approach like the Entity-Relationship approach is carefully used. Normalization can however play a major role when the bottom-up approach is used. Normalization is then essential to build appropriate tables to hold the information of the enterprise. Whatever approach is used, normalization helps in measuring the quality of the database design based on some theoretical concepts.

The basic bottom-up design methodology, not used as commonly as the top-down design methodology, is based on relationships between the attributes (that is why this approach is sometimes called *design by synthesis*) while the top-down design is based on identifying entities and relationships between them (that is why it is sometimes called *design by analysis*). Therefore constructing a relational schema using a bottom-up design follows quite a different approach and involves grouping of related attributes into relations. It therefore becomes important to analyse the result of an initial bottom-up design to ensure that the design is good and this can be done by using normalization.

Some aspects of design quality have been discussed in Section 2.8 previously. These included aspects like correctness, completeness, readability and self-explanation. Although those aspects of quality are obviously important, assessing some aspects of quality, for example, structure and semantics of the relational schema, of a database design can be carried out using the concept we will now discuss. The concept of *functional*

dependency depends on understanding the semantics of the data and can be very effective in determining if the grouping of attributes in a relational schema is appropriate. Functional dependencies deal with what information in a table is dependent on other information in that table.

This chapter is organized as follows. In the next section, an example table is presented to illustrate problems that can arise if the database is not carefully designed. A number of anomalies are discussed. A sample decomposition of the table is presented to show how normalization can overcome some of the anomalies. The theoretical basis of normalization using the concept of functional dependency is then defined in Section 6.3. We discuss how to reason with functional dependencies. This is then followed by a study of single-valued normalization in Section 6.4. It will be shown how dependencies information can be used to decompose tables when necessary without losing any information in the original tables. The section includes topics like the First Normal Form (1NF), the Second Normal Form (2NF), the Third Normal Form (3NF) and finally the Boyce–Codd Normal Form (BCNF). Section 6.5 describes some desirable properties of decomposition of relations. The following section deals with multivalued dependencies and the Fourth Normal Form (4NF) and the Fifth Normal Form (5NF). Section 6.7 briefly explains the concept of denormalization. Section 6.8 explains inclusion dependency which concludes the chapter.

6.2 POSSIBLE UNDESIRABLE PROPERTIES OF RELATIONS AND SCHEMA REFINEMENT

Let us consider the following table, which we will label *ODIplayers* that gives details of runs scored and wickets taken by ODI players during various ODI matches.

Table 6.1 The table *ODIplayers*

PID	*LName*	*Country*	*Team 1*	*Team 2*	*MID*	*Date*	*Ground*	*NRuns*	*NWickets*	*YBorn*	*Place*
89001	Tendulkar	India	Australia	India	2689	4/3/2008	MCG	91	NULL	1973	Mumbai
89001	Tendulkar	India	Australia	India	2689	4/3/2008	MCG	91	NULL	1973	Mumbai
23001	Yuvraj	India	Pakistan	India	2717	26/6/2008	Karachi	48	0	1978	Delhi
23001	Yuvraj	India	Pakistan	India	2717	26/6/2008	Karachi	48	0	1978	Delhi
21001	Sehwag	India	Pakistan	India	2717	26/6/2008	Karachi	119	NULL	1978	Delhi
99002	Gilchrist	Australia	Australia	India	2689	4/3/2008	Gabba	2	NULL	1971	Bellingen
95001	Ponting	Australia	Australia	India	2689	4/3/2008	Gabba	1	NULL	1974	Launces
99001	Lee	Australia	Australia	India	2689	4/3/2008	Gabba	7	1	1976	Wollongg
91001	Jayasuriya	Sri Lanka	Sri Lanka	India	2755	27/8/2008	Colombo	60	0	1969	Matara
27002	ISharma	India	Pakistan	India	2717	26/6/2008	Karachi	NULL	0	1988	Delhi
25001	Dhoni	India	Sri Lanka	India	2755	27/8/2008	Colombo	71	NULL	1981	Ranchi
95003	Kallis	South Africa	South Africa	Australia	2839	9/4/2009	Cape Town	70	0	1975	Cape Town
23001	Yuvraj	India	Pakistan	India	2717	26/6/2008	Karachi	48	0	1978	Delhi
24007	Gambhir	India	Australia	India	2689	4/3/2008	Gabba	15	NULL	1973	Mumbai

We briefly describe the meaning of each attribute name although we have used these attribute names before. *PID* is player ID, *LName* is player's last name (there is no space in the table for player's first name), *Country* is the name of the country that the player represents, *Team 1* and *Team 2* are the two teams playing the match (usually *Team 1* is the host and *Team 2* is the visiting team), *MID* is match ID, *Date* is the date on which the match was played, *Ground* is where the match was played, *NRuns* is the number of runs scored by the player, *NWickets* is the number of wickets taken by the player, *YBorn* is the year of birth of the player while *Place* is where he was born. The space on the book page is so limited that we have to cut short some long place names.

Note the use of NULL values in the table. A value is NULL only if the player did not take part in bowling or batting in a match. A value of 0 means that the player did bowl or bat but the result was 0.

The table *ODIplayers* in Table 6.1 satisfies the properties of a relation that we discussed in Chapter 3 and any such relation is said to be in *first normal form (or 1NF).* The main properties of relations concerned here are structural and therefore we note that all attribute values of 1NF relations must be atomic. We acknowledge that relations have other important properties like no ordering of rows, no duplicate rows and unique names of columns but in this chapter we are not particularly concerned with such properties.

As in Table 6.1, conceptually it is convenient to have all the information in one table since it is then easier to query the database. But the above table has the following undesirable features.

6.2.1 Redundant Information

One of the aims of a good database design is that redundancy of information should be eliminated. Redundancy exists when same data is stored in several places in the database. In some cases although there is no duplicate data it may be possible to derive certain information from other information in the database. Some information in Table 6.1 is being repeated, for example, information about players, like *LName*, *YBorn*, *Place*, are being repeated for Tendulkar and Yuvraj Singh. We cannot show too many rows that have repeated information but if the table was large, for example, consisting of thousands of rows of players' batting and bowling, then *LName* will be repeated many times for the same *PID*; *Team 1*, *Team 2* and *Ground* will be repeated for the same *MID*. Every time we wish to insert a batsman's score we must insert values of the columns *Team 1*, *Team 2*, as well as the name of the batsman, his year and place of birth, in addition to *PID* and *MID*. Repetition of information results in waste of storage as well as problems we discuss below.

6.2.2 Update Anomalies

Redundant information makes updates more difficult, for example, changing the name of the ground sounds strange but such things do happen. For example, the name of Bombay was changed to Mumbai. Such a change would require that all rows containing information about that ground be updated. If for some reason, all rows are not updated, we might have a database that gives two names for the same ground. This difficulty is called the *update anomaly*.

6.2.3 Insertion Anomalies: Inability to Represent Certain Information

Let the primary key of the above table be *PID, MID*. Any new row to be inserted in the table must have a value for the primary key since existential integrity requires that a key may not be totally or partially NULL.

However, if one wanted to insert the number and name of a new player in the database, for example, a new player who has been selected to tour overseas, it would not be possible until the player plays in a match and we are able to insert values of *PID* and *MID*. Similarly information about a new ODI match that is already scheduled cannot be inserted in the database until a player has played in the match. These difficulties are called *insertion anomalies*.

6.2.4 Deletion Anomalies: Loss of Useful Information

In some instances, useful information may be lost when a row is deleted. For example, if we delete the row corresponding to a player who happens to be the only player selected for a particular ODI, we will lose relevant information about the match (viz., ground name, date, etc). Similarly deletion of a match from the database may remove all information about a player who was playing in only that ODI. Such problems are called *deletion anomalies*.

The above problems arise primarily because the table *ODIplayers* has information about players, their batting and bowling performance as well as ODI matches. The semantics of the relation is not quite clear; is it about players? Is it about matches? Is it about players' performance in various matches? If we can resolve the semantics of the relations then none of the above anomalies would occur in them. One solution to deal with the problems is to decompose the table into two or more smaller tables.

Decomposition may provide further benefits. For example, in a distributed database different tables may be stored at different sites, if necessary. Of course, decomposition does increase the cost of query processing as the decomposed tables will need to be joined, sometimes frequently.

The above table may be decomposed into three tables as shown below in Tables 6.2 to 6.4 so that the semantics of each decomposed table are clearer. It has been noted earlier that Table 6.1 appears to include information about players, matches and players' performances. Therefore the decomposition is based on that understanding of the table and removes most of the undesirable properties listed above.

Player2(*PID, LName, Country, YBorn, Place*)
ODIMatch(*MID, Team1, Team2, Date, Ground*)
Performance(*PID, MID, NRuns, NWickets*)

Figure 6.1 The schema of decomposed Table 6.1

The decomposed tables are given below. Table 6.2 presents information about players.

Table 6.2 The decomposed table *Player2*

PID	*LName*	*Country*	*YBorn*	*Place*
89001	Tendulkar	India	1973	Mumbai
21001	Sehwag	India	1978	Delhi
99002	Gilchrist	Australia	1971	Bellingen
95001	Ponting	Australia	1974	Launces

Contd.

Table 6.2 Contd.

PID	*LName*	*Country*	*YBorn*	*Place*
99001	Lee	Australia	1976	Wollongg
91001	Jayasuria	Sri Lanka	1969	Matara
27002	ISharma	India	1988	Delhi
25001	Dhoni	India	1981	Ranchi
95003	Kallis	South Africa	1975	Cape Town
23001	Yuvraj	India	1978	Delhi
24007	Gambhir	India	1973	Mumbai

Table 6.3 presents information about matches.

Table 6.3 The decomposed table *ODIMatch*

MID	*Team 1*	*Team 2*	*Date*	*Ground*
2689	Australia	India	4/3/2008	MCG
2717	Pakistan	India	26/6/2008	Karachi
2689	Australia	India	4/3/2008	Gabba
2689	Australia	India	4/3/2008	Gabba
2689	Australia	India	4/3/2008	Gabba
2755	Srin Lanka	India	27/8/2008	Colombo
2717	Pakistan	India	26/6/2008	Karachi
2755	Sri Lanka	India	27/8/2008	Colombo
2839	South Africa	Australia	9/4/2009	Cape Town
2717	Pakistan	India	26/6/2008	Karachi
2689	Australia	India	4/3/2008	Gabba

Table 6.4 presents information about players' performance.

Table 6.4 The decomposed table *Performance*

PID	*MID*	*NRuns*	*NWickets*
89001	2689	91	NULL
21001	2717	119	NULL
99002	2689	2	NULL
95001	2689	1	NULL
99001	2689	7	1
91001	2755	60	0

Contd.

Table 6.4 Contd.

PID	*MID*	*NRuns*	*NWickets*
27002	2717	NULL	0
25001	2755	71	NULL
95003	2839	70	0
23001	2717	48	0
24007	2689	15	NULL

Such decomposition is called *normalization* and in some cases, for example in Table 6.1, decomposition is essential if we wish to overcome undesirable anomalies. The normalization process essentially assists in representing the data required for an application with as little redundancy as possible such that efficient updates of the data in the database are possible and useful data is not lost by mistake when some other data is deleted from the database.

As noted earlier, normalization can have an adverse effect on performance. Data which could have been retrieved from one table before normalization may require several tables to be joined after normalization. Normalization does however lead to more efficient updates since an update that may have required several rows to be updated before normalization could well need only one row to be updated after normalization.

Although in the above example we were able to look at the original table and intuitively propose a suitable decomposition that eliminates the anomalies that we discussed, in general this intuitive approach is not effective. A table may have one hundred or more columns and it is then almost impossible for a person to conceptualize all the information and suggest a suitable decomposition. We therefore need an algorithmic approach to find if there are problems in a proposed database design and how to eliminate them if they exist.

There are several stages of the normalization process. These are called the *first normal form (1NF)*, the *second normal form (2NF)*, the *third normal form (3NF)*, *Boyce–Codd normal form (BCNF)*, the *fourth normal form (4NF)* and the *fifth normal form (5NF)*. For all practical purposes, 3NF or the BCNF are quite adequate since they remove the anomalies discussed above for most common situations. It should be clearly understood that there is no obligation to normalize tables to the highest possible level. Performance should be taken into account and this may result in a decision not to normalize, say, beyond second normal form.

The normalization process involves applying a number of tests to each table that is being normalized. If the table does not meet the tests then it may be decomposed in two or more tables as illustrated above such that the decomposed tables meet the tests. The tests are essentially based on the concepts of functional dependency and primary keys of the tables which are discussed in Section 6.3.

It should be noted that all normal forms are additive. That is, if a table is in 3NF then it is by definition guaranteed to be in 2NF which in turn by definition guarantees 1NF.

Intuitively, the first normal form ensures that the table satisfies the structural properties required of a table in the relational model (in particular, all values are atomic). The second and third normal forms are designed to ensure that each table contains information about only one thing (either an entity or a relationship). That is, all nonkey columns in each table provide a fact about the entity or the relationship that is being identified by the key. Again, a sound E-R model of the database would ensure that all tables either provide facts about an entity or about a relationship resulting in tables that are obtained from an E-R model being in 3NF.

It should be noted that decomposition of each table always has to be based on principles that ensure that the original table may be reconstructed from the decomposed tables if and when necessary. If we are able to reduce redundancy and not lose any information, this implies that all redundant information can be derived given the other information in the database. Therefore information that has been removed must be related or dependent on other information that still exists in the database; this is why the concept of redundancy is important. Careless decomposition of a table can result in loss of information. We will discuss this in detail later in Section 6.5.

To define the normal forms more formally, we first need to define the concept of functional dependency.

6.3 FUNCTIONAL DEPENDENCY—SINGLE-VALUED DEPENDENCIES

A database is a collection of related information and it is therefore inevitable that some items of information in the database would depend on some other items of information. The information is either single-valued or multivalued. The name of a person or his/her date of birth are single-valued facts (we are not concerned that the date of birth has three components, the day, the month and the year). Also, qualifications of a person or courses that an instructor teaches are multivalued facts assuming the instructor teaches more than one course. We first deal only with single-valued facts and discuss the important concept of functional dependency.

The concept of some information being dependent on other information is quite basic. For example, at any time in a database, a student number would always determine the student name (a person may change his/her name but at any one time the person's name can be determined by the student number) and a course with a given course number would always have the same course name. We formalize this concept.

6.3.1 Functional Dependency

Functional dependency (FD) is an important concept in designing a database or assessing a given database design. It is a constraint between two sets of attributes of a relation. The idea of functional dependency is a relatively simple one. Consider a table R that has two attributes A and B. The attribute B of the table is *functionally dependent* on attribute A if and only if for each value of A no more than one value of B is associated. Functional dependency therefore may be considered a kind of an integrity constraint.

Definition—Functional Dependency (FD)
Let a table R have two attributes A and B. The attribute B of the table is functionally dependent on attribute A (that is, $A \rightarrow B$) if and only if for each value of A no more than one value of B is associated.

In other words, the value of column A uniquely determines the value of B and if t_1 and t_2 are two rows in the table R and $t_1(A) = t_2(A)$ then we must have $t_1(B) = t_2(B)$.

The attributes A and B need not be single columns. They could be any subsets of the columns of the table R (possibly single columns). If B is functionally dependent on A (or A functionally determines B), we may then write

$$R.A \rightarrow R.B$$

The set of attributes $R.A$ are called the left-hand side of the functional dependency (FD) while $R.B$ is called the right-hand side of the FD. Each FD in a relation reveals something about the semantics of the relation.

Note that functional dependency does not imply a one-to-one relationship between A and B (since one value of B may be related to more than one value of A) although a one-to-one relationship may exist between A and B.

A simple example of the above functional dependency is when A is a primary key of an entity (e.g., PID) and B is some single-valued property or attribute of the entity (e.g., date of birth). $A \rightarrow B$ then must always hold.

Functional dependencies also arise in relationships. Let C be the primary key of an entity and D be the primary key of another entity. Let the two entities have a relationship with a primary key (C, D) since they are the keys of the two relations. If the relationship is one-to-one, we must have $C \rightarrow D$ and $D \rightarrow C$. If the relationship is many-to-one, we would have $C \rightarrow D$ but not $D \rightarrow C$. For many-to-many relationships, no functional dependencies hold. For example, if C is player number and D is ODI match number, there is no functional dependency between them.

If we consider the table *Batting* which includes the number of runs scored by each batsman, then we may write

$$(PID, MID) \rightarrow NRuns$$

For example, in the *ODIplayers* in Table 6.1 the following functional dependencies exist

$$PID \rightarrow Lname$$
$$PID \rightarrow Country$$
$$MID \rightarrow Ground$$

These functional dependencies imply that there can only be one last name for each *PID*, only one country for each player and only one ground name for each *MID*. It is of course possible that several players may have the same name and several players have played in the same ODI match. If we consider $MID \rightarrow Ground$, the dependency implies that no ODI match can have more than one ground. Functional dependencies therefore place constraints on what information the database may store. In the above example, one may be wondering if the following functional dependencies hold

$$Lname \rightarrow PID$$
$$Ground \rightarrow MID$$

Certainly there is nothing in the instance of the example database presented in Table 6.1 above that contradicts the first of these functional dependencies. However, whether this FD holds or not would depend on whether the database we are considering allows duplicate player last names (for example, Singh). The second FD does not hold because it implies that a cricket ground cannot have more than one ODI match which does not make a lot of sense.

Functional dependencies arise from the nature of the real world that the database models. Often A and B are facts about an entity where A might be some identifier for the entity and B some characteristic. Functional

dependencies cannot be automatically determined by studying one or more instances of a database. They can be determined only by a careful study of the real world and a clear understanding of the semantics of the attributes in the table.

We have noted above that the definition of functional dependency does not require that A and B be single attributes. In fact, A and B may be collections of attributes. For example:

$$(MID) \rightarrow (Team1, Team2)$$

When dealing with a collection of columns, the concept of *full functional dependence* is important and can be defined as follows:

Definition—Full Functional Dependence
Let A and B be distinct collections of columns from a table R and let R.A → R.B. B is then fully functionally dependent on A if B is not functionally dependent on any subset of A.

The above example of batsmen and ODI matches would show full functional dependence if runs scored by a batsman are not functionally dependent on either player number (*PID*) or ODI match number (*MID*) alone. This implies that we are assuming that a player may have played in more than one ODI match and an ODI match would normally have more than one batsman. Furthermore, it has been assumed that there is at most one number of runs value of each batsman in the same ODI match inning.

The above example illustrates full functional dependence. However, the following functional dependency

$$(PID, MID) \rightarrow Ground$$

is not full functional dependence because $MID \rightarrow Ground$ holds.

As noted earlier, the concept of functional dependency is related to the concept of the candidate key of a table as the candidate key is an identifier which uniquely identifies a row and therefore determines the values of all other columns in the table. Therefore, for any subset X of the columns of a table R that satisfies the property that all remaining columns of the table are functionally dependent on it (that is, on X), X is a candidate key as long as no column can be removed from X and still satisfy the property of functional dependence. In the example above, the columns (*PID, MID*) form a candidate key (and the only one) of the table *ODIPlayers* since they functionally determine all the remaining columns.

As noted earlier, functional dependence is an important concept and a large body of formal theory has been developed about it. We discuss the concept of *closure* that helps us derive all functional dependencies that are implied by a given set of dependencies. Once a complete set of functional dependencies has been obtained, we will study how these may be used to build normalized tables.

6.3.2 Closure

We first define the concept of closure.

Definition—Closure
Let a table R have some functional dependencies F specified. The closure of F (usually written as F^+) is the set of all functional dependencies that may be logically derived from F.

Often F is the set of the most obvious and important functional dependencies and F^+, the closure, is the set of all the functional dependencies, including *F,* that can be deduced from F. The closure is important and may, for example, be needed in finding one or more candidate keys of the table.

We will use the example table in Table 6.1 given below as a relational schema.

ODI players (PID, LName, Country, Team1, Team2, MID, Date, Ground, NRuns, NWickets, YBorn, Place).

This relation has the functional dependencies listed in Fig. 6.2.

Let these dependencies be denoted by F. We can derive the closure of F. We first discuss the Armstrong's axioms that assist in deriving the closure.

PID → *LName*
PID → *Country*
MID → *Team1*
MID → *Team2*
PID → *YBorn*
PID → *Place*
MID → *Date*
MID → *Ground*
(*PID, MID*) → *NRuns*
(*PID, MID*) → *NWickets*

Figure 6.2 Functional dependencies in table *ODIplayers*

6.3.3 Armstrong's Axioms

To determine F^+, we need rules for deriving all functional dependencies that are implied by F. A set of rules that may be used to infer additional dependencies was proposed by Armstrong in 1974. These rules (or axioms) are a complete set of rules in that all possible functional dependencies may be derived from them. The rules are as follows:

1. *Reflexivity Rule*—If X is a set of attributes and $Y \subseteq X$, then $X \rightarrow Y$ holds.
   The reflexivity rule is the simplest (almost trivial) rule. It states that each subset of X is functionally dependent on X.
2. *Augmentation Rule*—If $X \rightarrow Y$ holds and W is a set of attributes, then $WX \rightarrow WY$ holds.
   The augmentation rule therefore is also quite simple. It states that if Y is determined by X then a set of columns W and Y together will be determined by W and X together. Note that we use the notation WX to mean the collection of all columns in W and X and write WX rather than the more conventional (W, X) for convenience.
3. *Transitivity Rule*—If $X \rightarrow Y$ and $Y \rightarrow Z$ hold, then $X \rightarrow Z$ holds.
   The transitivity rule therefore states that if X functionally determines Y and Y functionally determines Z, then X functionally determines Z. Transitivity is perhaps the most important rule.

These rules are called *Armstrong's Axioms*.

Further axioms may be derived from the above although the above three axioms are *sound and complete* as they do not generate any incorrect functional dependencies (soundness) and generate all possible functional dependencies that can be inferred from F (completeness). The most important additional axioms are

1. *Union Rule*—If $X \rightarrow Y$ and $X \rightarrow Z$ hold, then $X \rightarrow YZ$ holds.
2. *Decomposition Rule*—If $X \rightarrow YZ$ holds, then so do $X \rightarrow Y$ and $X \rightarrow Z$.
3. *Pseudotransitivity Rule*—If $X \rightarrow Y$ and $WY \rightarrow Z$ hold then so does $WX \rightarrow Z$.

Based on the above axioms and the functional dependencies specified for table *ODIplayers*, we may write additional functional dependencies. Some of these are given in Fig. 6.3.

(*PID, MID*) → *PID*	Reflexivity Rule
(*PID, MID*) → *MID*	Reflexivity Rule
PID → (*LName, Country*)	Union Rule
MID → (*Team1, Team2*)	Union Rule
PID → (*YBorn, Place*)	Union Rule
MID → (*Date, Ground*)	Union Rule

Figure 6.3 Additional functional dependencies

Often a very large list of dependencies can be derived from a given set F since Rule 1 itself will lead to a large number of dependencies. Since we have twelve attributes (*PID, LName, Country, Team1, Team2, MID, Date, Ground, NRuns, NWickets, YBorn, Place*) there are 4096 (that is, 2^{12}) subsets of these 12 attributes. These 4096 subsets could form 4096 values of X in functional dependencies of the type $X \rightarrow Y$. Of course, each value of X will then be associated with a number of values for Y (Y being a subset of X) leading to several thousand dependencies. These numerous dependencies are not particularly helpful in achieving our aim of normalizing tables.

Although we could follow the present procedure and compute the closure of F to find all the functional dependencies, the computation requires exponential time and the list of dependencies is often very large and therefore not very useful. There are two possible approaches that can be taken to avoid dealing with the large number of dependencies in the closure. One is to deal with one attribute or set of attributes at a time and find its closure (i.e., all functional dependencies relating to them). The aim of this exercise is to find what attributes depend on a given set of attributes and therefore ought to be together. The other approach is to find the *minimal covers*. We will discuss both approaches briefly.

As noted earlier, we do not need to deal with the large number of dependencies that might arise in a closure as often one is only interested in determining closure of a set of attributes given a set of functional dependencies. Closure of a set of attributes X is all the attributes that are functionally dependent on X given some functional dependencies F while the closure of F is all functional dependencies that are implied by F. Computing the closure of a set of attributes is a much simpler task if we are dealing with a small number of attributes. We will denote the closure of a set of attributes X given F by X^+.

6.3.4 Minimal Cover

We now define the concept of minimal cover.

Definition—Minimal Cover

The minimal cover for a set of functional dependencies F is given by a set of functional dependencies E such that:

1. *Every dependency in E is of the form X → Y where Y is a single attribute.*
2. *No dependency X → Y can be replaced with a dependency Z → Y where Z is a proper subset of X and still have a set of dependencies that is equivalent to E.*
3. *No dependency from E may be removed and still have a set of dependencies that is equivalent to E.*

The minimal cover of a set of functional dependencies is minimal in two ways as defined above. Firstly, according to requirement 1 above, each dependency is as small as possible such that the RHS is a single attribute and the LHS cannot have any attribute removed from it, according to Rule 2. Secondly, each FD in the minimal cover is required and thus there are no redundancies.

Algorithm to Obtain Minimal Cover

A simple algorithm may now be devised for obtaining minimal cover of FDs F based on the definition above. It is given below

1. Transform each functional dependency in F to one or more equivalent FDs with RHS having only one attribute.
2. Minimize the LHS of each FD obtained in Step 1.
3. Delete redundant FDs from the list of FDs obtained in Step 2.

For every set of FDs there always exists an irreducible set of FDs.

We now define the concept of equivalence of functional dependencies.

Equivalent Functional Dependencies

Let F_1 and F_2 be two sets of functional dependencies. The two functional dependencies are called *equivalent* if $F_1^+ = F_2^+$. Of course, it is not always easy to test that the two sets are equivalent as each of them may consist of hundreds of functional dependencies. One way to carry out the checking would be to take each dependency $X \rightarrow Y$ in turn from F_1^+ and check if it is in F_2^+.

Irreducible Set of FDs

We now define the concept of an irreducible set of functional dependencies. A set of FDs is irreducible if and only if the set satisfies the following properties:

1. The right-hand side of every FD is just one attribute.
2. The left-hand side is irreducible (that is, no attribute from the left-hand side may be omitted without changing the meaning of the set).
3. No FD from the set of functional dependencies may be discarded without changing the content of the set.

Now that we have the necessary background in functional dependencies, we may define the normal forms. Since normalization may involve decomposition of tables as shown above, the concept of functional dependencies is important in ensuring decompositions which preserve dependencies.

6.4 SINGLE-VALUED NORMALIZATION

Codd (1972) presented three normal forms (1NF, 2NF and 3NF) all based on functional dependencies among the attributes of a table. Later Boyce and Codd proposed another normal form called the Boyce–Codd normal form (BCNF). The fourth and fifth normal forms are based on multivalue and join dependencies and were proposed later.

We now discuss the normal forms and the tests which each table must meet. It should be remembered that the primary objective of normalization is to avoid some of the anomalies that we discussed at the beginning of the chapter.

6.4.1 The First Normal Form (1NF)—Atomic Attribute Values

A table satisfying the properties of a relation as discussed in Chapter 3 is said to be in first normal form. As discussed in Chapter 3, a relation cannot have multivalued or composite attributes. The 1NF requires the same condition.

Definition—1NF
A relation is in 1NF if and only if all underlying domains contain atomic values only.

The first normal form deals only with the basic structure of the table and does not resolve the problems of redundant information or the anomalies discussed earlier. All tables discussed in this chapter are in 1NF. Some tables may have multiple valued attributes but such tables must be modified to meet the requirements of the relational model. The modified tables are then in *first normal form*.

For example, Table 6.1 presented earlier is in 1NF since all attribute values in this table are atomic and indivisible. For example, a 1NF table could not have included more than one value of *NRuns*.

PID	*LName*	*Country*	*Team 1*	*Team 2*	*MID*	*Date*	*Ground*	*NRuns*	*NWickets*	*YBorn*	*Place*
89001	Tendulkar	India	Australia	India	2689	4/3/2008	MCG	91	NULL	1973	Mumbai
21001	Sehwag	India	Pakistan	India	2717	26/6/2008	Karachi	119	NULL	1978	Delhi
99002	Gilchrist	Australia	Australia	India	2689	4/3/2008	Gabba	2	NULL	1971	Bellingen
95001	Ponting	Australia	Australia	India	2689	4/3/2008	Gabba	1	NULL	1974	Launces
99001	Lee	Australia	Australia	India	2689	4/3/2008	Gabba	7	1	1976	Wollongg
91001	Jayasuria	Sri Lanka	Sri Lanka	India	2755	27/8/2008	Colombo	60	0	1969	Matara
27002	ISharma	India	Pakistan	India	2717	26/6/2008	Karachi	NULL	0	1988	Delhi
25001	Dhoni	India	Sri Lanka	India	2755	27/8/2008	Colombo	71	NULL	1981	Ranchi
95003	Kallis	South Africa	South Africa	Australia	2839	9/4/2009	Cape Town	70	0	1975	Cape Town
23001	Yuvraj	India	Pakistan	India	2717	26/6/2008	Karachi	48	0	1978	Delhi
24007	Gambhir	India	Australia	India	2689	4/3/2008	Gabba	15	NULL	1973	Mumbai

Table 6.1 is in 1NF as long as every attribute is considered an atomic value. For example, if one of the attributes was a player's date of birth, then it could not be considered as consisting of three components (*day, month, year*) for it to be considered atomic.

The above table of course suffers from all the anomalies that we have discussed earlier and needs to be normalized. We first discuss the *second normal form* (2NF) which will resolve most of the anomalies.

6.4.2 The Second Normal Form (2NF)—Candidate Keys

The second normal form attempts to deal with the problems that were identified in Table 6.1. The aim of the second normal form is to ensure that all information in one table is only about one thing.

Definition—2NF
A relation is in 2NF if it is in 1NF and every nonkey (also called nonprime) attribute is fully functionally dependent on each candidate key of the relation.

To understand the above definition of 2NF, we need to define the concept of *key* attributes. Each attribute of a table that participates in at least one candidate key is a *key* attribute of the table. All other attributes are called *nonkey* attributes.

The concept of 2NF requires that all attributes that are not part of a candidate key be *fully* dependent on each candidate key. Being fully dependent requires that a functional dependency $A \rightarrow B$ must be such that it if an attribute from A is removed (assuming A to be a composite attribute) then the dependency does not hold. This condition essentially involves A being a candidate key (often the primary key) and B being any of the nonkey attributes.

If we consider the table

ODIplayers(PID, LName, Country, Team 1, Team 2, MID, Date, Ground, NRuns, Nwickets, YBorn, Place)

and the functional dependencies below, given earlier in Fig. 6.3.

$PID \rightarrow LName$
$PID \rightarrow Country$
$MID \rightarrow Team1$
$MID \rightarrow Team2$
$PID \rightarrow YBorn$
$PID \rightarrow Place$
$MID \rightarrow Date$
$MID \rightarrow Ground$
$(PID, MID) \rightarrow NRuns$
$(PID, MID) \rightarrow NWickets$

Let us assume that (*PID, MID*)[1] is the only candidate key (and therefore the primary key), the table is not in 2NF since *LName, Team 1, Team 2* and *Ground* are not fully dependent on the key. To resolve difficulties we could remove those attributes from the table *ODIplayers* that are not fully dependent on the candidate keys of the table and put them in a separate table. Therefore we decompose the table into a number of tables based on our understanding that the table includes information about players as well as about ODI matches. We now

1. Does the relation *ODIplayers* have only one candidate key (*PID*, *MID*)? Look carefully and we may discover others. For example, could (*PID*, *Date*, *Ground*) be a candidate key? What about (*LName*, *MID*)?

have a table about players, another about ODI matches and finally another about the relationship between the two, resulting in the following decomposition of the original table:

Player2 (*PID, Lname, YBorn, Place, Country*)
ODI (*MID, Team1, Team2, Date, Ground*)
Performance (*PID, MID, NRuns, NWickets)*

We may recover the original table by taking the natural join of the three tables.

6.4.3 The Third Normal Form (3NF)

Although transforming a table that is not in 2NF into a number of tables that are in 2NF removes many of the anomalies that appear in the table that was not in 2NF, but not all anomalies are removed and further normalization may sometimes be needed to ensure removal of all anomalies. These anomalies arise because a 2NF table may have attributes that are not directly related to the thing that is being described by the candidate keys of the table. Let us first define 3NF.

Definition—3NF
A relation R is in third normal form (3NF) if it is in 2NF and every nonkey attribute of R is nontransitively dependent on each candidate key of R.

To understand the third normal form, we need to define *transitive dependence* which is based on one of Armstrong's axioms. Let A, B and C be three attributes of a table R, such that $A \rightarrow B$ and $B \rightarrow C$. From these functional dependencies, we may derive $A \rightarrow C$. As noted earlier, this dependence $A \rightarrow C$ is *transitive*.

The 3NF differs from the 2NF as all nonkey attributes in 3NF are required to be *directly* dependent on each candidate key of the table. The 3NF therefore insists that all facts in the table are about the key (or the thing that the key identifies), the whole key and nothing but the key. If some attributes are dependent on the key's transitivity then that is an indication that those attributes provide information not about the key but about a nonkey attribute. So the information is not directly about the key, although it obviously is related to the key.

The tables we get after decomposing for 2NF actually meet the test of 3NF and are therefore in 3NF.

To illustrate the concept of 3NF, consider the following tables that are slight modifications of those we got after normalization to 2NF. We have added the ground capacity to the table ODI. The other tables have not been modified.

Player2 (*PID, Lname, YBorn, Place, Country*)
ODI (*MID, Team1, Team2, Date, Ground, Capacity*)
Performance (*PID, MID, NRuns, NWickets)*

The following new functional dependency now exists:

Ground $\rightarrow$ *Capacity*

This FD may be used to derive the following FD:

MID $\rightarrow$ *Capacity*

Since the FD *MID* $\rightarrow$ *Ground* holds. Because we can derive *MID* $\rightarrow$ *Capacity* from the above functional dependencies, the above table is in 2NF. The table is however not in 3NF since *Capacity* is not directly

dependent on the primary key of the relation which is *MID*. This transitive dependence is an indication that the table has information about more than one thing (viz., ODI and *Ground*) and should therefore be decomposed. The primary difficulty with the above table is that a ground will have many matches played in it and therefore the capacity will need to be repeated many times. This leads to all the problems that we identified at the beginning of this chapter. To overcome these difficulties we need to decompose the table ODI given above into the following two tables:

ODI1(*MID, Team1, Team2, Date, Ground*)
Ground1(*Ground, Capacity*)

ODI1 is now in 3NF and so is *Ground1*. The primary keys of the two relations are *MID* and Ground respectively.

An alternative decomposition of the table *ODI* is possible:

A (*MID, Team1, Team2, Date*)
B (*Ground, Capacity*)
C (*MID, Ground*)

The decomposition into three tables is not necessary as the original table is based on the assumption that each ODI is played on only one ground.

Usually the 3NF is quite adequate for most relational database designs. There are however some situations, for example, the table *Player(PID, Lname, YBORN, MID)* discussed in 2NF above, where 3NF may not eliminate all the redundancies and inconsistencies. The problem with the table *Player(PID, LName, YBORN, MID)* is because of the redundant information in the candidate keys. This is resolved by further normalization using the BCNF.

6.4.4 The Boyce–Codd Normal Form (BCNF)

Consider the following table:

Player3(*PID, LName, MID, Date, Ground*)

Player3 has all attributes participating in candidate keys as all the attributes are assumed to be unique. We have assumed *LName* to be unique and assumed (*Date, Ground*) to be also unique since only one match can be held on a ground on a particular day, so it is a reasonable assumption. We therefore have the following candidate keys:

(*PID, MID*)
(*LName, MID*)
(*PID, Date, Ground*)
(*LName, Date, Ground*)

Since the table has no nonkey attributes, the table is in 2NF and also in 3NF, in spite of the table suffering the problems that we discussed at the beginning of this chapter.

The difficulty in this table is being caused by dependence within the candidate keys. The second and third normal forms assume that all attributes not part of the candidate keys depend on the candidate keys but do not deal with dependencies *within* the keys. BCNF deals with such dependencies.

Definition—BCNF
A relation R is said to be in BCNF whenever X → A holds in R, and A is not in X, then X is a candidate key for R.

It should be noted that most tables that are in 3NF are also in BCNF. Infrequently, a 3NF table is not in BCNF and this happens only if

(a) the candidate keys in the table are composite keys (that is, the keys are not single columns),
(b) there is more than one candidate key in the table, and
(c) the keys are not disjoint, that is, some columns in the keys are common.

Consider for example, the following relationship again:

Player3(PID, LName, MID, Date, Ground)

We now present the functional dependencies in this table.

PID → LName
MID → Date
MID → Ground

The table is in 3NF but not in BCNF because there are dependencies where attributes that are part of a candidate key are dependent on part of another candidate key. Such dependencies indicate that although the table is about some entity or association that is identified by the candidate keys, for example *(PID, MID)*, there are attributes that are not about the whole thing that the keys identify. For example, the above table is about an association (an ODI match) between players and matches and therefore the table needs to include only one identifier to identify players and only one identifier to identify matches. Providing three identifiers about players *(PID, LName, YBorn)* means that some information about players that is not needed is being provided. This provision of information will result in repetition of information and may lead to the anomalies that we discussed at the beginning of this chapter. If we wish to include further information about players and matches in the database, it should not be done by putting the information in the present table but by creating new tables that represent information about the entities *player* and *match.*

These difficulties may be overcome by decomposing the above table in the following three tables:

(*PID, LName*)
(*PID, MID*)
(*MID, Date, Ground*)

We now have a table that only has information about players, another only about matches and a third one about relationship between players and the matches they have played. All the anomalies and repetition of information have been removed.

BCNF has therefore ensured that no redundancy can be detected in the relation. Given the FDs for the relation being normalized, BCNF is the one that ensures no redundancy. 3NF in fact allows some redundancy within the composite candidate keys. It should be noted that although FDs give a clear signal that a 3NF relation has redundancies, it does not always point to a unique BCNF decomposition.

Summary of Normal Forms

Table 6.5 Summary of normal forms 1NF, 2NF, 3NF and BCNF

Normal Form	*Test*	*Remedy*
First Normal Form (1NF)	A relation is in 1NF if and only if all underlying domains contain atomic values only.	Ensure that all values are atomic
Second Normal Form (2NF)	A relation is in 2NF if it is in 1NF and every nonkey (also called nonprime) attribute is fully functionally dependent on each candidate key of the relation.	Decomposition based on nonkey attributes that are not fully dependent on the keys
Third Normal Form (3NF)	A relation R is in third normal form (3NF) if it is in 2NF and every nonkey attribute of R is nontransitively dependent on each candidate key of R.	Decomposition based on transitive FDs
Boyce–Codd Normal Form (BCNF)	A relation R is said to be in BCNF whenever $X \rightarrow A$ holds in R, and A is not in X, then X is a candidate key for R. This normally happens when a relation has more than one candidate key and the keys are composite.	Decomposition based on FDs within composite keys.

6.5 DESIRABLE PROPERTIES OF DECOMPOSITIONS

So far our approach has involved looking at individual tables and checking if they belong to 2NF, 3NF or BCNF. If a table was not in the normal form that was being checked for and we wished the table to be normalized to that normal form so that some of the anomalies could be eliminated, it was necessary to decompose the table into two or more tables. The process of decomposition of a table R into a set of tables $R_1, R_2, \ldots R_n$ was based on identifying different FDs and using that as a basis of decomposition, the decomposed tables $R_1, R_2, \ldots R_n$ are projections of R and are of course not disjoint, otherwise the glue holding the information together would be lost. Decomposing tables in this way based on a recognise and split method is not a particularly sound approach as we do not even have a basis to determine that the original table can be constructed if necessary from the decomposed tables. We now briefly discuss desirable properties of good decomposition and identify difficulties that may arise if the decomposition is done without adequate care. The next section will discuss how such decomposition may be derived given the functional dependencies.

Desirable properties of decomposition are as follows:

1. Attribute preservation
2. Lossless-join decomposition
3. Dependency preservation
4. Lack of redundancy

These properties are discussed in detail ahead.

6.5.1 Attribute Preservation

This is a simple and an obvious requirement that involves preserving all the columns that were present in the table that was decomposed.

Definition—Attribute Preservation
Therefore, decomposition of a table R into a set of tables R_1, R_2, KR_n must ensure that each attribute of R will appear in at least one relation schema R_i so that no attribute of R is lost. This is the property called attribute preservation.

6.5.2 Lossless-Join Decomposition

So far we have normalized a number of tables by decomposing them. We sometimes decomposed them intuitively but we used functional dependencies. We need a better basis for deciding decompositions since intuition may not always be correct. We illustrate how a careless decomposition may lead to problems including loss of information.

As a new example, consider the following table:

enrol(*sno, cno, date-enrolled, room-no, instructor*)

Consider decomposing the above table into two tables *enrol1* and *enrol2* as follows:

enrol1 (*sno, cno, date-enrolled*)
enrol2 (*date-enrolled, room-no, instructor*)

There are problems with this decomposition but we wish to focus on one aspect at the moment. Let an instance of the table *enrol* be as in Table 6.6.

Table 6.6 Table *enrol* before decomposition

S.no.	*C.no.*	*Date-enrolled*	*Room no.*	*Instructor*
830057	CP303	1FEB2004	MP006	Pranay Bose
820159	CP302	10JAN2004	MP006	Krishna Gupta
825678	CP304	1FEB2004	CE122	Prem Kumar
826789	CP305	15JAN2004	EA123	Inderjit Singh

Let the decomposed relations *enrol1* and *enrol2* be as given in Tables 6.7 and 6.8.

Table 6.7 Table *enrol1*

S.no.	*C.no.*	*Date-enrolled*
830057	CP303	1FEB2004
820159	CP302	10JAN2004
825678	CP304	1FEB2004
826789	CP305	15JAN2004

Table 6.8 Table *enrol2*

Date-enrolled	*Room no.*	*Instructor*
1FEB2004	MP006	Pranay Bose
10JAN2004	MP006	Krishna Gupta
1FEB2004	CE122	Prem Kumar
15JAN2004	EA123	Inderjit Singh

All the information that was in the table *enrol* appears to be still available in *enrol1* and *enrol2* in Tables 6.7 and 6.8 but this is not so. Suppose we wanted to retrieve the student numbers of all students taking a course from Prem Kumar, we would need to join *enrol1* and *enrol2*. The join would have six rows as follows:

Table 6.9 Join of *enrol1* and *enrol2*

S.no.	*C.no.*	*Date-enrolled*	*Room no.*	*Instructor*
830057	CP303	1FEB2004	MP006	Pranay Bose
830057	CP303	1FEB2004	CE122	Prem Kumar
820159	CP302	10JAN2004	MP006	Krishna Gupta
825678	CP304	1FEB2004	MP006	Pranay Bose
825678	CP304	1FEB2004	CE122	Prem Kumar
826789	CP305	15JAN1984	EA123	Inderjit Singh

The join contains two spurious rows that were not in the original table *enrol* in Table 6.6. Because of these additional rows, we have lost information about which students take courses from Prem Kumar. (Yes, we have more rows but less information because we are unable to say with certainty who is taking courses from Prem Kumar). Such decompositions are called *lossy* decompositions. A *nonloss* or *lossless* decomposition is that which guarantees that the join will result in exactly the same table as was decomposed. The join must not produce spurious rows. One might think that there might be other ways of recovering the original table from the decomposed tables but no other operators can recover the original table if the join does not.

We need to analyse why some decompositions are lossy. The common attribute in the above decomposition was *date-enrolled*. The common attribute is the glue that gives us the ability to find the relationships between different tables by joining the tables together. If the common attribute is not unique, the relationship information is not preserved. If each row had a unique value of *date-enrolled*, the problem of losing information would not have existed. The problem arises because several enrolments may take place on the same date.

A decomposition of a table R into tables $R_1, R_2, \ldots R_n$ is called a *lossless-join* decomposition with respect to functional dependencies F if the table R is always the natural join of the tables $R_1, R_2, \ldots, R_n$. It should be noted that natural join is the only way to recover the table from the decomposed tables. There is no other set of operators that can recover the table if the join cannot. Furthermore, it should be noted when the decomposed tables $R_1, R_2, \ldots R_n$ are obtained by projecting on the table R, for example, R_1 by projection $\Pi_1(R)$, the table R_1 may not always be precisely equal to the projection since the table R_1 might have additional rows called the *dangling* tuples.

It is not difficult to test whether a given decomposition is lossless-join given a set of functional dependencies F. We consider the simple case of a table R being decomposed into R_1 and R_2.

Definition—Lossless Decomposition

Let R be a relation with functional dependencies F. The decomposition of R into two relations with attribute sets R_1 and R_2 is lossless if and only if F^+ contains one of the following two FDs.

$R_1 \cap R_2 \rightarrow R_1$

$R_1 \cap R_2 \rightarrow R_2$

That is, the common attributes in R_1 and R_2 must include a candidate key of either R_1 or R_2.

We can now go back to the example in Tables 6.6 to 6.9. The attribute sets and the decomposed relations are given below.

enrol1 (*sno, cno, date-enrolled*)
enrol2 (*date-enrolled, room-no, instructor*)

The intersection of these two tables is (*date-enrolled*) since that is the only attribute that is common to both. Clearly it is not a key of either of the two relations *enrol1* or *enrol2*.

Loss decomposition may also be defined in terms the join of all decompositions. This definition requires that the natural join of all decomposed relations results in exactly the original relation with no spurious tuples.

6.5.3 Dependency Preservation Decomposition

It is clear that each decomposition must be lossless so that we do not lose any information from the table that is decomposed and a natural join of all decomposed relations results in the original relation.

Definition—Dependency Preservation
Dependency preservation is another important requirement since a dependency is a constraint on the database and if X → Y holds then we know that the two (sets) attributes are closely related and it is expected that both X and Y would appear in the same table so that the dependency can be checked easily.

Let us consider a table $R(A, B, C, D)$ with dependencies F that include the following:

$A \rightarrow B$
$A \rightarrow C$

If we decompose the above table into *R1(A, B)* and *R2(B, C, D)* the dependency $A \rightarrow C$ cannot be checked (or preserved) by looking at only one table since A and C are now in different tables. It is desirable that decompositions be such that each dependency in F may be checked by looking at only one table and that no joins need be computed for checking dependencies. In some cases, it may not be possible to preserve each and every dependency in F but as long as the dependencies that are preserved are equivalent to F, it should be sufficient.

Let F be the dependencies on a table R which is decomposed into tables $R_1, R_2, \ldots R_n$.

We can partition the dependencies given by F, such that $F_1, F_2, \text{K}\, F_n . F_n$ are dependencies that only involve attributes from each of the tables $R_1, R_2, \ldots R_n$ respectively. If the union of dependencies F_i imply all the dependencies in F, then we say that the decomposition has preserved dependencies, otherwise not. This checking would normally require first finding all the FDs from each table and then taking all the dependencies together and finding the closure of their union. If the closure is equivalent to the closure of FDs F, then we can say all dependencies have been preserved.

In the example of normalizing in Table 6.1, ODIplayers was decomposed into the following three relations:

Player2 (*PID, LName, Country, YBorn, Place*)
ODIMatch (*MID, Team1, Team2, Date, Ground*)
Performance (*PID, MID, NRuns, NWickets*)

We now use the list of FDs for the relation in Table 6.1 and check that all dependencies have been preserved.

Table 6.10 Checking dependency preservation for Table 6.1

FDs for Table 6.1	*Which Relation?*	*FD Preserved?*
PID → Name	*Player2*	*Yes*
PID → Country	*Player2*	*Yes*
MID → Team1	*ODIMatch*	*Yes*
MID → Team2	*ODIMatch*	*Yes*
PID → YBorn	*Player2*	*Yes*
PID → Place	*Player2*	*Yes*
MID → Date	*ODIMatch*	*Yes*
MID → Ground	*ODIMatch*	*Yes*
(PID, MID) → NRuns	*Performance*	*Yes*
(PID, MID) → NWickets	*Performance*	*Yes*

It was easy to check dependency preservation in this case since each dependency could be checked by looking at only one table.

6.5.4 Lack of Redundancy Decomposition

We have discussed the problem of repetition of information in a database. Such repetition should be avoided as much as possible.

Lossless-join, dependency preservation and lack of redundancy are not always possible with BCNF although 3NF can be used for lossless-join, dependency preservation and lack of redundancy.

6.6 MULTIVALUED DEPENDENCIES

Consider the following table *employee* that represents an entity employee that has one mutlivalued attribute *proj*ect.

employee (ID, department, salary, project)

We have so far considered normalization based on functional dependencies, dependencies that apply only to single-valued facts. For example, *ID → department* implies only one *department* value for each value of *ID*. Not all information in a database is single-valued, for example, *project* in an employee database may be the list of all projects that the employee is currently working on. Although *ID* determines the list of all projects that an employee is working on, *ID → project* is not a functional dependency since *project* is not single-valued.

Recall that when we discussed database modeling using the E-R modeling technique, we noted difficulties that can arise when an entity has multivalue attributes. It was because in the relational model all attribute values must be atomic and therefore, if all the information about such an entity is to be represented in one table, it is necessary to repeat all the information other than the multivalue attribute value. This results in

many rows about the same instance of the entity in the table and the table having a composite key (the entity ID and the mutlivalued attribute). The other option suggested was to represent this multivalue information in a separate table as if it was a separate entity.

The situation becomes much worse if an entity has more than one multivalued attribute and these values are represented in one table by a number of rows for each entity instance, such that every value of one the multivalued attributes appears with every value of the second multivalued attribute to maintain consistency. The multivalued dependency relates to this problem when more than one multivalued attribute exists in a table.

So far we have dealt with multivalued facts about an entity by heeping a separate table for that multivalue attribute and then inserting a row for each value of that fact. This resulted in composite keys since the multivalued fact must form a part of the key. In none of our examples so far have we dealt with an entity having more than one multivalued attribute in one table. This condition is discussed below.

The fourth and fifth normal forms deal with multivalued dependencies. Before discussing these normal forms we discuss the following example to illustrate the concept of multivalued dependency. We are now using an example which has little to do with ODI cricket. We have decided to find something different, another preoccupation in India, namely Bollywood!

We present a relation *BStars*, presented in Table 6.11, that gives information about some Bollywood stars, the cities they have lived in and names of their children. The information is not complete but the number of children and their names are correct.

Table 6.11 The relation *BStars*

Names	*Cities*	*Children*
Amitabh Bachchan	Allahabad	Abhishek
Amitabh Bachchan	Mumbai	Shweta
Madhuri Dixit	Mumbai	Arin
Madhuri Dixit	Los Angeles	Ryan
Shah Rukh Khan	Delhi	Aryan
Shah Rukh Khan	Mumbai	Suzana

The above table includes two multivalued attributes of entity *BStars*: *Cities* and *Children.* There are no functional dependencies[2].

The attributes *Cities* and *Children* are assumed to be independent of each other but Table 6.11 appears to show that there is some relation between the *Cities* and *Children* attributes. How can this be resolved? If we were to consider *Cities* and *Children* as separate entities, we would have two relationships (one between *Names* and *Cities* and the other between *Names* and *Children*). Both the above relationships are one-to-many.

2. Can we be sure about this? Could *Children* → *Name* be a functional dependency? It can only be so if we assume that the children's names are unique. We are assuming children's names are not unique in a large table.

That is, one Bollywood star is assumed to have lived in several *Cities,* though we have included only two cities for each person, and may have several children, though every film star we have used has only two children. Also many Bollywood stars may have lived or be living in the same city, for example, Mumbai. This is not an issue that concerns us.

Table 6.11 above is therefore in 3NF, it is even in BCNF, but it still has some disadvantages. We cannot decompose it based on FDs since there are no FDs in this table. So how should this information be represented? There are several possibilities. One is given in Table 6.11. Another possibility is given in Table 6.12 in which every value of *Cities* is combined with every value of *Children*. That at least overcomes the problem that the two attributes *Cities* and *Children* appeared related in Table 6.11.

Table 6.12 Another representation of the relation *BStars*

Names	*Cities*	*Children*
Amitabh Bachchan	Allahabad	Abhishek
Amitabh Bachchan	Mumbai	Shweta
Amitabh Bachchan	Allahabad	Shweta
Amitabh Bachchan	Mumbai	Abhishek
Madhuri Dixit	Mumbai	Arin
Madhuri Dixit	Los Angeles	Ryan
Madhuri Dixit	Mumbai	Ryan
Madhuri Dixit	Los Angeles	Arin
Shah Rukh Khan	Delhi	Aryan
Shah Rukh Khan	Mumbai	Suzana
Shah Rukh Khan	Delhi	Suzana
Shah Rukh Khan	Mumbai	Aryan

The two representations that have been presented in Tables 6.11 and 6.12 have some disadvantages. If the information is repeated we face the same problems of anomalies that we did when second or third normal form conditions were violated. If there is no repetition, there are still some difficulties with search, insertions and deletions. For example, if a Bollywood star had several values of *Cities*, more than the values of *Children*, then the value of *Children* in Table 6.11 for some values may have to be NULL. The role of NULL values then is confusing. Also the candidate key in the above tables is (*Names*, *Cities*, *Children*) and existential integrity requires that no NULL be allowed in the key. In a representation like in Table 6.12, every value of *Cities* (there may be five or more) will need to be repeated for every value of *Children*. This leads to a lot of redundancy. These problems may be overcome by decomposing a table like the one above in the following manner.

The basis of the decomposition is the concept of multivalued dependency (MVD). Functional dependency $A \rightarrow B$ relates one value of A to only one value of B while multivalued dependency $A \text{—>>} B$ defines a relationship in which a set of values of attribute B is determined by a single value of A.

Let us now define the concept of multivalued dependency.

Definition—Multivalued Dependency (MVD)
The multivalued dependency X —>> Y is said to hold for a relation R(X, Y, Z) if for a given set of values (set of values if X is more than one attribute) for attribute(s) X, there is a set of (zero or more) associated values for the set of attributes Y and the Y values depend only on X values and have no dependence on the set of attributes Z.

We can define MVD in another, somewhat more formal, way as well.

Another Definition—Multivalued Dependency (MVD)
The multivalued dependency X —>> Y is said to hold for R(X, Y, Z) if, whenever t_1 and t_2 are two rows in R that have the same values for attributes X and therefore $t_1[x] = t_2[x]$ then R also contains rows t_3 and t_4 (not necessarily distinct), such that

$t_1[x] = t_2[x] = t_3[x] = t_4[x]$
$t_3[Y] = t_1[Y]$ and $t_3[Z] = t_2[Z]$
$t_4[Y] = t_2[Y]$ and $t_4[Z] = t_1[Z]$

In other words, if t_1 and t_2 are given by

$t_1 = [X, Y_1, Z_1]$, and
$t_2 = [X, Y_2, Z_2]$

then there must be rows t_3 and t_4, such that

$t_3 = [X, Y_1, Z_2]$, and
$t_4 = [X, Y_2, Z_1]$

We are therefore requiring that every value of *Y* appears with every value of *Z* to keep the table instances consistent. In other words, if *Y* and *Z* are determined by *X* alone and there is no relationship between *Y* and *Z* then *Y* and *Z* must appear in every possible pair, as shown in Table 6.12. These pairings present no information and are of no significance except that they show there is no relationship between *Y* and *Z*. If some of these pairings were not present, there would be some significance in the pairings because this is likely to mean that the two attributes are not independent.

The theory of multivalued dependencies is very similar to that for functional dependencies. Given *D* is a set of MVDs, we may find D^+, the closure of *D* using a set of axioms. We do not discuss the axioms here.

In the example above, if there was some dependence between the attributes *Cities* and *Children*, for example, perhaps the *Children* were born in the *Cities* or they studied there or live there; the table would not have MVD and cannot be decomposed into two tables as is described below.

In the above Table 6.12 whenever *X* —>> *Y* holds, so does *X* —>> *Z* as the role of the attributes *Y* and *Z* is symmetrical.

It should be noted that if *Z* was a single-valued attribute then there is no MVD and we deal with *R(X, Y, Z)* as before by entering several rows about each entity because *R* does not have MVD.

6.6.1 Multivalued Normalization—Fourth Normal Form

We have considered the example of the table *BStars*(*Name, Cities, Children*) and discussed the problems that may arise if the table is not normalized further. The table could be decomposed into *BS1*(*Names*, *Cities)* and *BS2*(*Names, Children*) to overcome these problems. The decomposed tables are in fourth normal form (4NF) which we now define after considering the concept of trivial MVDs.

The concept of multivalued dependencies (MVD) was developed to provide a basis for decomposition of tables like the one in Table 6.12. Therefore, if a table like *enrolment(sID, subjectID)* has a relationship between *sID* and *subjectID* in which *sID* uniquely determines the values of *subjectID*, the dependence of *subjectID* on *sID* is called a *trivial* MVD since the table *enrolment* with only two attributes cannot be decomposed any further. More formally, MVD X —>> Y is called a trivial MVD if either Y is a subset of X or X and Y together form the table R. The MVD is trivial since it results in no constraints being placed on the table.

A table having nontrivial MVDs must have at least three attributes, and two of them must be multivalued. Nontrivial MVDs result in the table requiring some constraints, otherwise all possible combinations of the multivalue attributes are required to be in the table.

4NF can be defined as follows:

Definition—Fourth Normal Form 4NF
A table R is in 4NF if, whenever a multivalued dependency X —>> Y holds in the table R, then either
(a) the dependency is trivial, or
(b) X is a candidate key for R.

As noted earlier, the dependency X —>> ϕ? or X —>> Y in a table $R(X, Y)$ is trivial since they must hold for all $R(X, Y)$. Similarly, $(X, Y) \rightarrow Z$ must hold for all tables $R(X, Y, Z)$ with only three attributes.

In fourth normal form, we have a table that has information about only one entity. If a table has more than one multivalue attribute, we decompose it to remove difficulties with the multivalued facts. Table 6.12 is therefore not in 4NF and needs to be decomposed in the following two tables that we have discussed before.

BS1 (*Names*, *Cities*)
BS2 (*Names, Children*)

The tables *BS1* and *BS2* are given below in Tables 6.13 and 6.14.

Table 6.13 A 4NF relation *BS1*

Names	*Cities*
Amitabh Bachchan	Allahabad
Amitabh Bachchan	Mumbai
Madhuri Dixit	Mumbai
Madhuri Dixit	Los Angeles
Shah Rukh Khan	Delhi
Shah Rukh Khan	Mumbai

Table 6.14 A 4NF relation *BS2*

Names	*Children*
Amitabh Bachchan	Abhishek
Amitabh Bachchan	Shweta
Madhuri Dixit	Arin
Madhuri Dixit	Ryan
Shah Rukh Khan	Aryan
Shah Rukh Khan	Suzana

These tables are clearly in 4NF since any table with only two attributes has no MVD.

Intuitively *R* is in 4NF if all dependencies are a result of keys. When multivalued dependencies exist, a table should not contain two or more independent multivalued attributes because the multivalued attributes must then be part of the key. The decomposition of a table to achieve 4NF would normally result in not only a reduction of redundancies but also avoidance of anomalies.

We leave it as an exercise for readers to find the join of relations in Table 6.13 and 6.14 and check if the join matches the original table.

6.6.2 Join Dependencies and the Fifth Normal Form

The normal forms discussed so far required that the given table *R* if not in the given normal form be decomposed into two tables to meet the requirements of the normal form. In some rare cases, a table can have problems like redundant information and update anomalies but cannot be decomposed into two tables to remove the anomalies. In such cases it may be possible to decompose the table into three or more tables using the 5NF.

The fifth normal form deals with join-dependencies which is a generalization of the MVD. The aim of fifth normal form is to have tables that cannot be decomposed further. A table in 5NF cannot be constructed from several smaller tables.

Definition—Join Dependency
*Let a relation R have subsets of its attributes A, B, C, …Then R satisfies the Join Dependency (JD) written as *(A, B, C, …) if and only if every possible legal value of R is equal to the join of its projections A, B, C, …*

Before we define 5NF, we present an example in Table 6.15 below that deals with names of some movies, and names of their heroes and heroines. We have chosen a small sample of movies from Bollywood.

Table 6.15 *Movies*, *Heroes* and *Heroines*

Movies	*Heroes*	*Heroines*
Lagaan	Aamir Khan	Gracy Singh
Ghajini	Aamir Khan	Asin Thotumkal
God Tussi Great Ho	Salman Khan	Priyanka Chopra
God Tussi Great Ho	Amitabh Bachchan	Manisha Koirala
Hey Ram	Kamal Hasan	Rani Mukerjee
Paheli	Shah Rukh Khan	Rani Mukerjee

Note that Table 6.15 has information on a number of movies; each movie has a hero and a heroine except in one case two heroes and two heroines are given. All three fields are needed to represent the information.

Table 6.15 does not show MVDs as the attributes *Movies, Heroes* and *Heroines* as independent. They are related to each other and the pairings have significant information in them. If we try to decompose the relation into the following two:

(Movies, Heroes), and
(Movies, Heroines)

the relations given in Tables 6.16 and 6.17 are obtained by carrying out the decomposition.

Table 6.16 The decomposed relation (*Movies*, *Heroes*)

Movies	*Heroes*
Lagaan	Aamir Khan
Ghajini	Aamir Khan
God Tussi Great Ho	Salman Khan
God Tussi Great Ho	Amitabh Bachchan
Hey Ram	Kamal Hasan
Paheli	Shah Rukh Khan

Table 6.17 The decomposed relation (*Movies*, *Heroines*)

Movies	*Heroines*
Lagaan	Gracy Singh
Ghajini	Asin Thotumkal
God Tussi Great Ho	Priyanka Chopra
God Tussi Great Ho	Manisha Koirala
Hey Ram	Rani Mukerjee
Paheli	Rani Mukerjee

If the two relations are now joined (on the attribute *Movies*), we obtain the relation in Table 6.18.

Table 6.18 The join of relations 6.16 and 6.17

Movies	*Heroes*	*Heroines*
Lagaan	Aamir Khan	Gracy Singh
Ghajini	Aamir Khan	Asin Thotumkal
God Tussi Great Ho	Salman Khan	Priyanka Chopra
God Tussi Great Ho	Salman Khan	Manisha Koirala
God Tussi Great Ho	Amitabh Bachchan	Manisha Koirala
God Tussi Great Ho	Amitabh Bachchan	Priyanka Chopra
Hey Ram	Kamal Hasan	Rani Mukerjee
Paheli	Shah Rukh Khan	Rani Mukerjee

The result of the join in Table 6.18 has two additional rows that were not in the original relation, Table 6.15. Therefore the decomposition is not lossless although it has more rows than the original relation. Other decompositions are possible but they also lead to exactly the same problem.

Table 6.15 can however be decomposed into the following three relations:

(*Movies, Heroes*),
(*Movies, Heroines*),
(*Heroes, Heroines*).

The first two relations are as given in Tables 6.16 and 6.17 and the third is given in Table 6.19.

Table 6.19 The decomposed relation (*Heroes*, *Heroines*)

Heroes	*Heroines*
Aamir Khan	Gracy Singh
Aamir Khan	Asin Thotumkal
Salman Khan	Priyanka Chopra
Amitabh Bachchan	Manisha Koirala
Kamal Hasan	Rani Mukerjee
Shah Rukh Khan	Rani Mukerjee

Now if the three relations, given in Tables 6.16, 6.17 and 6.19 are joined, a different result is obtained. Table 6.18 gives the result of a join of (*Movies, Heroes*) and (*Movies, Heroines*) so we can join this result with the relation given in Table 6.19 (on both attributes in Table 6.19). Now we find that the two additional rows that appeared in Table 6.18 disappear and we obtain exactly the relation in Table 6.15.

Therefore relations presented in Tables 6.16, 6.17 and 6.19 show the join dependency of the relation in Table 6.15.

We are now able to define the fifth normal form.

Definition—Fifth Normal Form (5NF)
A relation R is in 5NF (also called the project-join normal form or PJNF) if for all join dependencies at least one of the following conditions holds:

(a) The decomposed tables (A, B, C, …) are a trivial join-dependency, that is one of the A, B, C, … is itself the relation R)
(b) Every A, B, C, … is a candidate key for R.

It can be shown that the second condition of the definition above is met if the decomposition of *R* into (*A, B, C,* …) leads to a lossless join.

It was shown earlier that the decomposition of the relation in Table 6.15 into Tables 6.16, 6.17 and 6.19 is lossless. Therefore these three relations are in 5NF. They cannot be decomposed any further.

It should be noted that there is a significant difference between the nature of functional dependencies and multivalue and join dependencies. The functional dependencies deal more with the semantics in the relational table which results in deriving decompositions. The mutlivalue and join dependencies give little semantics information about the tables. It has more to do with the way information has been structured than with the semantics.

As noted earlier, 4NF and 5NF are often not necessary and normalization can be terminated after BCNF has been achieved.

6.7 DENORMALIZATION

So far we have considered the process of normalization which involves decomposing relations so that relations were at least in 3NF. Further normalization to BCNF, 4NF and 5NF has also been also been discussed and recommended. The primary aim of normalization is to overcome anomalies that can exist in relations that are not normalized. Although normalization aims to overcome anomalies, it does not consider database performance. Normalization involving decomposing relations can have a performance penalty since many queries require additional joins and join is an expensive operation. Query performance can often be improved by joining relations together if the relations are being joined frequently and the join is expensive to compute. For example, if every query that is using the relation *Batting* is requiring a join with the relation *Player* then it may be appropriate to either join the two tables and put the player names in the *Batting* table as well as in *Player*. This technique for improving performance is called *denormalization*. This is defined as follows:

Definition—Denormalization
Denormalization is the intentional duplication of information in the database tables leading to increased data redundancy but improving query processing performance by reducing the number of joins.

As noted above, normalizing tables is generally the recommended approach in database design but some applications require information from both tables that were on the table before normalization. The premise of normalization is that the information can be retrieved by joining the two tables. The problem is that normalization may lead to too many joins for some applications. As the number and sizes of tables increase, the access costs also increase because the join of large tables takes too much time. When this happens, denormalization is worth considering.

Denormalization should only be considered after a good understanding of the data and the access requirements of all frequently used applications is known. These may involve expensive decision support applications including report generation. It may be necessary to consult the application developers before deciding to denormalize tables. Data updates should also be considered because redundancy may lead to increased cost of updates. It is best to remember that even if denormalization solves one part of dealing with performance, it may create possible problems in some other areas like updates. Total performance needs to be evaluated before denormalization. Furthermore, data integrity risks should also be considered.

Denormalization may involve inclusion of some additional columns in a table or it may involve combining two entities and the relationship between them into one table. For example, if two entities have a one-to-one relationship then they could be denormalized into one table to reduce the number of joins if the queries are frequently accessing both tables together. Also, two entities that have a many-to-one relationship may also be denormalized into one table to reduce the number of joins if many queries are accessing both tables together. Denormalizing one-to-one and one-to-many relationships between two entities does not create too much redundancy and may be useful in improving query performance. Two entities that have a many-to-many relationship should perhaps not be denormalized because of significant redundancy that is likely to be introduced.

As an example consider the three tables in Fig. 6.4 that are in 2NF.

Player2(*PID, Lname, YBorn, Place, Country*)
ODI(*MID, Team1, Team2, Date, Ground*)
Performance(PID, MID, NRuns, NWickets)

Figure 6.4 A database schema in 2NF

If most common queries to the database are about bowling and batting performance using the player's last name then it might make sense to put the name in the *Performance* table and obtain the schema given in Fig. 6.5.

Player2(*PID, Lname, YBorn, Place, Country*)
ODI(*MID, Team1, Team2, Date, Ground*)
Performance(PID, Lname, MID, NRuns, NWickets)

Figure 6.5 Denormalized database schema for schema in Fig. 6.4.

This clearly increases the level of redundancy because a player like Tendulkar or Jayasuriya has played more than 400 ODIs and so their names will appear more than 400 times but it will improve the performance of some queries.

We now summarise our discussion in Fig. 6.6.

Item	Normalize	Denormalize
Retrieval Performance	High	Higher for some queries
Update Performance	High	Slower
Storage Performance	Efficient	Not so efficient
Level of Normalization	3NF or higher	3NF or lower
Typical Applications	OLTP	OLAP
Preserving Integrity	Yes	Not always

Figure 6.6 Comparing normalization and denormalization

In most cases, denormalization does not improve query performance significantly but there are always exceptions. Generally a normalized database delivers good performance and preserves data integrity which may be compromised when the database is denormalized.

6.8 INCLUSION DEPENDENCY

The concept of inclusion dependency was first presented in 1984 by Casanova, Fagin and Papadimitriou. Inclusion dependency (IND) is somewhat different than functional dependency and join dependency as inclusion dependency may deal with attributes in one table or from more than one able. Inclusion dependency may involve comparison of column values from the same table or from different tables. Inclusion dependency deals with the semantics of a database schema. INDs are basic constraints that are necessary in a database to represent relationships. While the focus of functional dependencies was the primary key, the focus of inclusion dependency is the concept of foreign keys (and referential integrity) in a database schema.

Definition—Inclusion Dependency
An inclusion dependency on a database is a statement where X and Y are compatible sequence of attributes in relations R and S respectively, then the dependent attribute X is a subset of referenced attribute Y.

A simple example of inclusion dependency is in the tables *Player, Match, Batting and Bowling. PID* in tables *Batting* and *Bowling* are a subset of *PlayerID* in *Player* while *MatchID* in *Batting* and *Bowling* are subsets of *MatchID* in *Match*.

A unary inclusion dependency is an inclusion dependency in which exactly one column appears on each side. Therefore, *X* and *Y* are single columns. Unary INDs are common when a primary key is a single attribute like in the examples presented above. A foreign key constraint is an example of inclusion dependency.

SUMMARY

In this chapter we have discussed the problems that can occur in a database system due to poor database design. We have described the following:

- Database modeling may be top-down (design by analysis) or bottom-up (design by synthesis).
- An example has been used to illustrate problems that can occur in a database system due to poor database design.
- Problems that may arise include redundancies, update anomalies, deletion anomalies, and insertion anomalies.
- The problems may be overcome by normalization of relations that contain information about more than one thing.
- The concepts of functional dependencies, closure and minimal cover are required in carrying out normalization.
- Functional dependencies (FD) show which attributes are dependent on which other attributes and FDs form the basis of normalization.
- Normalization has a number of steps. The first four steps of normalization are based on single valued dependencies. These are called First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF0 and Boyce–Codd Normal Form (BCNF). Definitions of these normal forms are presented.
- Normalizing a relation involves decomposing the relation.
- Relation decomposition must meet a number of desirable properties of relation decompositions. These desirable properties are attribute preservation, lossless-join decomposition, dependency preservation and lack of redundancy.
- The concept of multivalued dependencies was described and Fourth Normal Form (4NF) was discussed.
- The concept of join dependency and motivation for Fifth Normal Form (5NF) was described.
- Denormalization is the internal duplication of information in database relations in an attempt to improve system performance.
- Concept of inclusion dependency has been introduced.

REVIEW QUESTIONS

1. What are top-down and bottom-up approaches to database design? (Section 6.1)
2. Explain the reasons for the update, insertion and deletion anomalies? (Section 6.2)
3. Why is there a need for normalization of a relation that contains information about more than one thing? (Section 6.2)
4. Describe the problems that may arise if the relations are not normalized beyond 1NF, i.e., redundancies, update anomalies, deletion anomalies, and insertion anomalies. (Section 6.2)
5. Explain the concept of functional dependency. (Section 6.3)
6. What is closure and minimal cover? (Section 6.3)

7. What are Armstrong's axioms? (Section 6.3)
8. Explain how minimal cover may be derived. (Section 6.3)
9. Explain the role of functional dependencies in normalization. (Section 6.3)
10. Explain the basis of the first normal form, the second normal form and the third normal form. (Sections 6.4.1, 6.4.2, 6.4.3)
11. Explain what the Boyce–Codd normal form achieves and the inadequacy of BCNF in some situations. (Section 6.4.3)
12. What are the differences between 1NF, 2NF, 3NF and BCNF? (Section 6.4)
13. What is decomposition? (Section 6.5)
14. What is attribute preservation? (Section 6.5.1)
15. What is lossless-join decomposition? (Section 6.5.2)
16. What is dependency preservation? (Section 6.5.3)
17. What is lack of redundancy in decomposition? (Section 6.5.4)
18. Show how to normalize a simple familiar example. (Section 6.4)
19. Describe multivalued dependency by presenting an example. (Section 6.6)
20. Use the example to illustrate the fourth normal form. (Section 6.6.1)
21. Describe join dependency. (Section 6.6.2)
22. Explain fifth normal form and its motivation. (Section 6.6.2)
23. When should denormalization be used? What are its disadvantages? (Section 6.7)
24. What is inclusion dependency? (Section 6.8)

SHORT ANSWER QUESTIONS

1. What is update anomaly?
2. What is insertion anomaly?
3. What is deletion anomaly?
4. Give an example to illustrate the need for normalization of a relation?
5. Why do we need to normalize relations beyond 1NF?
6. Use the example above and identify three functional dependencies.
7. What is the purpose of the three Armstrong's axioms?
8. List the three properties of irreducible FDs.
9. Define the first normal form.
10. Define the second normal form.
11. Define the third normal form.
12. List two desirable properties of decompositions.
13. What is multivalued dependency?
14. Define the fourth normal form?
15. Give an example of use of the fifth normal form.
16. What is the major aim of denormalization?

17. Explain inclusion dependency using a simple example.
18. How does inclusion dependency differ from functional dependency?

MULTIPLE CHOICE QUESTIONS

1. Which two of the following are the primary aims of normalization?
   (a) Avoid data redundancies.
   (b) Avoid certain update, insertion and deletion anomalies.
   (c) Decompose large relations into smaller ones.
   (d) Determine functional and multivalue dependencies.
2. Which one of the following is correct?
   (a) All functional dependencies are many-to-many relationships.
   (b) All functional dependencies are many-to-one relationships.
   (c) All functional dependencies are one-to-one relationships.
   (d) None of the above.
3. Which one of the following is correct?
   (a) All functional dependencies must involve attributes of the same table.
   (b) All functional dependencies must involve only a single attribute on the left side.
   (c) All functional dependencies must involve only a single attribute on the right side.
   (d) All functional dependencies must involve only single attributes on both sides.
4. Which of the following is **not** correct if the table *R* is a binary table?
   (a) There can be no functional dependencies in *R*.
   (b) There must be two functional dependencies in *R*.
   (c) There can be no multivalue dependencies in *R*.
   (d) *R* cannot be in BCNF.
5. Some facts are single-valued facts while others are multivalue facts. For example, we will consider a person's name to be single-valued although it would normally consist of a first name, a last name and possibly one or more middle names. If a person had more than one different name then of course the name becomes a multivalue fact.
   Given the above, which two of the following are single-valued facts?
   (a) Place of birth of a person
   (b) Date of birth of a person
   (c) Courses taught by an instructor
   (d) Qualifications of an instructor
6. Once again we consider single-valued and multivalued facts. A person's address would normally be single-valued but may also be multivalued for some persons. For example, for a student who has a permanent address as well as an address during the academic year. Which one of the following is always a single-valued fact?
   (a) A player's telephone number
   (b) Number of ODI centuries scored
   (c) Countries a player has played against
   (d) None of the above

7. Which of the following is correct?
   (a) Functional dependencies are not associated with tables; they are based on the semantics of information that we are dealing with.
   (b) If a table has no redundant information, its attributes must not have any functional dependencies.
   (c) Functional dependencies may be determined if we are given several instances of a table.
   (d) None of the above
8. Consider a relationship Z between two entities X and Y. Let C be the primary key of X and D be the primary key of Y. Which one of the following is **not** correct
   (a) If the relationship is one-to-one, we must have $C \rightarrow D$ and $D \rightarrow C$.
   (b) If the relationship is many-to-one, we would have $C \rightarrow D$ but not $D \rightarrow C$.
   (c) For many-to-many relationships, no functional dependencies between C and D hold.
   (d) Let C be *student number* and D be *subject number*, and suppose there was an attribute *mark* in the table representing the relationship *enrolment* then no functional dependencies exist between C, D and *mark*.
   (e) None of the above
9. Which of the following is one of the three Armstrong's axioms?
   (a) Transitivity Rule: If $X \rightarrow Y$, $Y \rightarrow Z$ then $X \rightarrow Z$
   (b) Commutativity Rule: If $X \rightarrow Y$ then $Y \rightarrow X$
   (c) Union Rule: If $X \rightarrow Y$, $X \rightarrow Z$ then $X \rightarrow YZ$
   (d) Pseudotransitivity Rule: If $X \rightarrow Y$, $WY \rightarrow Z$ then $WX \rightarrow Z$
10. Let R be a relation with attributes (A,B,C,D,E,F) and let the following functional dependencies hold:

    $A \rightarrow B$
    $A \rightarrow C$
    $CD \rightarrow E$
    $CD \rightarrow F$
    $B \rightarrow E$

    Given the above functional dependencies, which of the following functional dependencies does **not** hold?
    (a) $A \rightarrow E$ (b) $CD \rightarrow EF$ (c) $AD \rightarrow F$ (d) $B \rightarrow CD$
11. Let a table R have three candidate keys A, B and (C, D). Which of the following *must* **not** be correct?
    (a) $A \rightarrow B$ (b) $B \rightarrow A$ (c) $A \rightarrow C$ (d) $C \rightarrow AB$
12. Let the functional dependencies of a table R be given by:

    sno → *sname*
    cno → *cname*
    sno → *address*
    cno → *instructor*
    instructor → *office*

    Given the above functional dependencies, how many functional dependencies are likely to be in the closure?
    (a) 5 (b) More than 5 but less than 15
    (c) More than 15 but less than 50 (d) Over 100

13. Which of the following is **not** correct?
    (a) Given a set of functional dependencies F, F^+ is the closure of F and includes all functional dependencies implied by F.
    (b) F^+ is useful since it enables us to find if some FD $X \rightarrow Y$ is implied by F.
    (c) Given a large set of functional dependencies F, it can be very time-consuming to compute F^+.
    (d) Given F and a set of attributes X, the closure of X (X^+) is the set of all attributes that are functionally dependent on X given F.
    (e) None of the above
14. Consider the following functional dependencies:

    $A \rightarrow B$
    $B \rightarrow A$
    $B \rightarrow C$
    $A \rightarrow C$
    $C \rightarrow A$

    Which of the above dependencies will be removed if we were to compute the minimal cover of the functional dependencies?
    (a) $B \rightarrow A$ (b) $A \rightarrow C$
    (c) $B \rightarrow C$ (d) Either $B \rightarrow A$ and $A \rightarrow C$ or $B \rightarrow C$ but not all three
15. Which of the following are true? A set of functional dependencies is irreducible if and only if
    (a) The left-hand side of every FD is just one attribute.
    (b) The right-hand side of every FD is just one attribute.
    (c) Both the left and right-hand side of every FD is just one attribute.
    (d) The left-hand side is irreducible.
16. A relation R is decomposed in two relations $R1$ and $R2$ during normalization. Which of the following must be correct if the decomposition is nonloss?
    (a) Every FD in R is a logical consequence of those in $R1$ and $R2$.
    (b) The common attributes of $R1$ and $R2$ form a candidate key for at least one of the pair.
    (c) $R1$ and $R2$ together have all the attributes that are in R.
    (d) $R1$ and $R2$ have no attributes that are common.
17. Which of the following properties the decomposition of a relation R into a number of relations must have?
    (a) It must be a nonloss decomposition.
    (b) It must be a dependency preserving decomposition.
    (c) The original relation R can always be recovered from the decomposed relations.
    (d) All of the above
18. A relation is required to be decomposed during normalization into tables that are lossless and dependency preserving. Which one of the following is true?
    (a) The highest normal form possible is 1NF.
    (b) The highest normal form possible is 2NF.
    (c) The highest normal form possible is 3NF.
    (d) The highest normal form possible is BCNF.

19. When R is decomposed into S and T, which of the following is true?
    (a) Joining S and T will always produce a table that includes all the rows that were in R but may include some others.
    (b) The decomposition is lossless if the join of S and T is precisely R.
    (c) $R(A, B, C)$ and $A \rightarrow B$ then (A, B) and (A, C) are lossless.
    (d) S and T are independent, that is, there are no inter-relation dependencies.
    (e) The decomposition is lossless even if the join of S and T does not recover R because in some cases other operators on S and T can recover R.
20. Consider the table *student*(sno, sname, cname, cno) where (sno, cno) and (sname, cname) are the candidate keys of the relation. Which one of the following is **not** true?
    (a) The above table is in 1NF.
    (b) The above table is in 2NF.
    (c) The above table is in 3NF.
    (d) The above table is in BCNF.
21. Which one of the following is **not** correct?
    (a) If a table is a relation, it is in 1NF.
    (b) If a table is in 3NF and does not have a composite key, it is in BCNF.
    (c) If an E-R model is mapped to a set of tables, the tables thus obtained are *normally* in 3NF.
    (d) If a table has no multivalued dependencies, the table is in 4NF.
22. Which of the following is **not** correct
    (a) 2NF and 3NF are based on functional dependencies.
    (b) BCNF and 4NF are based on multivalue dependencies.
    (c) 5NF is based on join dependencies.
    (d) 1NF is based on the structure of the data.
23. Which one of the following is a difference between a relation in BCNF or in 3NF?
    (a) A relation in BCNF has two or more candidate keys.
    (b) A relation in BCNF has composite candidate keys.
    (c) A relation in BCNF has overlapping candidate keys.
    (d) A relation in BCNF has all of the above.
24. Which of the following are correct?
    (a) All normal forms are additive.
    (b) 1NF satisfies the structural properties of a table.
    (c) 2NF and 3NF ensure that each table contain information about only one thing.
    (d) Decompositions must ensure that the original table may be reconstructed from the decomposed tables.
    (e) All of the above are correct.
25. Which of the following are correct?
    (a) Tables in a relational database are required to be normalized to the highest normal form.
    (b) 2NF and 3NF are designed to result in tables in which each nonkey attribute in each table provides a fact about the entity or relationship being identified by the key.
    (c) While normalizing tables, it is essential to decompose tables such that the original table can be constructed from the decomposed tables.

(d) Functional dependencies cannot be automatically derived from an instance of the table under consideration. Careful study of the real world is needed to derive the functional dependencies.

26. Consider the following table that provides details about shipments. The attributes are supplier ID, supplier name, part ID and the quantity supplied. The first three of the four attributes are unique. Also, there may be a number of shipments from the one supplier but they must involve different parts. Two shipments from the same supplier supplying the same part are not permitted.

SP(sid, sname, pid, qty)

Which of the following is true about the relation above?

(a) SP is in 2NF but not 3NF. (b) SP is in 3NF but not BCNF.
(c) SP is in BCNF but not 4NF. (d) SP is in 4NF.

27. If a table *R* consists only of its primary key (which may consist of a number of attributes) and has no other attributes, the table would always be in

(a) 2NF but may not be in 3NF. (b) 3NF but may not be in BCNF.
(c) BCNF but may not be in 4NF. (d) 4NF.

28. Which of the following is **not** correct for a table $R(X, Y, Z)$ where X, Y, and Z are not necessarily single attributes.

(a) X —>> Y then X —>> Z (b)$X \rightarrow Y$ implies X —>> Y
(c) X —>> Y and X —>> Z imply $Y \rightarrow Z$ (d) X —>> Y and Y —>> Z imply X —>> $Z - Y$

29. MVD requires that there be at least three attributes. Which one of the following is correct if the table is $T(C, H, R)$ where C is the course number and H is the time it is held and R is the room in which it is held:

(a) The only MVD is C —>> H (b) The only MVD is C —>> R
(c) The only MVD is C —>> HR (d) There are no multivalue dependencies

30. Consider the relation *Enrolment* (sno, sname, cno, cname, date enrolled). Which one of the following is true about the table Enrolment?

(a) The table is in 3NF.
(b) The table is in 3NF only if sname and cname are unique.
(c) The table is not in 2NF if sname is not unique.
(d) The table is in BCNF if sname and cname are unique.

31. Which one of the following is **not** true?

(a) MVD arises because the relational model does not allow non-atomic attribute values.
(b) MVD attempts to solve the problem of more than one multivalue attribute in one table; problems caused by having to store all possible combinations of the multivalue attribute values.
(c) MVD arises when two independent relationships are mixed together in one table.
(d) MVD arises when the values of the multivalue attributes cannot be determined by the primary key value.

32. Which one of the following is **not** correct about decomposition of a table in two or more tables:

(a) Attribute preservation is required so we do not lose any information.
(b) Dependency preservation is required so we do not lose any information.
(c) Lossless join is required so we do not lose any information.
(d) Lack of redundancy is required so we do not lose any information.

EXERCISES

1. Is a table with only one column always in 3NF? What about a table with two columns and a table with three columns? Explain your answer and give examples to support it.
2. Is a table with only two columns that is in 3NF always in BCNF? What about a table with three columns that is in 3NF? Explain your answer and give examples to support it.
3. Explain the concept of functional dependency. Does every relation have some functional dependencies? If not, give an example of a relation that has no functional dependencies.
4. Discuss the properties of decomposition including attribute preservation, dependency preservation and lossless join. Explain why each of these properties is important? Give an example of table decomposition which is not lossless.
5. Explain clearly what is meant by multivalued dependency? What types of constraint does it specify? When does it arise? Give an example of a relation with multivalued dependency.
6. Define BCNF. How is it different from 3NF? Present an example of a relation in 3NF that is not in BCNF.
7. Give an example of a table that is in BCNF but not in 4NF. Explain briefly.
8. A table R has attributes A, B, C, D, E and satisfies the following functional dependencies:

   $A \rightarrow BC$
   $B \rightarrow D$
   $CD \rightarrow E$
   $E \rightarrow A$

   (a) What are the candidate keys?
   (b) Is this table in 2NF? 3NF? BCNF?
   (c) Show that (*A, B, C*) and (*A, D, E*) are a lossless decomposition of R.
   (d) Show that (*A, B, C*) and (*A, D, E*) are not a dependency preserving decomposition of R.
9. Normalize the relation given below to the highest normal form possible. Are there any functional dependencies in this relation? Are there any multivalued dependencies? Does the relation have join dependencies? Explain the normalization process.

Directors, *Movies* and *Actors*

Directors	*Movies*	*Actors*
Mani Ratnam	Aayutha Ezhuthu	Madhvan
Mani Ratnam	Aayutha Ezhuthu	Suriya
Mani Ratnam	Guru	Abhishek Bachchan
Ashutosh Gowarikar	Lagaan	Aamir Khan
Mani Ratnam	Bombay	Manisha Koirala
Ashutosh Gowarikar	Jodha Akbar	Hrithik Roshan
Ashutosh Gowarikar	Jodha Akbar	Aishwarya Rai
Shankar	Sivaji	Rajnikanth

Contd.

Contd.

Shankar	Sivaji	Shriya Saran
Shankar	Anniyan	Vikram
Shankar	Anniyan	Yana Gupta
Mani Ratnam	Guru	Aishwarya Rai

10. Could the relation in the last exercise be normalized into the following three relations? Clearly explain the basis of your answer. If decomposition into these three relations is not suitable then what decomposition is needed for normalization?

The first decomposed relation has *Directors* and *Movies* as columns:

The decomposed relation (*Directors*, *Movies*)

Directors	*Movies*
Mani Ratnam	Aayutha Ezhuthu
Mani Ratnam	Guru
Ashutosh Gowarikar	Lagaan
Mani Ratnam	Bombay
Ashutosh Gowarikar	Jodha Akbar
Shankar	Sivaji
Shankar	Anniyan

The second decomposed relation has *Directors* and *Actors*:

The decomposed relation (*Directors*, *Actors*)

Directors	*Actors*
Mani Ratnam	Madhvan
Mani Ratnam	Suriya
Mani Ratnam	Abhishek Bachchan
Ashutosh Gowarikar	Aamir Khan
Mani Ratnam	Manisha Koirala
Ashutosh Gowarikar	Hrithik Roshan
Ashutosh Gowarikar	Aishwarya Rai
Shankar	Rajnikanth
Shankar	Shriya Saran
Shankar	Vikram
Shankar	Yana Gupta
Mani Ratnam	Aishwarya Rai

The third decomposed relation has *Movies* and *Actors*:

The decomposed relation (*Movies*, *Actors*)

Movies	*Actors*
Aayutha Ezhuthu	Madhvan
Aayutha Ezhuthu	Suriya
Guru	Abhishek Bachchan
Lagaan	Aamir Khan
Bombay	Manisha Koirala
Jodha Akbar	Hrithik Roshna
Jodha Akbar	Aishwarya Rai
Sivaji	Rajnikanth
Sivaji	Shriya Saran
Anniyan	Vikram
Anniyan	Yana Gupta
Guru	Aishwarya Rai

PROJECT

1. The following schema gives information about students, instructors and courses. Some of the names are defined below.

   University (SID, Sname, gender, caddress, haddress, dob, nationality, ssresults, ccompleted, ccgrades, cenrolment, staffID, staffname, gender, saddress, sdob, nationality, qualifications, appointment, courseID, cname, department, syllabus, instructor, quota)

   - caddress = current address
   - haddress = home address
   - ssresults = student's school results
   - ccompleted = courses completed
   - ccgrades = grades in courses completed
   - cenrolment = current enrolment
   - saddress = staff address
   - sdob = staff date of birth

   Make suitable assumptions and list them. Now find all the FDs in this schema and normalize the relations to 1NF, 2NF, 3NF and BCNF.

LAB EXERCISES

1. Are there a set of FDs, MVDs and JDs that are always true for all relations? Describe them.
2. Prove or disprove the following:
   (a) If $AB \rightarrow C$, $A \rightarrow D$, $CD \rightarrow EF$ then $AB \rightarrow F$
   (b) If $XW \rightarrow Y$ and $XY \rightarrow Z$ then $X \rightarrow (Z - W)$
3. A table R has attributes P, Q, R, S, T, U, V, W and satisfies the following functional dependencies:
   $PQR \rightarrow S$
   $PQ \rightarrow U$
   $Q \rightarrow V$
   $R \rightarrow W$
   What are the candidate keys? Is this an irreducible set of functional dependencies?
4. What is inclusion dependency? Give an example to illustrate the concept.

BIBLIOGRAPHY

For Students

The books by Churcher, Teorey and Date are a good start for a student who wants to learn more about normalization. The article by Kent is a good introduction to normal forms. It is available on the Web.

For Instructors

The book by Simsion is excellent and the book by Hoberman is easy to read. A number of original papers on normalization have been included in the bibliography below. Chapters 6 and 7 of the book by Maier are a somewhat advanced discussion of normal forms. The whole book is available free online.

Beeri, C., and P. A. Bernstein, "Computational Problems Related to the Design of Normal Form Relational Schemas", *ACM Transactions of Database Systems*, Vol. 4, No. 1, September 1979, pp 30-59.

Bernstein, P. A., "Synthesizing Third Normal Form Relations from Functional Dependencies", *ACM Transactions of Database Systems*, Vol. 1, No. 4, October 1976, pp 277-298.

Churcher, C., *Beginning Database Design: From Novice to Professional*, APress, 2007, 300 pp.

Date, C. J., *Date on Database: Writings 2000-2006*, APress, 2006, 568 pp.

Fagin, R., "Multivalued Dependencies and a New Normal Form for Relational Databases", *ACM Transactions of Database Systems*, Vol. 2, No. 3, September 1977, pp 262-278.

Fagin, R., "A normal form for relational databases that is based on domains and keys", *ACM Transactions of Database Systems*, Vol. 6, No. 3, Sept. 1981, pp 387-415.

Hoberman, S., *Data Modeling Made Simple: A Practical Guide for Business & Information Technology Professionals*, Technics Publication, 2005, 144 pp.

Kent, W., "A simple guide to five normal forms in relational database theory", *Communications of the ACM*, Vol. 26, No. 2, February 1983, pp 120-125. http://www.bkent.net/Doc/simple5.htm

Maier, D., "Databases and Normal Forms", In *Theory of Relational Databases*, Chapter 6. http://www.dbis.informatik.hu-berlin.de/~freytag/Maier/C06.pdf

Maier, D., "Multivalued Dependencies, Join Dependencies and Further Normal Forms" *Theory of Relational Databases*, Chapter 7. http://www.dbis.informatik.hu-berlin.de/~freytag/Maier/C07.pdf

Simsion, G., *Data Modeling Theory and Practice*, Technics Publication, 2007, 416 pp.

Teorey, T. J., S. S. Lightstone, T. Nadeau, and H. V. Jagdish, *Database Modeling and Design: Logical Design*, Fourth Edition, Morgan Kaufmann, 2005, 296 pp.

Scamell, R. W., and N. S. Umanath, *Data Modeling and Database Design,* Course Technology, 2007, 720 pp.

CHAPTER 7

Physical Storage and Indexing

OBJECTIVES

- Briefly discuss the types of computer storage, primary memory, secondary memory and tertiary storage, available for storing a database.
- Discuss the types of disks, hard disks and RAID, available for storing and accessing data in a DBMS.
- Introduce buffer management.
- Discuss unstructured and unordered files.
- Explain ordered sequential files.
- Describe the concept of indexing and different types of indexes.
- Discuss indexed-sequential files.
- Discuss the B-tree data structure and algorithms for search, insertion and deletion.
- Derive expressions for B-tree's performance.
- Explain the B^+-tree data structure for storing databases on disk.
- Discuss static hashing schemes.
- Describe dynamic hashing schemes including extendible hashing and linear hashing for storing files on disks.
- Introduce inverted file and its uses.

KEYWORDS

Storage media, primary memory, secondary memory, tertiary storage, main memory, disks, RAID, efficiency of storage structures, files, records, unstructured files, sequential files, indexing, primary index, clustering index, secondary index, indexed sequential files, B-tree, insertion, deletion, search, B^+-tree as an indexed sequential files, performance of B-tree, internal hashing, external hashing, linear hashing, extendible hashing, inverted files.

There are seven sins in the world: Wealth without work, Pleasure without conscience, Knowledge without character, Commerce without morality, Science without humanity, Worship without sacrifice and Politics without principle.

Mahatma Gandhi (1869-1948)

7.1 INTRODUCTION

We have so far discussed the Entity-Relationship and relational data models and have discussed query languages such as relational algebra, relational calculus as well as SQL. In our discussions so far, we have not considered how the information is represented physically in a database system and how it is accessed and modified. The focus of this chapter is on techniques for physical storage and retrieval of information from storage.

Much information in a database system is resident on a disk and is transferred from the disk to the main memory of the computer when required. The computer's operating system carefully orchestrates modifications of data by moving data from disk storage to the main memory, modifying it and then writing it back to the disk. Disk storage is often called *persistent* or *nonvolatile* storage because the information stored on a disk can survive system crashes like a power failure even for an indefinite amount of time. Other types of nonvolatile memory are CD-ROM and DVD. These memories are nonvolatile as well as relatively cheap and relatively slow.

However, computers also need faster memories to store large amounts of information such as the instructions of the DBMS being executed, and the bulk of the data on which the users are accessing and modifying. Computers therefore have large memory, not quite as large as a disk, made from faster but more expensive technology. This memory called the *main* memory (and *cache* memory) is *volatile* storage because the contents of such storage are always lost with a power failure and other types of computer failures. The disk-resident nonvolatile information can also be lost although it happens only rarely. To protect against loss of nonvolatile information it is desirable to have a copy of the information backed up and stored carefully.

In addition to the main memory and the disk memory, computers have registers which are part of the CPU and which are usually constructed from the fastest available technology. This makes them quite expensive, and so each computer has only a small number of registers that are used mainly to hold information which is required frequently. In this chapter we will discuss the different types of computer memories and how information is stored in them by the DBMS.

This chapter is organized as follows. After introducing the different types of computer memories in Section 7.2, storage on a magnetic disk and RAID disk systems are discussed in Section 7.3. Section 7.4 describes how data transfer from and to disk takes place using buffers. Section 7.5 deals with basics of file structures. The following sections discuss different types of file organizations starting with unstructured and unordered files in Section 7.6 and sorted sequential files in Section 7.7. Index and index types including primary index, clustering index, secondary index, multilevel index and ordering of composite index, are discussed in Section 7.8 followed by a description of indexed-sequential files in Section 7.9. The most commonly used data structure in databases, the B-tree, is introduced in Section 7.10. The algorithms for searching, deleting and inserting keys are described here. Performance of the B-tree is theoretically analysed in Section 7.10.4. A modified B-tree, B^+-tree is described in Section 7.11. Section 7.12 deals with advantages and disadvantages

of B-tree and B^+-trees. Section 7.13 deals with static hashing followed by a brief discussion of external hashing in Section 7.14. Section 7.15 describes two dynamic hashing techniques, extendible hashing and linear hashing. The chapter concludes with a brief discussion of inverted files in Section 7.16.

7.2 DIFFERENT TYPES OF COMPUTER MEMORIES

We briefly discuss computer storage media which may be classified as follows:

1. Primary storage
2. Secondary storage
3. Tertiary storage

7.2.1 Primary Storage

Primary storage is directly connected to the CPU and is the fastest and most expensive. It includes the main memory, cache memory and the registers. These memories are volatile (that is, information is lost when the power is switched off) and therefore are suitable only for temporary storage of data.

The registers (usually less than 100) are usually the fastest and most expensive amongst the primary storage. The cache memory (usually around one MB) is also very fast and an expensive computer memory, often used to store frequently accessed data, thereby speeding up the execution of programs. There are a number of different types of cache memory. They are called L1, L2 and L3, depending on the nearness to the CPU with L1 being the closest. The main memory (or RAM—Random Access Memory) is the bulk of primary storage and can be as large as 4 GB or more. The data in the main memory, like that in the registers and the cache, can be directly accessed by the CPU. The access is quite fast although not as fast as the registers or the cache.

Normally, RAM is used to hold the instructions and the data for execution in a computer. As RAM can be altered very easily (as can data), it is ideal for this situation. RAM can be either *static* RAM or *dynamic* RAM (DRAM). DRAM is less expensive, uses less power and can be more compact than static RAM, although static RAM can be faster. Since CPU speeds have been growing quickly, faster RAM is needed to improve performance. One faster memory technology is RDRAM (Rambus Direct RAM) which is very fast and expensive; another is DDR RAM (Double Datarate RAM) which is also expensive like RDRAM.

7.2.2 Secondary Storage

The secondary storage devices used in the early computer systems included paper tapes, cards and magnetic tapes. Paper tapes and cards have disappeared now. Magnetic tapes perhaps are still in use at some computer installations but their use is often restricted to legacy files and archiving.

Secondary storage is not directly connected to the CPU. The CPU interacts with the secondary storage devices through controllers to read or write information. Secondary storage devices are used for more permanent and larger amounts of data. It is much cheaper than the cheapest primary memory which is the RAM. The

secondary storage devices include the hard disk (which is the most widely used), flash memory, CD ROM, DVD ROM, and USB mass storage devices.

All online processing now uses disks; even personal computers are based on disks of one type or another. The reasons for storing information on disk are twofold. Although the main memory has data that can be directly accessed by the CPU, the access is quite fast but is almost always *volatile* while the secondary storage is not volatile (sometimes called *persistent*). The other reason is the size of the secondary storage which is often orders of magnitude bigger than the primary storage (the computer I am using has one terabyte disk!) because secondary storage is cheaper than the primary storage (although large main memories are becoming more common). If the amount of information being manipulated is so large that it cannot be accommodated in the main memory, secondary storage must be used. Primary memory costs continue to decline faster than the cost of the secondary storage, and eventually the cost of primary and secondary storage will become similar. Diskless computer systems can become a possibility if the problems of archiving the database can be resolved. This will have substantial impact on the nature of file systems.

To permanently store data, secondary memory uses a magnetic or optical medium. The transfer of data from secondary memory to the CPU is slower than transfer from the main memory because the information has to be mechanically read from the secondary media. Magnetic disks are metal or plastic platters usually coated with ferric oxide. The oxide can be magnetised at different points on a disk's surface. The orientation of the magnetised spots defines the organisation of the information on the disk.

In order to store information on the surface of the disk effectively a format has to be defined in which the surface can be magnetised. A hard disk is typically a number of disks stacked on top of one another. The name hard disk comes from the centre of the disk being metal or possibly hard plastic. The sector method is typically used for single plattered disks. Each track on the disk is divided into a number of sectors, each of which stores some information. To access any information, the computer needs to know the surface number, the track number and the sector number of where the information is stored. More details on this are provided in the next section.

Different rules apply to different types of disks. We will list the major considerations for handling disks as well as special considerations for handling hard disks, soft covered floppies and hard covered floppies. One of the most important things to remember when handling a disk is that the read/write heads that put information on and get information off the disk are situated very close to the surface of the disk. When the disk is spinning at hundreds or thousands of revolutions per minute, any direct contact between the head and the disk will mean that substantial damage both to the disk and more importantly to the head will occur. Such a moment of contact of head and disk is called a *head crash*.

Magnetic disks, as the name suggests, are magnetic media and therefore the data stored on the disk will be lost if a strong magnet is placed near the disk. The disk will not be damaged in any physical manner but the data stored on the disk will be erased. Sources of strong magnetic fields include any speaker whether from stereo equipment or a telephone, and electric motors.

7.2.3 Tertiary Storage

Tertiary storage is the third level of computer storage whose defining characteristic is low cost. It is often separate removal media that cannot be accessed directly by the computer system. It is usually suitable for

storing large amounts of data for archival or backup purposes but is also being used in applications that generate huge amounts of data. The primary benefit of using tertiary storage is that it is often cheaper than secondary storage. It may be a stack of hard disks, optical disks like CD-ROM, DVD or even magnetic tapes. Tape is a very economical medium but access to tape is very slow. Tertiary storage devices are often sequential, not random access, as all primary and secondary storage devices.

Just as compact disks have changed the way we listen to music, optical disks are changing the way data is stored. Some benefits of optical storage over magnetic media are as follows:

- Higher capacity than magnetic storage
- Cheaper
- More compact
- More versatile

Data is stored on the surface of a metallic disk by using a series of lasers. One laser hits a metallic surface spread over a disk and burns tiny spots into the surface. Another laser can then be used to scan these spots and the differences in light reflections can be detected.

CD-ROM

They are plastic disks sealed with a protective coating. They are written from centre to the outside edge using a single spiralling track. Data is stored in 2352-byte sectors along the track. A CD ROM is a low cost storage option and is also easily available. It is widely used for archiving purposes. A CD ROM can store about 700 MB of data.

DVD

DVDs (or Digital Versatile Disks) can store much more information than CD-ROMs. A single-side DVD can store 4.78 GB of data.

Magnetic Tapes

Magnetic tapes have been around for a long time. They resemble conventional audio cassettes which have essentially disappeared now. The earlier tapes were more like tapes that were used on very old tape recorders. Tapes were one of the first storage systems used on computers. They are cost-effective storage systems that provide large cheap storage but are quite slow in reading and writing data. They are most commonly used for backup and archiving databases.

7.2.4 Summary of Computer Storage

A simple computer memory hierarchy representing different types of memories is shown in Fig. 7.1

The primary reason for having several levels of memory hierarchy is to obtain good performance economically. Database systems do not access data uniformly and most users spend majority of their time in one small part of the database. If this data can be moved to a faster (and more expensive) memory, a DBMS's performance is likely to improve significantly without using a large amount of expensive memory. For this reason the registers and the cache memory store the most frequently used instructions and data. This is somewhat similar to a browser cache on a computer which stores the frequently used pages in the browser cache thereby saving time to retrieve pages from the Web.

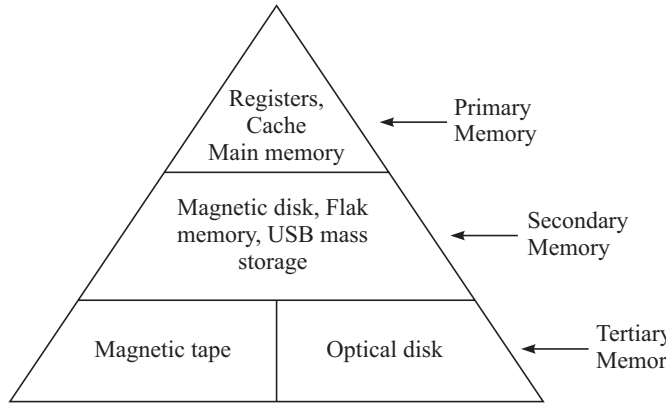

Figure 7.1 Memory hierarchy

In Table 7.1, we present another view of the memory hierarchy given in Fig. 7.1. There are of course other types of memories as well. Recently flash memory (for example, USB memory sticks) has become very popular.

Table 7.1 Different types of memories and their characteristics

Type of memory	*Size*	*Speed*	*Cost*
Registers	Very small	Very fast	Very high
Cache memory	Small	Very fast	High
Main memory	Large	Fast	Moderate
Disk storage	Very large	Medium	Cheap
Tertiary storage	Very large	Slow/Medium	Cheap

Before discussing each type of memory we note that memory costs have been going down for many years. It therefore follows that computer manufacturers can put larger and larger memories of each type in contemporary computers. The total annual disk storage sales over the last few years have been growing rapidly as shown in Fig. 7.2. Total annual disk sales in 2003 were almost 16,000 petabytes (or 16 exabytes) of storage compared to only around 4000 petabytes in 2000. Some of these disks, one assumes, are for storing increasingly large databases. Annual disk sales in 2003 were close to 3 GB per person in the world per year! In a keynote presentation at the 2005 International Conference in Data Engineering (ICDE) it was suggested that 250MB of data was being generated every year for each person in the world and the cost of disk storage was now much lower than cost of storing information on paper. Secondary storage is discussed ahead in more detail.

7.3 SECONDARY STORAGE–HARD DISKS

The first disk was invented by IBM in 1956. The current disk drives are very different from the first disks. The 1956 disk stored 5 MB of characters. Today's disks can perhaps store 100,000 times more in a very

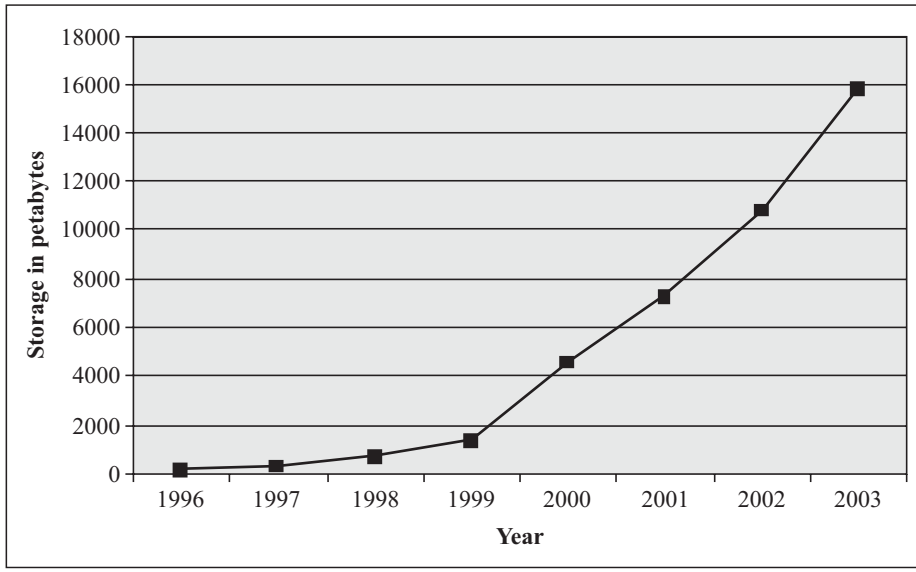

Figure 7.2 Annual disk storage sales in Petabytes

compact and much cheaper disk assembly. We first briefly discuss magnetic disk technology and then the RAID technology.

7.3.1 Magnetic Disk Technology

A hard disk normally consists of a spindle holding one or a number of hard rotating disk platters that usually spin at up to 10,000 revolutions per minute (RPM). An extremely thin polished magnetic coating is layered onto the surface of the platter. The platter usually cannot be bent or flipped and hence it is called a hard disk. Modern disks usually consist of a number of platters that use both surfaces for storing information.

A disk consists of bits that are very small magnetized areas. The bits are so small that a disk with 5" diameter can store more than a 100 GB of information. The storage density appears to be growing at the rate of 100% per year and the cost of hard disk in USA is now 0.1 per cent megabyte. Even a disk with a terabyte (1000 GB) is now available for less than Rs 5000. As noted above, disk capacities have been growing and price/MB continues to fall. A single platter 1.8 inch disk with 160 GB capacity has been announced in late 2009. A 3.5 inch single platter disk with capacity of 500 GB is also now available (at the time of writing) and it is expected that single platter disks with two terabyte capacity might be available soon.

As shown in Fig. 7.3, a disk has a number of concentric circles called *tracks* which often consist of *sectors* which are normally 512 bytes of data, the basic unit of data storage on the disk. Each sector also stores information about the sector's id, error correcting code as well as additional information required by the disk drive controller. For convenience, most disks store the same number of bytes on each track although the size of the inner tracks is smaller than the outer tracks. Each track has the same number of sectors and each sector has the same number of bytes. The tracks and sectors within them are numbered. The outermost track is numbered track 0, the next track is numbered 1 and so on. There can be more than a thousand tracks on a small disk. Sector numbers depend on the disk controller and may start with 0 or 1 or even 129.

Information is stored in tracks so that it can be read while the disk is rotating. Sectors may be combined into blocks by the operating system so that the block size on the disk is same as used by the system. This is sometimes called formatting the disk. Therefore a disk formatted by one operating system cannot usually be read by another operating system unless the disk is reformatted. The file on a disk is divided into blocks to save space as well as improve efficiency. A block is a fixed size of information (often 512 bytes to 4 KB) that may be read from the secondary storage at one time (reading one block is called a disk access). Often a block is much bigger than a record in a file. There is however no reason why in some applications a block can not be smaller than a record if the records are very large, for example, records that include images.

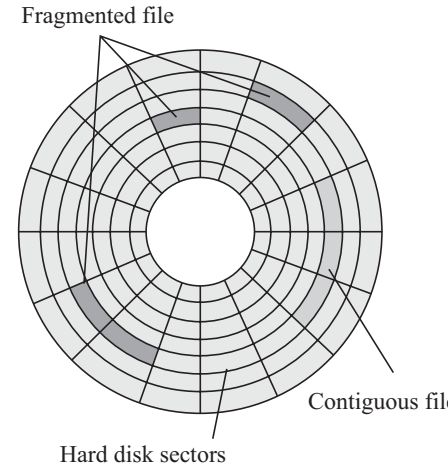

Figure 7.3 Organization of files on a magnetic disk

A hard disk normally contains 1 to 10 identical platters that are stacked on a spindle to form a cylinder which is all the tracks directly above and below each other. There is usually one Read Write (RW) head designated per platter face which hovers close to the face of the platter never touching the platter, and all the heads move in unison and carry out reading or writing of data. If a head ever touches a platter then, as noted earlier, it is called a head crash. When the disk is turned off, the heads retract to a side by the disk mechanism and are parked away from the platter surfaces.

Accessing a disk involves the following:

Seek Time

This is the time it takes for the disk head to position itself over the required track. Some very high performance disks eliminate this time by providing a disk head for each track. Such disks are called *fixed head disks*.

Rotational Delay

This is the waiting time it takes for the required sector to come under the disk once it is positioned over the required track.

Transfer Time

This is the time it takes the disk head to read the information and transfer it.

7.3.2 RAID (Redundant Arrays of Independent Disks)

A redundant array of independent disks or RAID is a single logical disk system that uses a stack of magnetic disks to share or replicate data amongst the disks. The initial idea, proposed in 1988 by David Patterson, Garth Gibson and Randy Katz at Berkeley, was to develop a disk system with an array of small, inexpensive disks to replace large expensive disks that were being used with large mainframe computers. Furthermore, the idea was to improve reliability and improve performance of disk systems since a disk head crash meant

that all the data on the disk was lost and the file system or the database had to be rebuilt from the point of the last backup.

RAID technology therefore combines the following two ideas:

1. Multiple inexpensive disks accessed in parallel to give higher throughput.
2. Storage of redundant data to provide improved fault tolerance.

The second idea is just as important as the first since cost of disk failure is high and using a suitable redundant data scheme, RAID can provide a much higher fault tolerance.

Therefore, the RAID approach provides increased capacity, increased throughput, improved data, improved integrity and improved fault tolerance. RAID disks can provide a vast storage capacity. As the CPU performance continues to increase, increased disk throughput is required to overcome the disk I/O bottleneck. RAID technology is a part of the solution of this bottleneck.

Since the RAID systems have many disks, it makes sense to chop every file (in particular, the large ones) into pieces and store these pieces across the entire disk drives in the system. This is called *striping*. RAID disks use the concept of striping files into segments of logically sequential data so that the segments may be written across different disks to improve reliability. Striping can be done at the byte level or at block level.

There are a number of levels of RAID implementation, proposed to go from RAID level 0 to RAID level 10 although the original paper suggested only levels up to 5. We list the first five levels as well as level 10 below.

• RAID Level 0—Striping but no redundancy

RAID 0 is a scheme that uses a striped disk array and distributes data segments across all the different disk drives. It achieves maximum throughput with no redundancy. Because of lack of redundancy, there is no disk overhead but level 0 is not a true RAID system.

• RAID Level 1—Mirroring and Duplexing

RAID 1 is the simplest RAID system. It provides complete redundancy by storing mirror copies of all files on a pair of disks with writing of duplicate copies carried out in parallel. 100% redundancy means that any single disk failure does not result in loss of data and the system can continue to perform without any down time. Level 1 involves high disk overhead because of 100% redundancy. Level 1 is suitable for many database systems that need 100% mirroring.

• RAID Level 2—Error Correction

This RAID system stores the Hamming Code of each data word on a separate disk. Level 2 is not used in commercial database systems.

• RAID Level 3—Parallel Transfer and Bit-interleaved Parity

This is a high performance RAID system. It involves striped segments written on two or more disks in parallel and an additional disk to store parity information. It is similar to RAID 2 but is more economical and uses only one drive for data protection. It is not suitable for transaction-oriented databases.

• RAID Level 4—Block Interleaved Parity using Shared Parity Disk

This is not used widely because if implemented it would perform poorly. It uses stripes of uniform size, storing them all across the disk drives. It also uses a parity disk. One could think of it as RAID 0 plus a parity disk to provide some fault tolerance.

• RAID Level 5—Block Interleaved Distributed Parity

Perhaps the most widely used RAID system that is similar to RAID 3 except read and write do not take place in parallel. Parity information is spread across all disks. RAID 5 is a high performance disk system. It is suitable for most database systems.

• RAID Level 10—Striped Mirrors with High Reliability and High Performance

The concept of RAID 10 combines the features of RAID 0 and RAID 1. RAID 10 may be able to sustain more than one disk failure at the same time. It is suitable for high performance and fault tolerant databases.

7.4 BUFFER MANAGEMENT

We briefly discussed buffer management in Chapter 1. In this chapter we have noted that transfer of data between the main memory and a disk takes place in units of *blocks*. The disk is divided into blocks (or pages), usually of size 4 KB or 8 KB. When data is read from a disk drive, it is copied to the main memory which is itself divided into blocks to enable data to be copied into a block. This area is called a *buffer*. To read data, data from the disk is read into a main memory buffer. To write to the disk, the data in the main memory buffer is copied to a block on the disk. When several blocks need to be read from the disk, a main memory must allocate a number of buffers from the buffer pool for the transfer. This allocation of main memory for buffers is handled by an operating system module called *buffer manager* which makes decisions like how many buffers to allocate, which buffer block needs to be replaced when new data is being transferred from the disk. The buffer manager maintains a variety of information for each block in the buffer pool. This includes the number of times each buffer has been accessed as well as a flag if the buffer has been modified. Modified buffer blocks are often called *dirty.* There is a considerable range of techniques that a buffer manager uses in deciding which buffer block must be replaced when a new block is read from the disk and there is no free space in memory. We briefly discuss techniques for replacing blocks when a new request needs a block.

A number of techniques have been proposed for replacing a buffer with the aim of reducing the disk traffic and enhancing system performance. The two most commonly used techniques are LRU and FIFO described below.

LRU—Least Recently Used

The buffer manager may use the LRU approach to replace an old buffer that has not been used for the longest time. This can be done by the buffer manager maintaining a list or queue of buffers in the memory that has been inactive for the longest time. The basic idea behind LRU is locality of reference which implies the following:

- Pages that have been requested recently will very likely be requested again soon.
- Pages that have not been requested recently are unlikely to be requested soon.

The LRU scheme is not as simple as it sounds since the queue will change every time one of the buffers on the LRU queue gets used in a read or a write. This can be done by replacing the buffer at the front of the queue and moving the buffer to the back of the queue. Some schemes also include the free buffers in the LRU queue while others maintain a separate free list. Some other schemes maintain more than one LRU queue so that every request for buffer does not need to go to the same queue.

FIFO—First In First Out

This scheme is based on the idea that the age of the buffer is the most important and therefore the oldest buffer page should be replaced when another buffer is requested. FIFO may be implemented by a simple queue in which the oldest block is at the front of the queue and the most recent at the back of the queue. When the next buffer is requested, the front of the queue is replaced and the new block is put at the rear of the queue. FIFO is therefore the simplest buffer replacement scheme but it may create problems in some situations. Databases can have hot spots but FIFO will keep replacing the hot spot pages after some time, resulting in another buffer request being made for the recently replaced page which will have to be reread.

Studies have been carried out to compare LRU, FIFO and other buffer replacement schemes. In one study the LRU and FIFO schemes were compared and the LRU scheme was found to be close to optimal if access probabilities were known but was always better than FIFO if the probabilities were not known.

Much of the DBMS implementation does not need to know anything about buffer management, since it is only the lower level modules managing the files and main memory that need to directly deal with the buffer manager. A variety of information is transferred from the disk to the main memory and main memory to disk including the database files, the database metadata and the database indexes.

A database normally has a number of structures available to store the data that it contains. The structure chosen would normally depend on the application and most DBMS provide the DBA with a number of options for physical storage. These are discussed in detail later in this chapter.

We now discuss simple file structures. These include unstructured and serial files presented in Section 7.6 and sequential files in Section 7.7.

7.5 INTRODUCTION TO FILE STRUCTURES

A database consists of a variety of files and a file consists of a collection of records. It is therefore important that files be stored efficiently. We first define a record and a file.

Definition—Record
A record consists of a non-empty set of attributes that identify it uniquely (often called the identifier or the key) and a set of related values (often called the fields).

Definition—File
A file is a collection of similar records usually stored on a secondary memory device.

As an example of records and a file consider a table. If the table is stored as a file, each row of the table is then a record. Files may consist of fixed-length records or variable-length records. In relational database systems when a table is stored as a file, it is always a fixed-length record file.

Efficiency of Storage Structures

The most common operations on files are updates, insertions, deletions and searches. Therefore to determine efficiency of a storage structure the factors listed in Fig. 7.4 are commonly evaluated.

1. *Cost of insertion*—Usually an insertion operation involves inserting one row in a table. This includes identifying the table in which the new row will be inserted in and where. It involves fetching the block where the row will be inserted, inserting the row in it and writing the block back. It may require reading, modifying and writing blocks of data.
2. *Cost of retrieval*— Retrieval can involve a variety of data retrieval. The simplest retrieval involves fetching all rows of a table that satisfy an equality condition. For example, finding all *Player IDs* for players with scores over 99.
3. *Cost of deletion*—Deletion may involve deleting one row of a table or deleting a set of rows meeting a given condition. This involves identifying the table that has the row or rows to be deleted, then fetching the block or blocks that include rows to be deleted, modifying them and writing the blocks back.
4. *Cost of update*—Update may involve updating one row of a table or may involve updating a set of rows meeting a given condition. A number of blocks may have to be read, modified and written back.
5. *Storage utilization*—Although storage is now relatively cheap, the cost of storage is still a consideration for large databases.
6. *Cost of report generation*—Some types of reports can be very time consuming.

Figure 7.4 Factors that determine efficiency of a storage structure

We will discuss these efficiency measures for each file structure we study. The efficiency will normally be measured by the number of disk accesses since that often dominates the cost of typical database applications, for example, retrieval. We will initially assume that a retrieval involves only simple queries, for example, finding the record with key value *k*. A technique called indexing is often used for efficient retrieval. We will discuss indexing later in this chapter.

Other types of queries are frequently posed, for example:

- Find names of all players that have played in the ODI match 2755 (sometimes called *associative* search).
- Find IDs of all players who scored between 75 and 86 runs in the ODI match 2689 (called *range* query) or find all players who have scored more than 50 in a match.
- Find IDs of all players who scored below the mean score in the match 2755 (called *function* queries) or find the maximum number of runs scored by each player in all the matches in the database.
- Find all the articles from a document database that have "index" or "file" or "hashing" in their title (called *partial match* query) or find all the articles that have words starting with "comput".

Even more complex queries may sometimes be posed but we will not consider techniques for dealing with them here.

7.6 UNSTRUCTURED AND UNORDERED FILES (HEAP FILES)

These are the simplest and often the most inefficient techniques for storing information in files. These files have a number of names, for example, unstructured files, unordered files, serial files or heap files.

An unstructured file may be used for storing raw data, for example, data dumped directly from some sensors or the computer memory. It may consist of records stored one after another as the data arrives. The records are assumed to have no fixed structure or length. Each record therefore needs to include some information about its length as well as its attributes. If the records in the file do have a fixed structure, but the records are stored one after another as they arrive, the file is called a *serial file*.

A serial file is also usually a sequence of varying length records. As data is written to the file, records are added at the end of the file. The records are normally accessed sequentially. Any data may be written to a serial file. Byte count is usually used to determine the size of each record. The record size is saved at the start of each record. To retrieve information from such a file, some attribute values would have to be specified. These may then be searched in the file sequentially.

Unstructured files are useful for collecting information before the data is organized based on the requirements of an application. Unstructured files are also sometimes used when data is collected without any particular application in mind, for example, text retrieval, military intelligence, etc. Serial files also have similar applications and are often used for storing temporary or historical data.

7.7 SEQUENTIAL FILES (SORTED FILES)

Sequential files differ from the unstructured and serial files as in addition to the records having a similar structure (that is, the fields are in the same order and often of the same length), the records are arranged in position according to some key order. Two familiar examples of sequential files are a dictionary and a phone book. We give an example of a sequential file *Player* for the cricket database. Only a part of the file is given in Fig. 7.5. The file is ordered on the attribute *LName*. We assume the relation has been stored as a sequential file with each row being a record.

PlayerID	LName	FName	Country	YBorn	BPlace	FTest
96001	Dravid	Rahul	India	1973	Indore	1996
94004	Fleming	Stephen	New Zealand	1973	Christchurch	1994
96002	Gibbs	Herschelle	South Africa	1974	Cape Town	1996
95003	Kallis	Jacques	South Africa	1975	Cape Town	1995
93003	Kirsten	Gary	South Africa	1967	Cape Town	1993
90002	Kumble	Anil	India	1970	Bangalore	1990
90001	Lara	Brian	W Indies	1969	Santa Cruz	1990
92002	Muralitharan	Muthiah	Sri Lanka	1972	Kandy	1992
95002	Pollock	Shaun	South Africa	1973	Port Elizabeth	1995
95001	Ponting	Ricky	Australia	1974	Launceston	1995
93002	Streak	Heath	Zimbabwe	1974	Bulawayo	1993
89001	Tendulkar	Sachin	India	1973	Mumbai	1989
92003	Ul-Huq	Inzamam	Pakistan	1970	Multan	1992
94002	Vaas	Chaminda	Sri Lanka	1974	Mattumagala	1994
99003	Vaughan	Michael	England	1974	Manchester	1999
92001	Warne	Shane	Australia	1969	Melbourne	1992

Figure 7.5 A sequential file ordered on attribute *LName*

The main attraction of sequential organization is that it is conceptually simple and provides efficient access to successive records. Sequential files were originally developed for storing information on magnetic tapes. The structure has continued to be popular for storing files on disks because of its simplicity.

To retrieve a record from a sequential file, the entire file is normally scanned until the desired record is found. For a large file, binary search could be used. Even if binary search is used, several disk accesses will be required necessarily before the desired record is found.

Inserting a record in a sequential file requires that the record be inserted into its proper place according to the key. Therefore insertion of a record involves the insertion point to be identified, the records after the insertion point to be copied, the new record to be inserted and the copied records to be written back after the newly inserted record. Consider the case of adding a new player, say M S Dhoni or Ishant Sharma to the file shown in Fig. 7.5. A similar algorithm must be used in updating records if the file is not ordered on an attribute that is unique. Deletion of a record also leads to similar problems although a simple technique that is often used is to leave the deleted record on the disk but mark it as deleted. The record can then be removed when the next major update (sometimes called the *master update*) is carried out.

Insertion, update, deletion and retrieval of a single record at a time are therefore expensive when the sequential file structure is used. The cost of insertions may be reduced by keeping a separate small file (called the *overflow area*) of insertions. This is merged with the main file when the next major update is carried out. Another technique for overcoming the problem of expensive insertions, updates and deletions is to accumulate transactions, say for one day, and then sort them according to the same key as the file at the end of the day. All the sorted transactions (called the *transaction file*) are then processed together. Such batching techniques can make the sequential file structure very efficient. The structure however is not appropriate when an application requires quick response from the system (e.g., online applications).

The main advantages of sequential files are as follows:

- Sequential files are easy to implement.
- They provide fast access to the next record using lexicographic order.

However, sequential files have the following major disadvantages:

- Sequential files are difficult to update. Inserting a new record may require moving a large proportion of the file.
- Random access in sequential files is extremely slow.
- Sequential files are not appropriate for online applications.

Sometimes a file is considered to be sequentially organized despite the fact that it is not ordered according to any key. The date of acquisition may be considered the key value; the newest entries are added to the end of the file and this poses no difficulty to updating.

As noted above, the performance of a sequential file is usually poor if an application requires random processing of records. The primary technique used for enhancing the performance of sequential files is based on addition of an index to the file. We discuss indexing and index types in the next section.

7.8 INDEX AND INDEX TYPES

A sequential file with an associated index is like an old library card catalogue and is called an *indexed-sequential* (or ISAM) file.

Definition—Index
A file index is a data structure that improves the speed of common operations on the file. Indexes can be created using one or more columns of the file.

Indexes assist in rapid retrieval of records. Index often only stores values of the key or some other attribute(s) of the file and therefore the memory space required to store an index is substantially less than that required to store the whole file. This can sometimes lead to an index being stored in the main memory for a large file stored on disk, further improving the efficiency of file processing.

Several variations of indexed-sequential files exist and we will discuss one commonly used simple version that involves an index as well as an overflow area. As noted earlier, the overflow area provides a facility to handle insertions more efficiently than an ordinary sequential file.

In addition to the indexing technique ISAM, a commonly used indexing technique is based on a tree structure called *B-tree*. We will discuss both techniques later in this chapter.

The choice of a file index can have significant impact on the performance of a database system, especially if the database tables are large. The type of index should normally be decided based on the types of queries that are made to the given database system. For example, if we look at the type of queries discussed above, a simple query involving an equality condition may require a different type of index than range queries over the same table. The situation becomes more complex if queries involve more than one condition, for example, find rows from a table *Batting* such that they have scores of more than 50 in match number 2755.

In the simplest case of indexing, the index in an indexed-sequential file consists of one entry for each data record in the sequential file. This is called *full indexing*. Since a record in a file usually would have substantially more information than its key, the index would still be much smaller than the file, as each entry in the index would consist only of a key value and an address. (The index is normally effective only when the file is large with a substantially smaller index so that the index can be a main memory resident or can be read from the disk in a few accesses at most.) If the index is large, it is common to have a higher level index so that the appropriate part of the index can be accessed directly after the higher level index has been accessed. In practice, the size of a higher level index would often depend on the size of buffers and disk blocks. Also, there may be more than two levels of indexing if the file is very large. More details of multilevel indexes are presented later in this section.

Another approach of reducing the size of an index and keeping the size as small as possible is based on maintaining one index entry for a group of records rather than for each record. Since every time the disk is accessed, a block of records rather than a single record is usually read, one index entry for each record is not really necessary. Instead one index entry for each block is sufficient. This is sometimes called *partial indexing*. An example of partial indexing is a dictionary as shown in Table 7.2. The index in this file (that is, the dictionary) is very small; one entry for each letter of the alphabet. One could consider a dictionary as having a two-level index, the high level one for each letter and then the lower level index for each page.

Table 7.2 Example of an index used in a dictionary

Index	*Dictionary*
A	All words starting with letter A
B	All words starting with letter B
C	All words starting with letter C
D	All words starting with letter D
E	All words starting with letter E
F	All words starting with letter F
G and so on	All words starting with letter G and so on

We first define some terminology for indexing. A *primary index* is specified on the ordered key attribute of a table. In this index it is clear that the values of the key attribute are unique and so indexing is relatively simple. A primary index normally is built on the primary key of the table. If the indexing attribute does not have unique values then the type of index is somewhat different. It is called a *clustering index*. It is assumed that the indexing attribute is still ordered. For some attributes ordering may not be possible, for example, the file could be ordered according to the primary index although it is required that there be another index on another attribute that is not unique. The second index is then called a *secondary index*.

These indexing techniques are now discussed.

7.8.1 Primary Index

As noted above, a primary index is specified on an ordered key attribute of a table. In this index the values of the key attribute are unique. An example of the primary index is shown in Table 7.3. The index consists of two fields, viz., the index entry and a pointer to the block on the disk.

Table 7.3 An example of a primary index

Highest Record Number on Track	*Track Number*
20	1
60	2
85	3
120	4

Therefore, to search for a row with key 90 we find the first entry in column 1 that is bigger than 90 (that is, 120) and go to the track number given (that is, 4). We can then search for 90 on that track. It should be noted that this index is much smaller than the table which probably has many attributes and therefore each row of the table is much larger than the attributes on which the index has been built.

7.8.2 Clustering Index

If the values of the indexing attribute are not unique but the attribute is still ordered and is searched frequently, a clustering index may be used to improve searching on the attribute. An example of the clustering index is shown in Table 7.4. It also consists of two fields, viz., the index entry and a pointer to the block on the disk.

The index shown in Table 7.4 usually has an entry for each record and therefore the index is relatively large but much smaller than the actual file. The entries in Table 7.4 indicate that for the attribute value 22,

track number 3 should be searched which might have several records with the attribute value 22. For an attribute value 25 the index shows that there is no record with such value.

This type of index is expensive to maintain as records are updated, inserted and deleted but the index is efficient for a variety of queries, for example, searches, scans and range queries on the indexed attribute. Clustering indexes therefore should be used only where the cost of maintaining them can be justified by the savings in queries.

Table 7.4 A simple example of a clustering index

Clustering Attribute Value on Track	*Track Number*
15	1
18	2
22	3
29	4

7.8.3 Secondary Index

If the values of an attribute that justifies an index are not ordered, a secondary index may be used to improve searching on the attribute. As noted earlier, it is assumed that a primary index already exists on the search key which specifies the sequential order of the file. All other indexes on the file, of which there may be several, are called *secondary indexes*. The secondary index may be on a nonkey attribute which may also have duplicate values. A secondary index also has two values, viz., the indexing field and a pointer to the block or to the appropriate record. The secondary index then looks like that shown in Table 7.5. Since the secondary index attribute may have duplicate values, the index must take that into account by perhaps creating an entry for each duplicate value. Although the file is not ordered according to the secondary index attribute, the index is usually ordered according to the attribute so that the index can be searched quickly. It should be noted that the secondary index requires all values of the index attribute to be included in the index and therefore the index is usually relatively large and requires a longer search period than a primary index.

Table 7.5 An example of a secondary index

Secondary Attribute Value on Track	*Track Number*
15	1
18	2
25	1
49	4

It should be noted that a table may have one primary index and more than one secondary index, if appropriate.

As an example, with the file *Player* it may be desirable to build a primary index on *PlayerID* and a secondary index on *LName* while in the file *Match* it may be appropriate to have a primary index on *MID* and secondary indexes on attributes *Ground* and *Date* but this should be done only after a careful analysis of the likely queries.

Secondary indexes improve the performance of queries that use attributes other than the attribute used in the primary index, but the overhead costs of indexing are higher.

7.8.4 Multilevel Index

Since some indexes can be very large, as noted earlier, a multilevel (two levels or more) index is sometimes created to improve searching the index. If the index is stored on the disk then a multilevel index can substantially reduce the number of disk accesses needed by first retrieving the highest level index and then the lower level indexes until an entry is found in the lowest level index.

An example of a multilevel index is presented in Fig. 7.6. The index looks like a tree, with the top node being the highest level index which is searched first (it may be main memory resident) and then the appropriate block of the next level index is read from the disk followed by reading the appropriate block at the lowest level of the index, which then points to the block (the index in Fig. 7.6 is a very simplified version of an index that does not show pointers) that has the record.

Consider an example of searching using this index. Suppose we are searching for 60. First the top level of the index is searched[1]. Since 60 is larger than 50, we go to the next level on the right and search that part of the index which has only 70 and 90 keys. So the search continues to the left of 70 and the next level of the index is accessed. We find 60 and then follow the pointer (not shown in the figure) to access the block on disk which has the record.

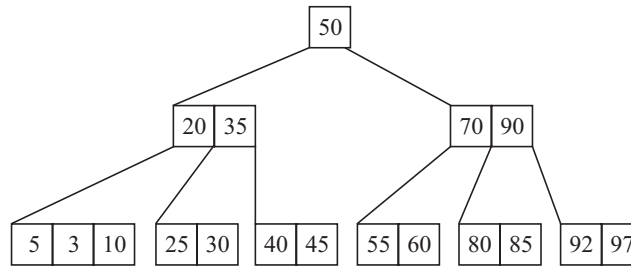

Figure 7.6 A multilevel index

Once again we emphasize that Fig. 7.6 is a simplified version of an index that would be used in practice. It does not show any pointers and each node is small with a maximum of only 3 keys.

7.8.5 Ordering Composite Index

Often a query would require search on more than one column when the WHERE clause has two conditions. To speed up such a query, a composite index that is made up of more than one column may be useful. In some cases, a composite index may include all the columns of a table. Such index is called a *covering index.* To speed up a query, the index should be on the columns that match the columns used in the WHERE clause. Composite indexes tend to be much larger than a single column index because, for example, the number of distinct values of, say (*PID*, *MID*), would be much larger than for *PID* or *MID* alone.

When a query uses a composite index, the order of the columns of the key in the index is important. The index is usually most efficient when the composite index has the columns in order from most to least distinct. Therefore, the column with the most number of distinct values should come first in the composite index.

Let us consider an example. Suppose several queries are posed on the cricket database about the number of runs scored by a batsman in a match. It has been decided to set up an index on *MID* and *PID*. Should this

1. Usually a node in the index would be bigger, for example, equal to the size of a block and include many values of the indexed attribute as well as pointers to the next level indexes. We assume that the pointers to the records are included only in the leaf nodes and therefore 50 does not point directly to a record. Furthermore, we have not stated whether this multilevel index is a full index to the file or a partial index.

index be (*PID*, *MID*) or (*MID*, *PID*)? We can make a decision if we look at the data in the database. Assume that the database includes information on all the ODIs that have been held so far. So we have 3000 different values of *MID*. Further assuming that many matches have 22 batsmen records when everyone bats while some have only, say, 10 batsmen records when only a few batsmen bat, in each of the two innings. Therefore, we may assume that there are on average 15 batting records for each match, giving us a total of 45,000 batting records in the table *Batting*. How many distinct values of *PID* are there? Let us assume there are 500, so that on average each player has 90 batting records. Some players have played more than 400 ODIs while many have played well below 100.

When a query involves *PID* and *MID*, we want to use the index to quickly get these records. If we use *PID* first then we narrow the search to 90 rows in *Batting* on average but it may be close to 400 for some values of *PID*. If on the other hand we use *MID* first, then the search is narrowed to no more than 22 records. Therefore, it is better to set up the index as (*MID*, *PID*).

7.9 THE INDEXED-SEQUENTIAL FILE

The indexed-sequential file organization has been successfully used in many business applications. IBM's well known ISAM file organization was based on using an ordered primary file and a two-level index. As described below the ISAM file structure uses an overflow area.

The organization of an indexed-sequential file is discussed below in more detail. The file usually consists of the *prime area* (the main file), the *overflow area* and the *index area*. When the file is created or reorganized, all the records reside in the prime area and the overflow area is empty. As in the sequential organization, records in an indexed-sequential file are ordered by a key value and therefore a primary index is being used as discussed above.

We illustrate the organization by a simple example. We assume that the size of the records is such that each block (we will also use the term *track* for block since normally a block is the size of a track) of the disk on which the file is resident stores only four records. Let the prime area of the file at some instance be as shown in Table 7.6.

Let the overflow area at this instance be empty. The track index then looks like Table 7.7.

Table 7.6 Records in the prime area of an indexed-sequential file

Track Number	*Records*			
1	REC10	REC14	REC18	REC20
2	REC26	REC31	REC41	REC60
3	REC72	REC74	REC75	REC76
4	REC77	REC78	REC79	REC82

Table 7.7 Index of the indexed-sequential file in Table 7.6

Track Number	*Highest Record Number on Track*
1	20
2	60
3	76
4	82

To locate record number 79, we first consult the index in Table 7.7 and find that record numbers greater than 76 and up to 82 are on track 4. Track 4 is read in the main memory and a search for record 79 is carried out. The search terminates successfully if the record is located in the block, otherwise the search terminates unsuccessfully.

Using the index does not completely eliminate the sequential search (as the block must still be searched sequentially) but it does reduce the magnitude of the search substantially. We now only need to search one track of records rather than the whole file.

The above search process was particularly simple since there were no overflow records. In a dynamic file, there will be deletions and insertions and almost always some of the tracks will overflow. The search procedure is then somewhat more complex and we discuss it now using an example.

Let us assume that record numbers 15, 30 and 90 are inserted to the file above and assume that track 5 has been designated as the overflow area. The prime area after inserting the three records then looks like the file in Table 7.8.

Table 7.8 An indexed-sequential file with overflow area

Track Number	*Records*			
1	REC10	REC14	REC15	REC18
2	REC26	REC30	REC31	REC41
3	REC72	REC74	REC75	REC76
4	REC77	REC78	REC79	REC82
The overflow track has the following records:				
5	REC20	REC60	REC90	

Note that the key sequences in each block have been preserved when records have been added although sequence preservation generally incurs additional costs. For example, the addition of record number 30 above required reading and writing track number 2 as well as track 5.

The index must now include some information about overflow records. The index may appear as shown in Table 7.9.

Table 7.9 Index for an indexed-sequential file with overflow area

Track Number	*Highest Record Number on Track*	*Highest Record Number in Overflow*	*Address of First Overflow*
1	18	20	5
2	41	60	5
3	76	Nil	Nil
4	82	90	5

Indexed-sequential files can be very efficient, however, the technique can become inefficient when the file is dynamic, since deletions and insertions are difficult and regular reorganization of the file becomes necessary to preserve efficient operations. We now consider a technique that is more suited to dynamic files.

7.10 THE B-TREE

The indexed-sequential file is not a particularly satisfactory solution for dynamic files although IBM's ISAM system using overflow areas became very popular during the 1960s and 1970s. As noted earlier, ISAM used a two-level index which may be considered as a shallow tree with the top level index being the root of the index tree and the next level being the tree leaves. We now consider another tree data structure for the index. This tree structure does not have a fixed number of index levels. It allows as many levels as are needed and allows the index to grow and shrink gracefully.

We first consider the classical B-tree structure which combines the index and the file. Later we will look at modifications of this structure that may be more suitable for use as an indexing technique.

We assume that the reader is familiar with binary search trees. The binary search tree is not suitable for storing information on disk due to the following reasons:

(a) In a binary search tree, one node of the tree is examined at a time and the node to be inspected next depends on the result of the comparison at the present node. This process would, on average, require reading ten nodes to search a file of only 1000 records.
(b) As noted earlier, information stored on a disk is read one block at a time. One block would normally include several nodes but the binary tree scheme does not readily take advantage of this feature of reading disks.

The B-tree is a multiway tree which appears like that in Fig. 7.7 below.

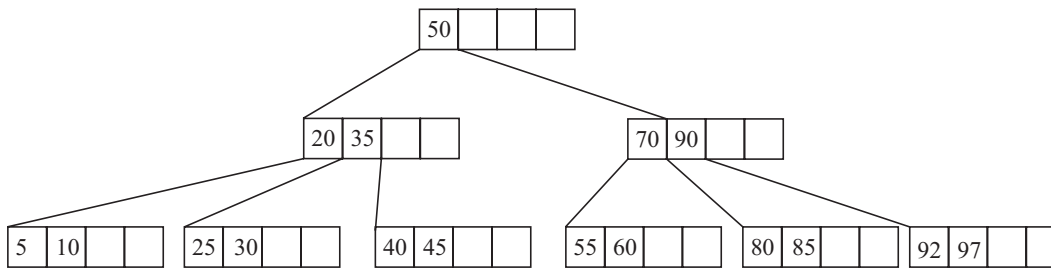

Figure 7.7 A typical B-tree

This tree shows each block having a capacity to store four keys or records. A block with k keys has $k + 1$ pointers to its children. We will consider B-trees in more detail soon.

Figure 7.7 can be looked at as an attempt to modify the binary tree structure so that the above problems are overcome. The technique includes a number of records in one node; the node size is usually equal to that of a block.

We first consider the binary tree in Fig. 7.8. In an attempt to store the tree on the disk, we may partition the binary tree as shown.

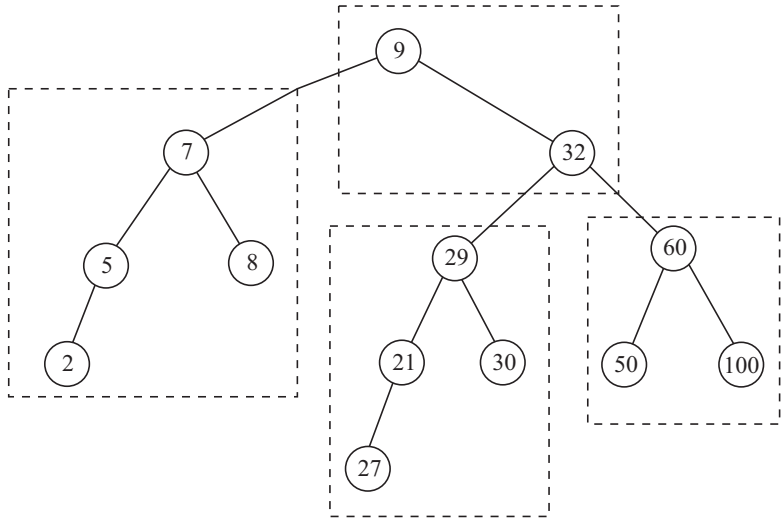

Figure 7.8 A binary search tree

The binary tree in Fig. 7.8 can be represented as a multiway tree in which each node is allowed to have more than one key. The tree is presented in Fig. 7.9.

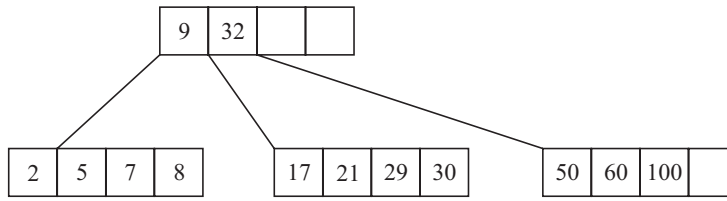

Figure 7.9 A B-tree representation for the binary tree of Fig. 7.8

The multiway tree in Fig. 7.9 is a representation of the binary search tree in Fig. 7.8. Each node in the multiway tree has a number of nodes of the binary search tree. In the binary tree scheme, it was necessary to use some technique like height-balancing to keep the tree balanced. The B-tree on the other hand uses a technique that keeps it always balanced but the cost of keeping it balanced is that the nodes are not required to be full to capacity. We will explain this in more detail soon. Let us first define a B-tree of order *m*.

Definition—B-tree

A B-tree of order m is a multiway search tree such that

(a) Every node except the root has at most (m – 1) keys and at least [m/2] – 1 keys.
(b) The root may have as few as one key.
(c) Every node with k keys has k+1 descendants except the leaves.
(d) All leaf nodes are on the same level.

Figure 7.10 presents a B-tree of order 5. Note that the root has only one key in it and every other node of the tree is only half full. A full capacity tree will have every node containing four keys and five pointers to the nodes at the next level.

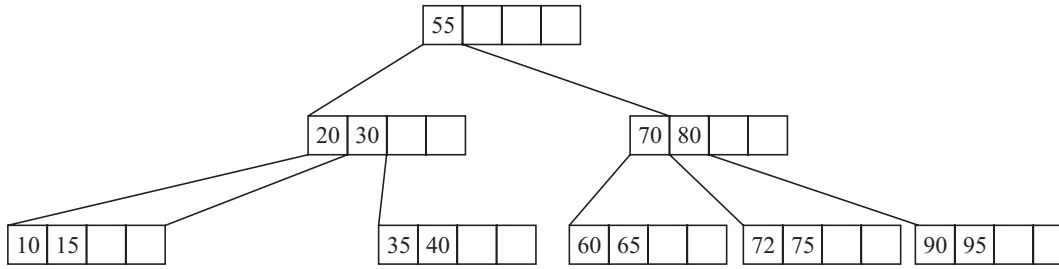

Figure 7.10 A B-tree of order 5

We now discuss how a B-tree can be searched and how insertions and deletions are made to it.

7.10.1 Searching a B-Tree

Consider searching keys from the B-tree in Fig. 7.10. Let us assume we wish to search for 95, 55, 30, 27, and 100. It does not matter in what order these searches are carried out since a search leaves the B-tree unchanged[2].

To search for 95, we start by comparing it with the key in the root. Since 95 is larger than 55, the only key in the root, the right branch is taken and the node on the right is accessed. This node has keys 70 and 80, and 95 is bigger than 80, so the rightmost branch is taken. The key 95 is located in this node and therefore the search has concluded successfully.

To search for 55, we start by comparing it with the key in the root. Since the root has the key 55, the search concludes successfully.

To search for 30, we start by comparing it with the key in the root. Since 30 is smaller than 55, the only key in the root, the left branch is taken and the node on the left is accessed. This node has keys 20 and 30 and therefore the search has concluded successfully.

To search for 27, we start by comparing it with the key in the root. Since 27 is smaller than 55, the only key in the root, the left branch is taken and the node on the left is accessed. This node has keys 20 and 30 and therefore we take the middle branch to reach the node that has 25 and 27. Therefore, the search has concluded successfully.

To search for 100, we start by comparing it with the key in the root. Since 100 is larger than 55, the only key in the root, the right branch is taken and the node on the right is accessed. This node has keys 70 and 80, and 100 is bigger than 80, so the rightmost branch is taken. This node has keys 90 and 95 and therefore the search has concluded unsuccessfully.

2. Not all data structures are left unchanged by a search. There are a number of data structures that every search will normally modify in an attempt to improve the structure's search performance. *Splay* trees are one such data structure.

7.10.2 Insertion to a B-tree

Let us consider an example of building a tree of order 5, that is, the maximum number of keys in each node is 4 and the minimum number is 2.

We build the tree by inserting records with keys 1, 3, 5, … 21 (all the odd numbers up to 21) followed by 2, 4, 6, … 20 (all the even numbers up to 20) in that order. Initially, starting from a null tree, one node is all that is needed as the first four keys are inserted in it to obtain the B-tree in Fig. 7.11[3].

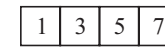

Figure 7.11 A B-tree of order 5 with only four keys and one node

The node is now full and cannot accommodate the next key. When an attempt is made to insert 9, the node *overflows* and is split into two nodes such that the middle key of the five keys migrates to a newly created root. The following algorithm is used:

1. The middle key of the *m* keys (*m* – 1 keys in the node and the key that is being inserted) is found. If *m* is even, then either of the two middle keys may be chosen.
2. This middle key partitions the keys into two groups: keys that are smaller than it and keys that are greater.
3. One of these groups is put in the present node and the other in a new node. The middle key migrates to the parent node. If there was no parent node, a new root node is created.

The above procedure is called *node splitting*. We now obtain the B-tree in Fig. 7.12 after splitting the single node shown in Fig. 7.11.

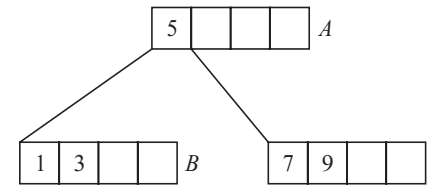

Figure 7.12 A B-tree of order 5 after a node split

Let us label the nodes *A*, *B*, *C* as in Fig. 7.12 with *A* being the root node and *B* and *C* being the two leaf nodes. To insert keys 11 and 13, we look at the root and find that keys 11 and 13 must go to node *C* (since 11 and 13 are larger than 5, the only key in the root). We insert 11 and 13 and find that node *C* is full to capacity. Inserting key 15 now causes node *C* to split since 15 must also go to node *C*. We now obtain the B-tree shown in Fig. 7.13 after splitting node *C* into nodes *C* and *D*.

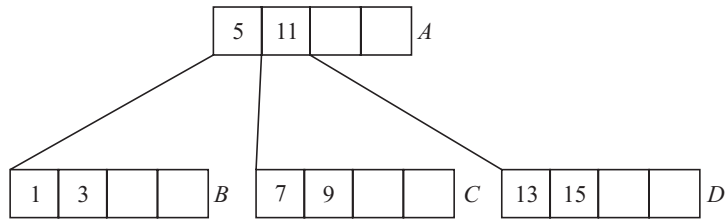

Figure 7.13 The B-tree of order 5 after node *C* is split

3. Hopefully the difference between a node and a key is clear but to ensure that it is, a node is a collection of keys while a key is often the primary key of a record although it may be another attribute or even a combination of attributes used for searching the file.

We may now insert keys 17 and 19 (in node *D*) without split, but 21 leads to splitting of node *D* sending key 17 to the root and creating a new node *E* with keys 19 and 21 in it. We now continue insertions starting with the even numbered keys and keys 2 and 4 are inserted in node *B* filling it to capacity, 6 and 8 are inserted in node *C* filling it to capacity as well, but insertion of 10 causes node *C* to overflow. A split takes place sending 8 to the root and creating a new node (node *F*). Keys 12 and 14 are now inserted in node *D* and this node is also now full. The B-tree is given in Fig. 7.14.

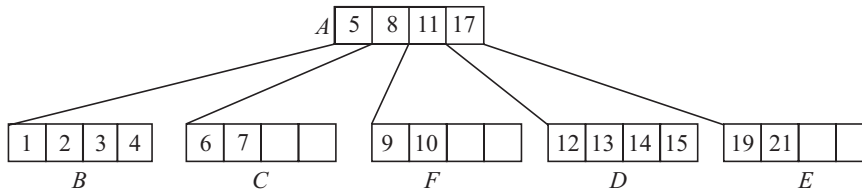

Figure 7.14 The B-tree of order 5 after more insertions

Now insertion of 16 leads to an interesting situation. It needs to be inserted in node *D* which is already full and so causes a split of node *D* sending key 14 to the root and creating a brother node *G*. Unfortunately, the root itself is full to capacity and it must also be split resulting in a new node *H* and a new root node *I* containing the middle key. The B-tree is given in Fig. 7.15.

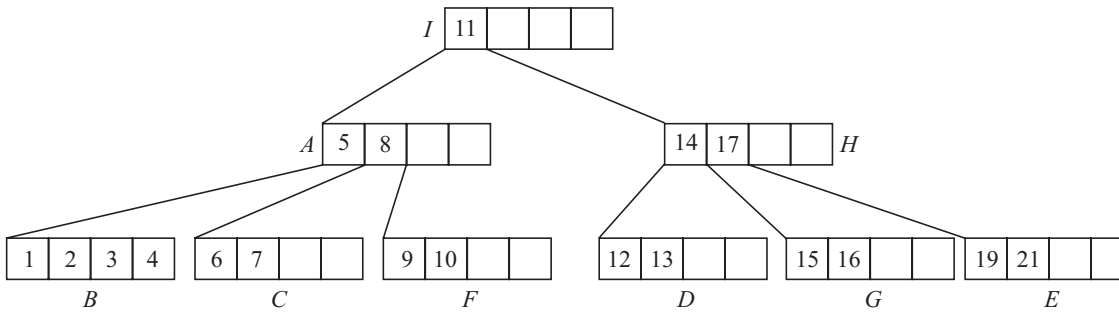

Figure 7.15 The final B-tree of order 5

We may now insert keys 18 and 20 to node *E* of the tree in Fig. 7.15 without any further node splitting and obtain a tree similar to that shown in Fig. 7.15 except that the node *E* is full with keys 18, 19, 20, 21. We note that the tree grows only when the root node is split which results in a new root node. Also we note that the final tree obtained in the above example has poor storage utilization since most nodes are only 50% full. In fact the root is only 25% full.

7.10.3 Deletion from a B-tree

We now discuss deletion of keys from the B-tree in Fig. 7.15. Node *E* is full and has keys 18, 19, 20 and 21. First the keys 4, 7, 9, 10 and 8 are deleted in that order.

To delete 4, it is searched by comparing it with the key in the root (node *I*). Since 4 is smaller than 11, the only key in the root, the left branch is taken and node *A* is accessed. Node *A* has keys 5 and 8, and since 4 is

smaller than 5, the leftmost branch is taken to reach node *B*. The key 4 is located in this node and is deleted. The node is still more than 50% full and therefore no further action is needed.

To delete 7, we search for it through the root *I*, then node *A* and locate it in node *C* which is only half full and delete key 7 from it. Node *C* now violates the B-tree definition by being less than half full, it has *underflowed*, that is, it has less than the minimum number of keys allowed for a node in a tree of the order 5. To overcome this problem we look at the brother nodes of node *C* (nodes *B* and *F*) and either coalesce with one of them if that brother node is also close to underflowing or borrow some keys from one of them. To coalesce requires the keys from both the nodes that have underflowed and the brother node as well as the one key that is the separator key in the parent node to be put in one node. Therefore, the total number of these keys must be less than or equal to *m*–1, *m* being the order of the tree. If the total number of keys is greater than *m*–1, then coalescing is not possible and it is necessary to redistribute the total keys from the two nodes amongst them, such that both nodes have an almost equal number of keys, and the separator key from the parent node is included in the redistribution with the middle key again going to the parent node.

In this particular case, it is decided to coalesce the two nodes *C* and *F* and remove the separator key (key value 8) from node *A* and include it in the newly combined node *C*. Node *F* has been removed. This results in node *A* underflowing since it is left with only one key. The B-tree at this stage appears as given in Fig. 7.16.

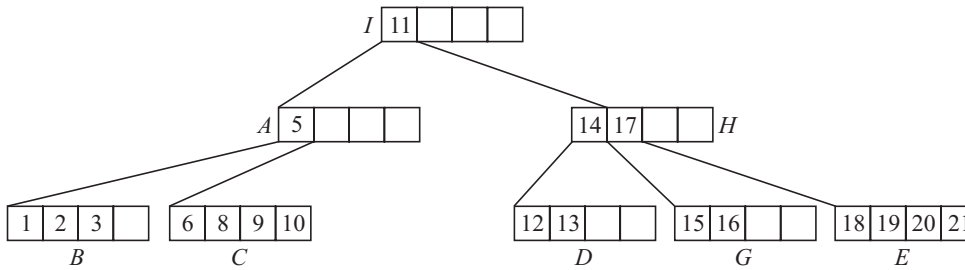

Figure 7.16 The B-tree after a number of deletions (Node *A* has underflown)

Because node *A* has now underflowed, we must consider redistributing keys between it and a brother node (that is, node *H*) or coalesce it with that node. In this case we have no option since node *H* is also only 50% full and redistribution is not possible. Therefore, we must coalesce nodes *A* and *H*. In combining *A* and *H*, we need to include the separator key from the parent node (the root) but the root node (node *I*) has only one key and therefore merging nodes *A* and *H* results in the root node as well as node *H* disappearing as shown in the B-tree in Fig. 7.17.

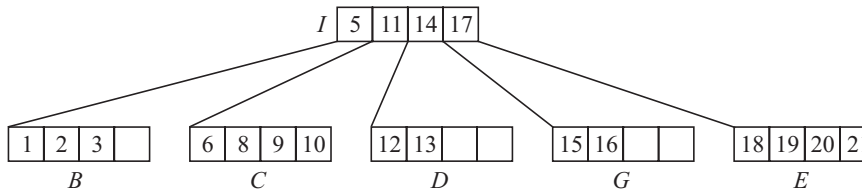

Figure 7.17 The B-tree after further deletions

Next the key 9 is deleted from node *C* and then key 10 from *C* reducing node *C* to only two keys. Deletion of 8 from node *C* causes node *C* to underflow. We look at the brother node (node *B*) and see if the two nodes can be merged together. Node *B* has three keys and cannot be coalesced with the underflowed node *C* which has only one key since the coalesced node must also include the separator key from the node above. We therefore redistribute the four keys in the two nodes and the separator key from the parent, which is key 5. Now the new middle key is sent to the parent, this is key 3, and we obtain the B-tree in Fig. 7.18.

Redistribution of keys in nodes *B* and *C* involved moving only one key (key 3) from node *B* to the parent node (root in this case) and the separator key (key 5) from the root to node *C* since we are dealing with a B-tree of order 5. In a tree of higher order, enough keys will be moved from the brother node to the node that has underflowed to make the number of keys in each node almost equal.

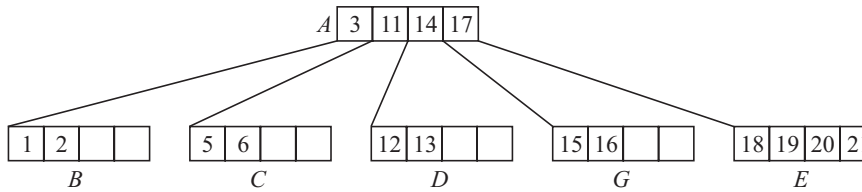

Figure 7.18 The B-tree after deleting keys 9, 10 and 8

In general, if a deletion at level k leads to a combination of two nodes, it may also lead to underflow of a parent node resulting in two nodes at level $k-1$ coalescing or redistributing keys. This in extreme cases may lead to underflows at levels $k-1$, $k-2$ and so on, and eventually reduce the tree height by one.

7.10.4 Analysis of B-tree Performance[4]

We now consider the cost of insertion, deletion and retrieval in a B-tree. These costs are directly related to the height of the tree. We first consider the worst case height of a B-tree with n nodes.

The Worst Case

Let the top level of the tree (i.e., level of the root) be 0 and let the level of any other node be one higher than the level of its parent. We wish to compute the largest value of height h that a B-tree with n keys might have.

To simplify the analysis below, we will assume the order m to be odd and further assume $m = 2d + 1$.

The B-tree of height h with the worst storage utilization would have only one key in the root node and all other nodes would be only 50% full. That is, if the order of the tree is $2d + 1$, then the number of *nodes* (not keys) in the worst B-tree of height h would be given by the equation on the next page.

Note that the root is one node (with only one key) and it has two children nodes. Since we are considering the worst case, these nodes are only half full (i.e., they only have d keys) and each has exactly $(d + 1)$ children nodes [note that a node with d keys must have $(d + 1)$ children]. Since each of these nodes is also only half full, each will also have $(d + 1)$ children nodes and so on. Therefore,

4. This section may be left out without any loss of continuity.

Table 7.10 The number of nodes in a B-tree that is half full

Level	*Number of Nodes*	*Number of Keys*	*Comment*
0	1	1	Root is allowed only one key
1	2	$2d$	Each node is only half full
2	$2(d+1)$	$2d(d+1)$	Each node has $d+1$ children
3	$2(d+1)^2$	$2d(d+1)^2$	Each node is only half full
4 etc	and so on		

Given these, we may now compute the total number of nodes in the B-tree of height h as follows. Note that a height h tree will have h levels 0, 1, 2, … $(h-1)$.

$$N_{\min} = 1 + 2[1 + (d+1) + (d+1)^2 + \text{K} + (d+1)^{h-1}]$$
$$= 1 + 2\frac{(d+1)^h - 1}{(d+1) - 1}$$

The number of keys in $N_{\min}$ nodes is $1 + (N_{\min} - 1)(d)$ since in the worst case the root has only one key. If a B-tree of height h has n keys, we may write

$$1 + (N_{\min} - 1)(d) < n$$

Substituting for $N_{\min}$ and cancelling out d from numerator and denominator, we obtain

$$1 + 2[(d+1)^h - 1] < n$$

$$\boxed{h < \log_{(d+1)} \frac{n+1}{2}}$$

This is the worst height a B-tree of order m (or $2d+1$) with n keys could have. We now look at the best case.

The Best Case

A B-tree with the best storage utilization would have all the nodes full. This means that the number of nodes (*not* keys) in a tree of height h would be the root (which is assumed full) that has m children that are also all full, each in turn has m children and so on. Note that in the following, we do not need to use the notation $m = 2d + 1$ since it is easier to use m as the order of the tree.

Table 7.11 The number of nodes in a B-tree that is half full

Level	*Number of Nodes*	*Number of Keys*	*Comment*
0	1	$(m-1)$	The root is full
1	m	$m(m-1)$	Every node is full ($m-1$ keys)
2	m^2	$m^2(m-1)$	Every node has m children
3	m^3	$m^3(m-1)$	Every node has m children
4 etc.	and so on		

Given these, we may now compute the total number of nodes in the best case B-tree of height h as follows:

$$N_{\max} = 1 + m + m^2 + \mathrm{K} + m^{h-1}$$

$$= \frac{1}{m-1}(m^h - 1)$$

Let the number of keys in the above tree of height h be n. Since each node is full and has $(m - 1)$ keys, we must then have

$$N_{\max}\,(m-1) \geq n$$

Substituting for $N_{\max}$ and cancelling out $(m - 1)$, we obtain

$$(m^h - 1) \geq n \text{ or}$$

$$\boxed{h \geq \log_m (n+1)}$$

The above expression provides a lower bound for h for a B-tree of order m with n keys. Therefore, the height of a B-tree of order $m\,(m = 2d + 1)$ with n keys must be such that

$$\boxed{\log_{(2d+1)} (n+1) \leq h \leq \log_{(d+1)} \frac{n+1}{2}}$$

Example

Let us consider a tree of order 101 (this order is not too large since a node is often equal to a block and a block is often 4 KB). Table 7.12 lists the minimum and the maximum number of keys that may be stored in such a tree of height h.

Table 7.12 Minimum and maximum number of keys in a B-tree of order 101

Height (h)	*Minimum Number of Keys*	*Maximum Number of Keys*
0	1	100
1	101	102,000
2	5201	1,030,300
etc.		

It is clear from the above figures that only 3 or 4 accesses of disk will be necessary for even the largest file if the order of the tree is close to 100. Clearly if the storage utilization in the B-tree is high, the height of the B-tree for the same number of keys will be smaller and the cost of insertion, deletion and retrieval will be lower. If the storage utilization is low, the height of the tree would be larger and the cost of insertion, deletion and retrieval will be higher. We are therefore interested in knowing the average storage utilization of B-trees that are built randomly. The derivation of average storage utilization is beyond the scope of this book. We only present the final result. The storage utilization of randomly built B-trees is $\log_e 2 = 69\%$ on average.

The B-tree therefore is an efficient data structure for storage of information on disk. The structure is always balanced and is efficient for insertion, deletion and retrieval but it should be noted that the B-tree is not very efficient for sequential processing. For example, if we want to traverse the keys in the B-tree in Fig. 7.15 we would need to start from the root, then visit A then B, back to A then C, back to A then to F and

so on. In sequential processing, the B-tree structure that we have discussed is quite different from the indexed-sequential file structure.

The indexed-sequential structure is very convenient when the file needs to be accessed sequentially. As the above example shows, it is possible to use the B-tree structure for sequential processing but it is not as efficient as the indexed-sequential file. B-tree sequential traversing requires first finding the leftmost leaf and then processing the key that separates it from the next leaf (this separator key is in the parent node). The next leaf may now be processed followed by the next separator key in the parent node and so on. Any modification of the B-tree structure that permits more efficient sequential processing is clearly desirable. We now discuss one such modification.

7.11 THE B$^+$-TREE

This B-tree modification, called the B$^+$-tree is based on the observation that when a B-tree is searched for a key, the chances of finding the key before reaching the leaf are quite low if the order of the tree is high (say, 100). For $m = 101$, level $i + 1$ would contain at least 51 times the number of nodes at level i since each node must have at least 50 keys and 51 children. Therefore, the number of nonleaf nodes is not expected to be more than 1/50th (or 2%) of the number of leaf nodes. This suggests a simple modification of the B-tree scheme that has some useful properties in addition to the B-tree properties.

The modification involves duplicating every key that would normally appear in a nonleaf node in the appropriate leaf node. As noted earlier, this should not require much additional storage if the order is high (say, higher than 10).

In addition to putting all the keys in the leaf nodes, each leaf node is now required to have a pointer to the next leaf node in sequence. The structure given in Fig. 7.14 therefore looks like the Fig. 7.19. Note that each key in the root node in Fig. 7.14 has been duplicated in the appropriate leaf node so that all the keys are now resident in the leaves and form a sequential file.

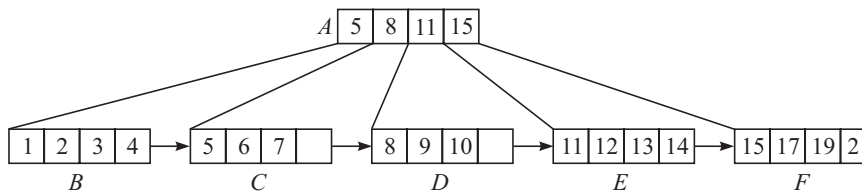

Figure 7.19 A B-tree modified for sequential processing

The modified B-tree structure is called the B$^+$-tree and it provides another very useful property. If we look at the B$^+$-tree without the leaf nodes (for a tree that has more than two levels, for example, the tree in Fig. 7.20), this is really an index for the sequential file which is the leaf nodes. The sequential file is stored in the leaf nodes (often the leaf nodes will have pointers to the records from the keys in them although the pointers are not shown in the figure) by linking the leaf nodes through pointers from one leaf node to the next. The B$^+$-tree structure may therefore be considered an indexed-sequential file.

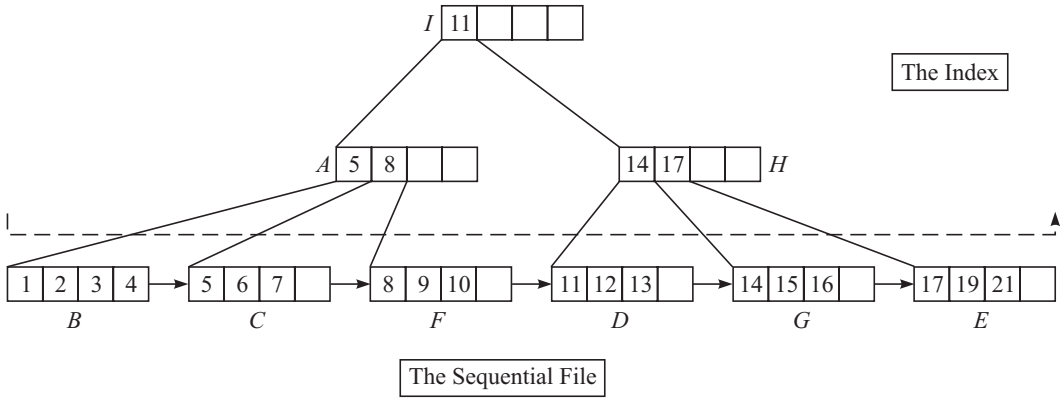

Figure 7.20 A B^+-tree of order 5 as an indexed sequential file

Since the internal nodes of a B^+-tree are the index and the leaves are the sequential file, more entries may be packed into the internal nodes by, for example, using only the first four letters of the keys in the internal nodes. As noted earlier, a higher order B^+-tree structure means that the index is more compact and faster to search, since the height of the index could well be smaller by packing the keys in the internal nodes. Also, by linking the leaf nodes, the B^+-tree structure makes it easy to traverse the sequential file.

A number of other modifications of the B-tree have been proposed to improve the efficiency of the basic tree. A commonly used modification is to change the condition that the nodes must be at least half full to the condition that they be at least two-third full. This improves the storage utilization and for large indexes reduces the index tree depth, thus making it more efficient.

The B^+-tree indexed-sequential file has a number of advantages over the conventional indexed-sequential file. These advantages and disadvantages are listed in the following section.

7.12 ADVANTAGES AND DISADVANTAGES OF B-TREES AND B^+-TREES

B-trees have substantial advantages over other file structures. The main advantages are as follows:

1. B-trees are efficient external storage structures. If the node access time far exceeds search time within nodes, the B-tree structure is usually efficient.
2. B-trees are good for searches in dynamic files since they are always balanced.
3. The only disadvantage of the B-tree and B^+-tree structure is that storage utilization tends to be close to 70% but this can be improved if the modification described above is used.
4. Developing insertion and deletion algorithms can be somewhat complicated in B-trees and B^+-trees.
5. As compared to indexed-sequential files, the performance of B^+-trees does not degrade as the file grows, since the file grows and shrinks gracefully.
6. As compared to indexed-sequential files, the B^+-tree file requires no overflow area and no periodic reorganization to maintain good performance.

We now discuss another data structure that is also commonly used in databases. It is called hashing. Static hashing followed by dynamic hashing is discussed in the sections ahead.

7.13 STATIC HASHING

We have so far studied sequential and indexed-sequential files. Indexed-sequential files are quite efficient for inserting, deleting and accessing a particular record from a file as well as for processing the whole file sequentially although for very large files the index becomes large impacting the efficiency of indexed-sequential files. Another major file organization technique that is often better for retrieval is called *hashing*.

7.13.1 Basics of Hashing

Hashed files can often lead to better performance in insertion, deletion and retrieval but the performance of sequential processing of the file is very poor. For example, a file *Player* that is stored using hashing on the last name *LName* can very quickly find the record for player with *LName = Sehwag*, much faster than even an indexed-sequential file would if the file is large.

File organizations based on hashing were developed as early as the mid 1950s. It has been reported that the idea of hashing originated at IBM in 1953. Significant developments have taken place since then. We will discuss classical hashing techniques briefly and then some more recent developments in dynamic hashing.

Dynamic hashing is used extensively for file organizations on external storage but hashing is equally applicable for organizing tables in the main memory. For example, static hashing is often a suitable technique for organizing a symbol table in a compiler.

The idea behind hashing techniques is to use the key of a record or another important attribute of the file and calculate the storage location from it. The transformation from the key to the storage location is done using a formula that is called a *hash function*.

Let k be the key of a record (or value of another attribute which is frequently used in search[5]) and let $h(.)$ be the hash function being used. $h(k)$ is then the address of the record whose key is k. The hash function can be as simple as dividing by a prime number N and using the remainder for numeric keys. Clearly it is necessary that the function $h(.)$ demonstrate some useful properties. Some of these are as follows:

1. The addresses generated by the function are within a given interval which specifies the acceptable addresses. For a table of size N, or file of size N, the addresses are 0, 1, 2, ... $N - 1$.
2. The hashed address often refers to a *bucket* which may be able to store a number of records. The size of the bucket may be equal to a block when the hashed file is stored on the disk.
3. The hash function should generate addresses that are approximately uniformly distributed over the range of acceptable addresses and should break-up naturally occurring clusters of key values.

5. We will keep using the term key for the attribute on which the file is hashed. This attribute does not need to be the primary key of the table that is being stored using hashing.

4. The hash function transformation should use all the characters or digits in the key to compute the address.
5. The hash function should be easy and fast to compute.

We will discuss several hash functions that satisfy the above requirements.

We first define some terms that are used in this context. We assume that the records or rows of a table are being stored in a table of size N (as noted earlier, each entry in the table could be an address of a block or bucket on secondary storage, a bucket being a collection of one or more blocks of storage connected by pointers). Therefore, only addresses from 0 to $N - 1$ are acceptable.

Figure 7.21 shows a very simple hashed file with a table of only 4 addresses that are pointing to buckets that store a number of records each. The hash table entry 2 has currently no records hashed to it.

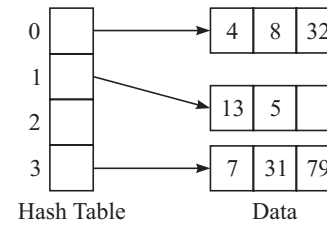

Figure 7.21 A simple static hashing file

The address generated by the hash function for a record is called the record's *home address* and the block to which the record is hashed is sometimes called the *primary block* for the record. If a record is hashed to an address where there is no space for it, we call it a *collision*. Records having the same home address are called *synonyms*.

Search

We now briefly discuss how to search the key for a record given. Let the key be K.

1. The key K is first hashed using the hash function used in building the hash table. Let the address be A.
2. The hashed address A refers to a *bucket* which stores a number of records. Search the bucket at address A for the record with key K.
3. The search concludes successfully if the key K is found otherwise unsuccessfully.

Of course, it is not as simple as the above algorithm shows because we often need to deal with collisions. If the hash table has collisions then the search algorithm becomes more complex depending on the collision technique used in building the hash table.

Collision

When a collision takes place, some technique of collision resolution is needed. We will consider the collision resolution techniques listed in Fig. 7.22.

1. Linear probing
2. Double hashing
3. Chaining

Figure 7.22 Collision resolution techniques

These collision resolution techniques will be discussed after we describe some hashing functions.

7.13.2 Hash Functions

As explained earlier, a hash function $h(.)$ transforms the attribute value k such that $h(k)$ has a value between 0 and $N - 1$ if N is the size of the hash table. Often N is a power of 2 and the function $h(k)$ produces a bit string of length b such that $N = 2^b$ and therefore the addresses generated are from 0 to $2^b - 1$.

As noted earlier, we desire that the hash function generate addresses that are uniformly distributed over the table even if the key values are not uniformly distributed. If some addresses are generated more frequently than others, we call the phenomenon *primary clustering*.

We now discuss some hash functions that have been found to perform satisfactorily for many different types of data. A reader familiar with pseudorandom number generators would have noticed the similarities between the requirements of hashing functions and pseudorandom number generators. This similarity enables us to use the techniques that have been developed for pseudorandom number generation in designing hash functions.

Most hash functions assume that a binary representation of the key (the key could well be alphanumeric or it may be another attribute of interest) is available for transformation. We therefore assume that a key k may be interpreted as an integer that is represented in binary form. Let this integer be z.

We now discuss three commonly used hash functions, viz., integer divide, mid-square and compression.

Integer Divide

The simplest hash function involves dividing z (the integer representation of the key k) by the size of the table N and using the remainder as the address. Therefore, $h(k) = z \bmod N$.

The above hash function is simple and fast to compute. The computation can be made even faster if N was 2^b for some suitable b. $h(k)$ is then given by the least significant b bits of the binary representation of z and the distribution of the computed addresses would be dependent on the distribution of these bits. It can be shown that when an integer divide is used, choosing $N = 2^b$ often does not provide a good distribution of computed addresses although the approach is very efficient. The reasons for this behaviour is not hard to find. Let the key of records be names having three components (first name, middle initial, last name). If the b least significant bits were used as the hashed address, all records with the same last name could well be mapped to the same location and the hash function may ignore the first name and the middle initial in the transformation. This is not desirable since some last names (for example, Singh) might be very common. That is why one of our requirements is that the hash function must use the entire key (not just a part of it) in computing the address.

It can be shown that by choosing N as a prime number, the problem described above usually disappears and the distribution is often uniform.

Mid-Square Function

Another quite commonly used hash function is based on squaring z (the integer representation of k) and then selecting the middle b bits as the home address of the record in a table of size 2^b.

The function satisfies the requirement that the mapping be based on all characters in the key and it has been observed that the middle b bits are usually distributed uniformly even if the keys are not.

Compression Function

This function involves partitioning the binary representation of k (that is, z) into several sections that are b bits long. These sections are then added together and the least significant b bits are taken as the address in a table of size 2^b.

Example

Consider a simple example to illustrate the use of the above hash functions. Let the key value be only two characters long. We only use uppercase letters and transform the keys to numeric values by replacing *A* by 1, *B* by 2, ... *Z* by 26. We will then apply the three hash functions to the numeric representation of the key.

Our table size is assumed to be 10 for hash functions mid-square and compression and 11 for the integer divide function, since integer divide requires table size to be a prime number for good performance.

Table 7.13 A simple example of three hash functions

Key (*k*)	*Numeric Representation* (*z*)	*Integer Divide*	z^2	*Mid-digit of* z^2	*Compression Function*
AB	0102	3	0010404	0	3
AZ	0126	5	0015876	5	0
BN	0214	5	0045796	5	7
NB	1402	5	1965604	5	7
CD	0304	7	0092416	2	7
EF	0506	0	0256036	6	1
GH	0708	4	0501264	1	5
HK	0811	8	0657271	7	0

The integer divide hash function was *z* mod 11. The mid-square function involved squaring *z* and extracting only one middle digit since the table is of size 10. We chose the 4^{th} digit from the right (it being one of the two middle digits). The compression function also needs to transform *z* into one digit since the table size is 10. We achieve the transformation by adding the digits of *z* and then taking the least significant digit.

The above example had a hash table of size 10 for mid-square and compression functions, and size 11 for integer divide. The number of keys that were hashed to the table was 8. Although the tables were not quite full, we found that there were several collisions whatever hash function was used. We now discuss techniques for collision resolution.

7.13.3 Collision Resolution

It should be noted that collisions are an unavoidable consequence of the use of a hash function. We are after all trying to map a very large range of key values onto a much smaller hash table range. Unless we know beforehand the key values of the records to be inserted (and it is very rarely that we do), we cannot hope to devise a function which maps each key value into a unique position in the table.

Since collisions cannot be avoided, the best we can hope for is to reduce the number of collisions and spread the collisions uniformly across the table. We have already discussed hash functions that usually perform well in distributing records over the table. In most cases, no prior knowledge is available about the distribution of key values and then one of the commonly used hash functions may be used.

Before we discuss collision resolution techniques we need to consider the definition and discuss a minor problem that has to be resolved.

We define the term *load factor* as the ratio of the records in the table to its capacity. Load factor α is thus given by S/N where S is the number of records in the table which has a capacity of N records.

When hashing is used, some locations in the table have records in them while other locations are vacant. When inserting an element into the table or searching for an element, it becomes necessary to check whether a location in the table is vacant. Some simple mechanism is therefore needed to initialize a table before records can be inserted in it. If the key value is never expected to be zero then it may be used to indicate a vacant location. If this is not possible, some other value or a flag bit may have to be used. In addition, some techniques require that an empty location be differentiated from a location that has become empty due to deletion. It may be possible to represent a deleted entry by –1.

We now discuss collision resolution techniques.

Linear Probing

When a collision takes place on insertion of a new key, the linear probing method involves storing the new key k in the address $h(k) + 1$ if the home address of the key is $h(k)$. Should the address $h(k) + 1$ also be full, we continue looking at the next address till an address is found that can store the record. Of course, if we reach the end of the table, we wrap around to the top of the table. If all the addresses are full, we declare the table to be full and no insertions can be made.

Consider Table 7.14 of size 11:

Our hash function is k mod 11. If we now try to insert 238, 195 and 75 in that sequence, we find that 238 is hashed to location 7 and results in a collision. Since location 8 is also occupied, we insert 238 in location 9. The element 195 hashes to location 8 and is inserted in location 10 since that is the first vacant location. The element 75 is hashed to 9 and is inserted in location 0 producing Table 7.15.

Table 7.14 An example of a hash table using linear probing

0	
1	716
2	
3	25
4	509
5	
6	
7	40
8	30
9	
10	

Table 7.15 An example of the hash table after collisions using linear probing

0	75
1	716
2	
3	25
4	509
5	
6	
7	40
8	30
9	238
10	195

If we now want to insert 51, which hashes to location 7, it would have to be inserted in location 2 since all locations before 2 are occupied.

Searching for 51 would involve first searching if 51 already exists in the table. It therefore involves looking at location 7 and then going down the table till 51 is found or an empty location is found. The empty location could also have been created by a deletion after a previous insertion of 51 and therefore a vacant location and a deleted entry need to be differentiated. The search for an element in the table can terminate only at a vacant location or after the whole table has been searched.

Deleted elements create difficulties not only in searching but also in insertion. To illustrate the difficulty, let us assume that 238 has been deleted from the table and has been replaced by –1 to indicate a deleted entry. If we now wish to insert 51 into the table, we hash it to the table (location 7). Since location 7 is occupied, we look at location 8 which is also occupied. Location 9 however is free, having a deleted flag, and so we insert 51 in location 9. If this algorithm was followed, we could have a table with two key values 51 which probably was not what was intended! The correct insertion procedure would have involved a search for 51 and if unsuccessful, following it by insertion.

We now briefly discuss how the average cost of successful and unsuccessful search is computed. Consider the cost of searching each key in the last table. It is the average number of comparisons needed to find a key. For example, searching for 75 needs 3 comparisons; 716 needs 1 comparison; and 25 needs 1 comparison. Similarly searching for 509, 40, 30, 238 and 139 require 2, 1, 1, 3 and 4 comparisons respectively. The average cost of successful search therefore is

$$s = \frac{3+1+1+2+1+1+3+4}{8} = 2$$

Similarly, we can find the average cost of unsuccessful search in the above table. The unsuccessful search could start at any of the 11 locations with equal probability. The home address of the key being searched unsuccessfully and the cost of unsuccessful search starting at that address are presented in Fig. 7.23.

	Cost of each table address
1	search starting at address 0 would cost 3 comparisons.
2	search starting at address 1, costs 2 comparisons
3	search starting at address 2, costs 1 comparison
4	search starting at address 3, costs 3 comparisons
5	search starting at address 4, costs 2 comparisons
6	search starting at address 5, costs 1 comparison
7	search starting at address 6, costs 1 comparison
8	search starting at address 7, costs 7 comparisons
9	search starting at address 8, costs 6 comparisons
10	search starting at address 9, costs 5 comparisons
11	search starting at address 10, costs 4 comparisons

Figure 7.23 The cost of searching each hash table entry

The average cost of unsuccessful search therefore is given by

$$u = \frac{3+2+1+3+2+1+1+7+6+5+4}{11} = \frac{35}{11} \approx 3.2$$

It should now be clear that linear probing will perform well as long as the load factor is not high. Let the load factor be α, then the average number of comparisons needed for successful search and an unsuccessful search using linear probing for collision resolution are given by the expressions in Fig. 7.24.

$$s \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$
$$u \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Figure 7.24 Average number of comparisons needed for successful search and unsuccessful search using linear probing

For derivations of the above results, the reader is referred to the book by Knuth (1973).

Double Hashing

The major problem with linear probing is that collisions upset adjacent locations which in turn upset further adjacent locations when insertions occur. This results in long scans when a search is initiated and is called *secondary clustering*.

To minimize this clustering effect, double hashing handles collision by stepping along the table using variable step sizes based on the key value rather than with a unit increment as in linear probing.

For example, for a table of size 11 using an integer divide hash function, 18, 29, 40 and 51; all hash to the location 7. If a second hash function z mod 7 was used to determine the step length, we would obtain values of step length to be 4, 1, 5 and 2 for the key values 18, 29, 40 and 51 respectively. This reduces secondary clustering since it splits the path that keys hashed to the same location follow. A hash function like z mod 7 is however not satisfactory since it would map some keys to 0 and a step length of 0 obviously would create problems. Therefore, we must insist that the second hash function satisfy $1 \leq h(z) \leq N$.

Table 7.16 An example of the hash table after collisions using double hashing

0	232
1	716
2	236
3	25
4	509
5	242
6	244
7	40
8	30
9	250
10	252

Let us now consider Table 7.16 to illustrate double hashing.

We now wish to insert 57 and 238 both hashing to location 7. We therefore need to use a second hash function to find the step length. Let the step function be (1 + z mod 7). This maps 51 to 3 and 238 to 1. 51 can therefore be inserted in location 7+3 (if this location was occupied, we would have tried 7+3+3 (mod 11). 238 need to be inserted in location 7+1 but location 8 is also occupied. We try 8+1 and are successful in placing 238 in that location. If we now wish to insert 195 and 75, we compute their home addresses to be 8 and 9 respectively. Both these addresses are occupied. We therefore use the second hash function to find the step length. The hash function (1+ z mod 7) maps 195 and 75 to 7 and 6 respectively. We now try location 8+7 (mod 11) for 195. This location (4) is occupied and we therefore try 4+7 and insert the element there (location 0). For 75, we

need to try location 9+6 (mod 11) but this is occupied. We now try 9+6+6 (mod 11) and find that this location (5) is available and insert 75 there. It should be noted that if N is prime, and there is a vacant space in the table, the double hashing technique will automatically find it.

The average cost of a successful search and an unsuccessful search using double hashing are given by the expressions in Fig. 7.25.

$$s \approx \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

$$u \approx \frac{1}{1-\alpha}$$

Figure 7.25 Average number of comparisons needed for successful search and unsuccessful search using double hashing

Once again, for details of derivations of the above formulas, the reader should consult Knuth (1973).

The above two techniques resolve collisions by finding a free space within the hash table (this type of techniques are called *open addressing*). This leads to more and more probing as the table fills. If the elements on collision are put not in the table itself but somewhere else, then the behaviour of the hash table should improve markedly (as compared to when open addressing is used) as the table becomes full.

Chaining

The chaining scheme involves making each entry in the hash table either null or a pointer to a chain of synonyms. For example, a hash table of size 10 is shown in Fig. 7.26. This figure shows the hash table with six null entries and four entries that are pointers to chains. Three of the chains include only one entry each, while one entry has two items in the chain.

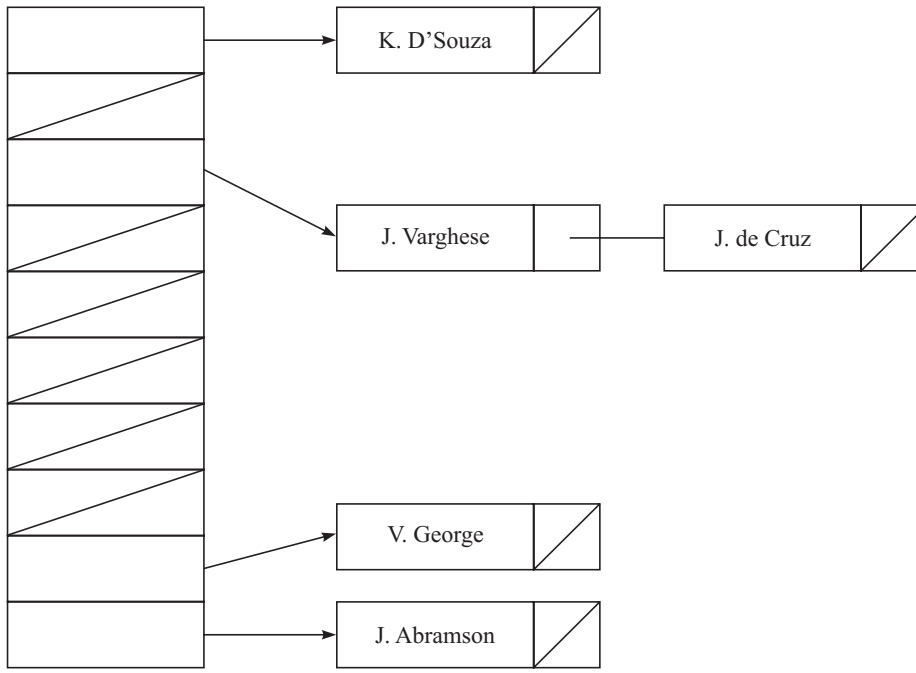

Figure 7.26 An example of open chaining to deal with collisions

Chaining also allows much easier deletion of records than open addressing since the record to be deleted must occur in the chain originating at the home address of the key value.

There are several different ways chaining may be implemented. The most common approach is to have a table, each element of which is a list header (possibly null). When elements are mapped to the table, an element is attached to the chain being pointed to by the header and the pointers are updated accordingly. We are assuming that the table and records are main memory resident.

A successful search then involves hashing the key to the table and then searching the chain that starts at that location. The cost of a successful search is the number of comparisons needed to find the element; the cost of accessing the header pointer is not included in the cost.

An unsuccessful search involves mapping the key to the table and searching the chain till the end of the chain. Of course, there may be no chain at all starting at the home address of the key being searched and then the search concludes immediately. The cost of an unsuccessful search is the number of comparisons needed to reach the end of the chain. In the case of a null chain, the cost is assumed to be zero although some authors assume the cost to be one.

The costs are then given by the expressions in Fig. 7.27.

In chaining, the load factor could become larger than 1 and the above formulas still apply. This is another advantage of chaining. The file does not overflow when the number of elements is one greater than the table capacity.

$$s \approx 1 + \frac{1}{2}\alpha$$
$$u \approx \alpha$$

Figure 7.27 Average number of comparisons needed for successful search and unsuccessful search using chaining

The primary disadvantage of chaining is that additional storage is needed for pointers. But a higher load factor could be used with chaining without much degradation in performance. Also deletion and an unsuccessful search are much faster when chaining is used.

Comparing Performance

We have presented formulas that give the approximate cost of successful and unsuccessful searches for each of the three collision resolution techniques that we have discussed. We now compare these costs for several different values of the load in Tables 7.17 and 7.18.

Table 7.17 Comparing performance of the three methods for a successful search

Load Factor	0.4	0.6	0.8	0.9
Linear Probe	1.3	1.7	3.0	5.5
Double Hashing	1.2	1.5	2.0	2.5
Chaining	1.2	1.2	1.3	1.4

Table 7.18 Comparing performance of the three methods for an unsuccessful search

Load Factor	0.4	0.6	0.8	0.9
Linear Probe	1.8	3.5	9.0	50.0
Double Hashing	1.6	2.5	4.0	10.0
Chaining	1.1	1.2	1.3	1.4

We have assumed that the table size is large, well above 50. The comparison in Tables 7.17 and 7.18 clearly shows that linear probe and double hashing are not very effective in dealing with collisions.

7.14 EXTERNAL HASHING—FILES ON DISK

As we have noted earlier in the chapter, when information is stored on a disk, a block of information is transferred to the primary memory (and from memory) at any one time. The information on disk therefore is organized in blocks; each block usually consisting of a number of records.

Hashing techniques discussed so far have assumed that the hash table is in main memory although the techniques are applicable for files residing on a disk. Rather than having a table in which each entry has a capacity of one record, we could view a file on a disk as a table in which each entry in the table is a block (sometimes called a bucket, as noted earlier) on the disk. In fact, we normally would have a table in the main memory that would at each location in the table store the disk address of the primary bucket corresponding to the home address. Let the capacity of each bucket be b so that the capacity of the file is Nb if N buckets are allocated to the file.

We now have a situation where each home address can accommodate several records and therefore the insertion and deletion algorithms are somewhat different. When inserting a record, we compute the home address and then read the bucket corresponding to that address, and insert the record in the bucket if there is a vacancy. If the bucket is full, we must insert the record elsewhere.

When discussing a hashed file on secondary storage, the term *overflow* is often used. An overflow is said to occur when a record is mapped to a bucket that is full. When an overflow occurs, techniques similar to those for collision resolution discussed earlier can be used to find a suitable location for the new record.

It can be shown that if there were two tables of the same size (that is, same Nb) one with bucket size of 1 and another with bucket capacity of b, the performance of the larger bucket file is much better at the same load factor.

We now discuss two techniques for dynamic hashing, viz., extendible hashing and linear hashing.

7.15 DYNAMIC HASHING FOR EXTERNAL FILES

The static hashing techniques discussed in Section 7.8 work satisfactorily if the table is not close to being full and the table size is not changing. A major problem with these traditional hashing schemes is that the storage must be statically allocated based on an estimate of the size of the file. The difficulty with this approach is that very often performance deteriorates if the file grows, and then the size of the file is not close to the estimate used initially to allocate storage. If the storage requirements are underestimated, the number of collisions will be large. On the other hand, if the storage requirements are overestimated, storage utilization will be low and storage will be wasted. Many files are dynamic in nature, that is, they grow and shrink dynamically by several orders of magnitude during their lifetime. For these files, reorganization of the whole file is required at regular intervals in order to maintain good performance if static hashing is used.

This major difficulty with static hashed files can be overcome by using one of the *dynamic hashing* schemes. With these schemes, the size of the hash file grows (and shrinks) dynamically as records are inserted and deleted from the file. No periodic reorganization of the file is required. The dynamic schemes maintain good performance independent of the size of the file.

It is possible to categorize dynamic hashing methods into two classes. The first class consists of those methods which use a directory structure; the second class of methods that do not use a directory. The best known methods based on directory structures are variations of a scheme called *extendible hashing*. The most well known methods that do not use a directory are based on a dynamic hashing scheme called *linear hashing*.

We note that it is possible to design a sequence of hashing functions h_i such that

$$h_i(k) = h_{i-1}(k), \text{ or}$$
$$h_i(k) = h_{i-1}(k) + 2^{i-1}$$

And both events are equi-probable.

The functions h_i, for example, could be generated if some function $H(k)$ was first used to generate a binary string and then h_i was the last $i + 1$ bits of that string.

A bucket that has records assigned to it using a hashing function h_i will be said to have been *split* if the records in it are redistributed in the file using the next hashing function $h_i + 1$. Records that are placed in more than one bucket (usually two buckets) using a hashing function h_i are said to *group* if they are collected and placed in one bucket using the hashing function h_{i-1}.

We now describe extendible hashing and then linear hashing.

7.15.1 Extendible Hashing

The data structure in extendible hashing consists of two components: a set of *buckets*[6] and a *directory*. The buckets reside on secondary storage and contain the data records. The directory is an array of pointers to the buckets. If the directory is not too large, it should reside in the main memory. An example is given in Fig. 7.28.

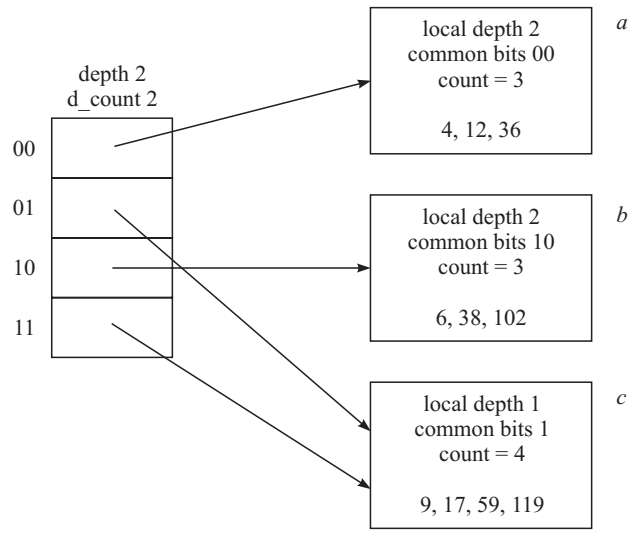

Figure 7.28 A simple example of extendible hashing

6. The term buckets is used in hashing for a block or a page on disk due to historical reasons.

Here, the directory consists of only four entries. At any time, the size of the directory will be of the form 2^d where d is called the *depth* of the directory.

In order to determine the location of a record with key k, a hash function is used to generate a very long bit pattern or *pseudokey* when it is applied to k. The rightmost d bits of this pseudokey determine the entry in the directory which must be examined. This entry contains a pointer to the appropriate data bucket. At most two disk accesses are required to locate any record in the file. If the directory can be held in the main memory, then only one access is required.

Note that more than one directory entry can point to one bucket. Associated with each bucket is a parameter called its *local depth* (*ld*). The number of directory entries pointing to a bucket is given by 2^{d-ld}. The local depth indicates that the pseudokeys contained within that bucket agree in that number of bits. We denote these bits as the *common bits* in Fig. 7.28. In an actual implementation of extendible hashing, the local depth is stored in each bucket but the common bits need not be stored. In our implementation we also store a counter *count* which shows how many keys are stored in each bucket.

As insertions occur, a bucket can become full. A new bucket is then allocated when an insertion is made to a full bucket and the contents of the original bucket and the newly inserted key are distributed over the two buckets. If the local depth of the original bucket was *ld*, then the placement of a record from this bucket will be determined by the bit $ld + 1$ of its pseudokey (numbering from the right starts at 1). The local depths of the two new buckets will be $ld + 1$ and the directory is updated so that the appropriate entry points to the newly allocated bucket. When the local depth of the original bucket is equal to the directory depth d (so that only a single directory entry pointed to it), then the directory doubles in size to enable the new bucket to be added and the depth increases by one.

Figure 7.29 shows what happens when keys 8, 13 and 16 are added to the file in Fig. 7.28. The first insertion 8 goes to bucket a making it full while 13 goes to c which is already full. A new bucket c' therefore must be added and the local depth of these two buckets becomes equal to the depth of the directory. When 16 is inserted, it goes to the bucket which is already full and therefore a new bucket a' is inserted in the file but given that bucket a had a local depth 2 which was equal to the depth of the directory, the directory must also be doubled and we obtain the result as shown in Fig. 7.29.

We recommend that the reader carefully go through the insertions 8, 13 and 16 to the file shown in Fig. 7.28 to arrive at the file in Fig. 7.29.

Similar to creation of new buckets during insertions, deletions may result in two buckets being merged. This in turn may require halving the directory and reducing its depth by one. Halving the directory is possible when the depth of the directory is greater than the local depths of each data bucket. In order to ascertain when the directory should be halved, one possibility is to store a count of the number of buckets for which the local depth equals the depth d. This count is referred to as the *depth count* (*d_count*). This is not discussed in detail here and is recommended as an exercise for the reader.

Search

In order to search for a record with key k, the hash function used in building the file is used to generate a long bit pattern or *pseudokey* when it is applied to k. Once we know the directory depth d, the rightmost d bits of this pseudokey determine the entry in the directory which must be first read from the disk (unless it is main memory resident) and then examined. This entry contains a pointer to the appropriate data bucket. The record

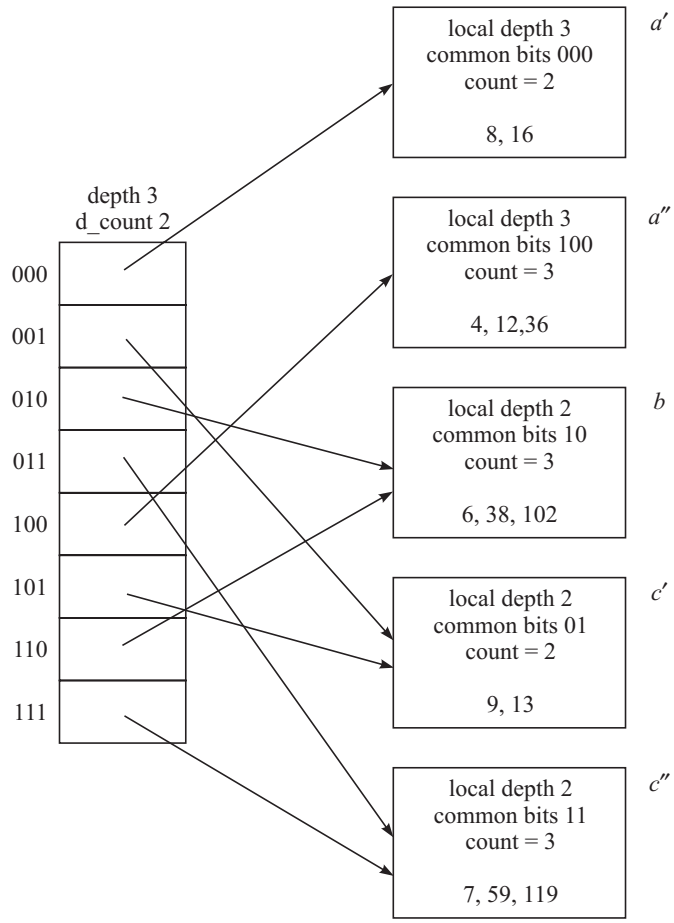

Figure 7.29 An example of bucket splitting and directory doubling in extendible hashing

with key k must be searched in that bucket. If found, we have a successful search, otherwise an unsuccessful search. Note that at most two disk accesses are required to locate any record in the file.

The performance of extendible hashing has been analysed by several researchers. We will only quote some of the important results. If N is the number of records in the system, and if the capacity of a bucket is b records, then the number of data buckets $L(N)$ required by extendible hashing is given asymptotically:

$$L(N) \sim N/(b \log_e 2)$$

If the directory can be stored in the main memory, then it follows that the storage utilization for extendible hashing is approximately $\log_e 2 \sim 0.69$. Again, assuming that the directory can be stored in the main memory, the insertion cost is two disk accesses for those insertions that do not require a bucket split, and three accesses for those that do. It can be shown that asymptotically, the average insertion cost, $I(N)$, approximates:

$$I(N) \sim 2 + 1/(b \log_e 2)$$

Deletion costs are similar to insertion costs.

Of course, if the directory is stored on disk, then these costs are higher. It can be shown that the probability that more than 2048 directory entries will be needed for a file of 40,000 records when $b = 40$ is practically 1, whereas the probability that more than 4096 entries will be needed is practically zero (it is in fact less than 0.00001). Thus, if we are satisfied with a reliability level of 10^{-5}, we can allocate 4096 entries for the directory for a file of 40,000 records with bucket capacity of 40.

Various extensions of extendible hashing have been proposed. One of these, called Bounded Exponential Hashing (BEH) is based on the argument that a desirable feature of any scheme proposed for dynamic files is that performance should be even throughout the life of the file. However, it is not quite true for extendible hashing since the pseudokeys are distributed uniformly and therefore data buckets with the same local depth are likely to contain similar numbers of records. Hence during insertion, a large number of buckets tend to split at about the same time (the time when the directory doubles). Thus the space utilization and cost of insertion in extendible hashing tends to exhibit oscillatory behaviour. BEH proposes a solution which involves distributing the pseudokeys non-uniformly to the buckets, but we will not discuss BEH further.

7.15.2 Linear Hashing

Unlike extendible hashing, linear hashing does not require a directory. Linear hashing was proposed by Witold Litwin in 1980 and is based on a hashing function which changes dynamically with the size of the file. In the extendible hashing scheme, a bucket which undergoes collision was split. This is natural but the buckets split were randomly distributed and then an index was necessary. The index is not needed if the buckets are split in a predefined order. Splitting the buckets in a predefined order is reasonable since, as noted above, when one bucket is full, all the others are also likely to be close to full, assuming a good hashing function is being used that distributes records uniformly. Therefore, the bucket where a collision takes place need not be split immediately. In linear hashing, it is split only when its turn comes in a predefined fashion after some delay.

In linear hashing, the file goes through a number of *expansions*. At the start of the d^{th} expansion[7], the file contains $s = 2^d$ home pages (see Fig. 7.30 where there are 16 buckets, $s = 16$ and d is 4). Associated with each bucket is perhaps 1 or more overflow buckets. At this stage, the bucket address for the record with key k is given by the rightmost d bits of the address generated by hashing the key k. Let us denote the address formed by these d bits as $h_d(k)$. We are using a simple hash function mod(s) which is known to be not a good hash function but is used here only for illustrating the technique. Figure 7.30 assumes bucket capacity of 2.

Now, as further records are inserted into the file, new buckets will be added to the end of the file. The rate at which new buckets are created is determined by a system parameter called the *load control*, L. With this approach a new bucket is appended to the end of the file after every L insertions. At a typical stage during the d^{th} expansion, the file will contain $s + p$ home blocks, assuming $16 < s + p < 32$. Figure 7.31 shows a linear hashing file undergoing an expansion.

A pointer, called the *split pointer*, which at the start of the d^{th} expansion points to bucket 0 of the file, now points to bucket p of the file. When another bucket is due to be appended to the end of the file, the following process takes place. Bucket p is split so that the records which previously had home address p are distributed using the value $h_{d+1}(k)$ between pages p and $s + p$. Here $h_{d+1}(k)$ is the integer formed from the rightmost

7. An expansion essentially doubles the file starting from just one bucket for which $d = 0$. In the first expansion the file goes to 2 buckets ($d = 1$) then to 4 buckets ($d = 2$) and so on.

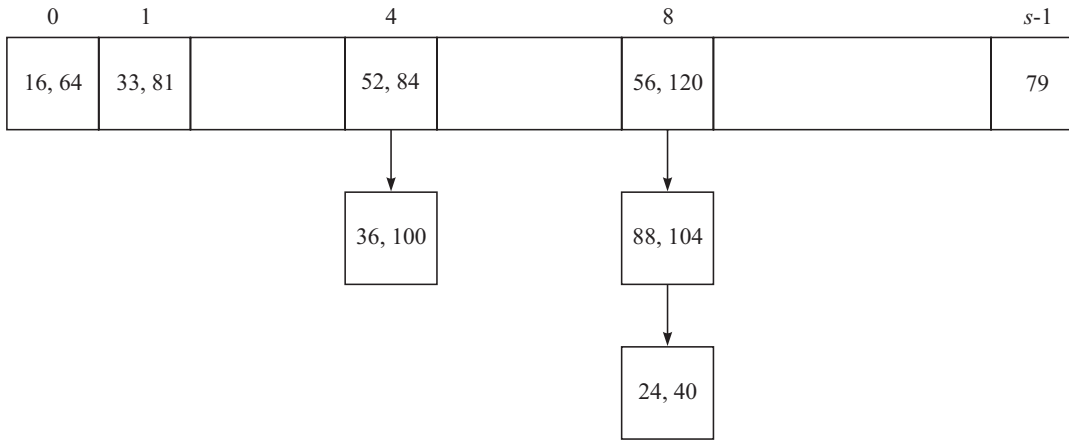

Figure 7.30 An example of a file using linear hashing

$d + 1$ bits of the address obtained by the hashing function. The split pointer is then incremented by 1. When the split pointer reaches s and the file size has doubled, the split pointer returns to the beginning of the file ($p = 0$) and the next expansion begins.

The number of expansions a file has undergone is called its *depth, d*. Initially $d = 0$. It can be seen that the current state of the file is characterized by the two variables d and p.

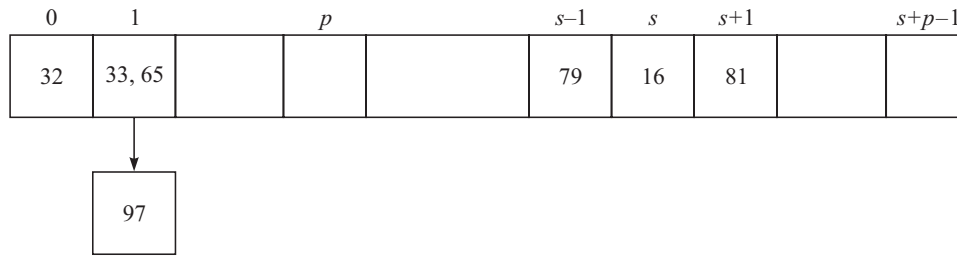

Figure 7.31 Start of the d^{th} expansion of the file

Figures 7.32(a), (b), (c) and (d) show how a full expansion, or doubling of the file size, consists of a number of partial expansions. At the start of the d^{th} full expansion, the file consists of 2^d bucket, and the split pointer is positioned at the first bucket of each group (see Fig. 7.32(a)).

When a new bucket is appended to the end of the file, the records in the buckets indexed by the split pointer are redistributed among these buckets and the newly created bucket. The split pointer is then advanced to the next set of buckets. The file grows this way until, at the end of the first partial expansion which as shown in the Fig. 7.32(b) expands the file by 50%. The next partial expansion begins with the split pointer positioned at the first page in each of the three groups. The pointers during the second partial expansion are shown in Fig. 7.32(c).

Suppose now that the file has g+1 groups, and when a new page is added, the records in these pages are distributed among the g+2 pages. At the end of the partial expansion, the file will have grown to g+2 groups of 2^d pages. Finally, after g partial expansions, the size of the file will have grown to $2g$ groups of 2^d pages.

At this stage the file is repartitioned into g groups of 2^{d+1} pages and the process starts over again. Thus a full expansion consists of g partial expansions. As in Litwin's scheme, overflow pages are used together with bucket-chaining in this method.

When expanding a set of buckets, the old buckets are scanned and only those records which are to be relocated in the new bucket are collected. Thus there is no allocation of records to the old buckets. With this approach the number of buffer pages required is minimized. However, in order to locate a record with this method, the history of the record's movements through all the previous expansions must be traced.

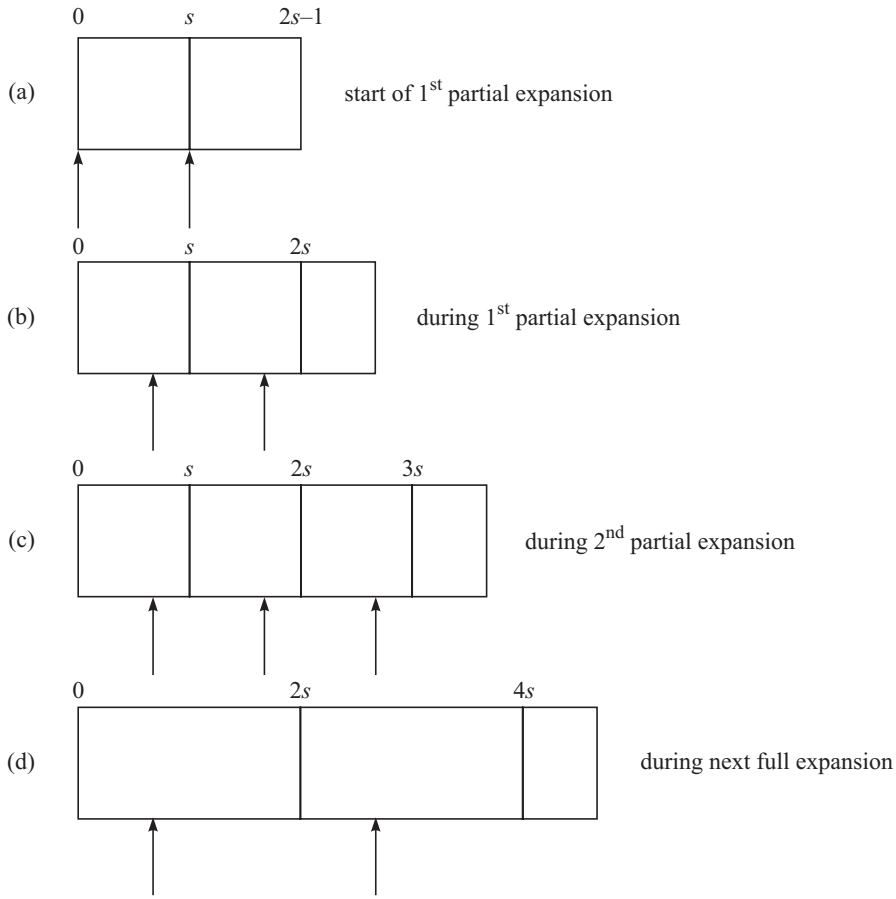

Figure 7.32 An example of file expansion using partial expansions

In Fig. 7.32, the first part shows a file that has been doubled from 16 buckets (assuming $s = 16$) to 32 buckets and is now being expanded to 64 buckets. In doubling by 100%, a partial expansion increases the size of the file by 50%. The first new bucket collects some records from bucket number 0 and bucket number 16 to put them in bucket number 32. The records collected are those that would hash to bucket 32 using the new hash function (for example, mod 64). This continues until the file is 48 buckets in size. Now the second partial

expansion starts, which adds another 16 buckets to the file. The full expansion doubles the file. Once the file is 64 buckets large after the full expansion, a new partial expansion starts when the file grows further.

Linear hashing, like extendible hashing, also suffers from the problem of oscillatory behaviour. There are a number of other difficulties with linear hashing. First, there are a number of problems associated with the use of overflow buckets which necessitate the provision and management of a large, dynamically changing storage pool. The costs of maintaining such a storage pool can be high, yet these costs are often not included in performance evaluations of schemes which use overflow buckets. Also, if the size of the overflow buckets is the same as the size of the home bucket, then performance will, in general, be poor. Thus, an overflow bucket with size different from the home bucket's size should be used. Although this will improve performance, the complexity of storage management is increased. Moreover, the optimal overflow bucket size depends on a number of file parameters and is difficult to calculate.

For a complete description of linear hashing the reader is referred to the paper by Litwin, in which various strategies for splitting a page and for deciding when to expand the file are described. In all strategies, the file grows linearly till it is as large as needed. At any stage, only few overflow buckets exist, and therefore a record on the average can be found in one access to the secondary storage. We have not defined a scheme for shrinking the file, but a load control parameter may be used for this purpose. For example, when a deletion leaves a bucket less than half full, the last bucket could be removed. The operation of removing a bucket and combining the records in it with another bucket in the file is called *grouping*. The split pointer moves one address backward when a grouping operation is performed.

Hashing techniques achieve the best performance when the records are distributed uniformly over the pages in the file. With linear hashing, the number of records per page can vary by factors of two or more over the length of the file, since a recently split bucket can have only half the number of records of a bucket about to be split. As a consequence, linear hashing suffers from the same sort of oscillatory behaviour as does extendible hashing.

Another issue related to linear hashing is as follows. Logically, a linear hashed file consists of buckets with consecutive page numbers. However, no operating system will support a large dynamic file of contiguously allocated pages without considerable cost overheads. Typically, the logical linear hashed file will be mapped onto a physical file made up of several quanta of contiguously allocated pages. The mapping will be done by means of a translation table stored in the core. An efficient implementation of recursive linear hashing, the amount of physical storage claimed at any time is just fractionally greater than the minimum required. The implementation is also designed so that split pages are written onto consecutive locations on disk. Litwin's original paper also proposes an implementation of linear hashing, for his original scheme.

Physical Address Computing

Although linear hashing techniques do not require an index, some information about the addresses of the various buckets must still be maintained. Storage on secondary storage devices is usually allocated in quanta. These quanta may be all of the same size or different sizes. Several schemes are possible.

1. *Contiguous*—address calculation is very simple.
2. *Noncontiguous*—we need a table which gives the address of each quantum. The table of addresses will need to be stored on disk if the allocation is so distributed that the table is quite large.

Table 7.19 Average performance of linear hashing with uncontrolled linear hashing

Performance		
	$b = 10$	$b = 50$
Successful Search	1.07	1.00
Unsuccessful Search	1.19	1.06
Insertion Cost	3.14	2.35
Split Cost	5.99	5.98
Load Factor	0.61	0.59

(Note that 2 accesses are needed to store the overflow record and 3 to split during splitting)

Table 7.20 Average performance of linear hashing with controlled linear hashing

	$b = 10$		$b = 50$	
	$g = 0.75$	$g = 0.90$	$g = 0.75$	$g = 0.90$
Successful Search	1.14	1.57	1.05	1.35
Unsuccessful Search	1.37	2.48	1.27	2.37
Insertion Cost	3.42	4.68	2.62	3.73
Split Cost	6.34	9.48	6.24	9.02

Concluding Remarks about Hashing

Hashing is an important data organization technique that has several advantages over indexing methods. Firstly, hashing usually requires no index (and therefore no storage for an index). Secondly, insertion and searching are usually very fast when hashing is used although deletion may not always be fast. Thirdly, the performance of hashing techniques is generally not dependent on the size of the table; the performance only depends on the file load factor. The primary disadvantage of hashing is that sequential processing and range queries are usually very expensive.

7.16 INVERTED FILES

A file with secondary indices is often called an *inverted* file. Rather than listing attributes for each record, we now list records for each attribute value. The roles of records and attributes have therefore been 'inverted'. This description is not quite accurate because an inverted file normally still maintains the normal file which includes full records, including all values of all attributes. This file could well be a sequential file on the primary key. A good example of an inverted file is a review journal which has a list of article titles, their reviews and a keyword index which lists keywords and lists of article numbers that contain each keyword.

Inverted files are particularly useful for searching documents, for example, through a search engine like Google. An inverted file essentially consists of lists (called inverted lists) of files (or documents) for each

query term. For example, for a query term 'Mumbai' the inverted list corresponding to it will consist of one entry for each document that has the term 'Mumbai' in it.

In search engines, a search would often involve a number of keywords. The simplest approach is to build an inverted index on every conceivable keyword (there are obviously millions of keywords since they not only include all the words in the language but also names of people, places and things like medicines) and then use these inverted indexes to find the list of documents that have each of the keywords specified in the search and then find the intersection of the lists. This final list then gives us the records that have all the keyword values, which have been specified. For large collections of documents, for example, a web document collection crawled by a search engine like Google (estimated to be above 10 billion in 2008), not only the number of inverted lists very large but each inverted list can be large. Since the number of lists is in millions, an index is needed to find the inverted list corresponding to each keyword. To reduce the number of inverted lists and the size of the lists, the search engines do not make inverted lists for common words like ''the'', ''an'', ''is'', ''are'' and the words are usually stemmed to reduce the number of keywords (for example, computing, computation, computer and computerization may all be stemmed to compute).

For example, if one were to search for keywords ''Gopal'' ''Gupta'' and ''database'', it will be necessary to first find inverted lists for each of the three keywords and then find the intersection of the three. The first two lists might be somewhat smaller (although ''Gupta'' is a common name in India and even outside India) than the inverted list for ''database''. It would be interesting to design efficient algorithms that would retrieve a number of inverted lists of different lengths that are to be intersected.

SUMMARY

This chapter has covered the following topics:

- The efficiency of a database depends on how it is physically stored in computer storage.
- Different types of storage or memories are available for storing databases. These include primary memory, secondary memory and tertiary memory.
- Primary memory includes registers, cache memory and RAM.
- Secondary storage includes hard disks, external disks, flash memories, CD, DVD.
- Tertiary storage includes low cost storage like magnetic tapes.
- Magnetic disks are usually a spindle holding a number of rotating hard platters each of which consists of tracks that are divided into sectors.
- Data is stored on a magnetic disk as blocks and data transfer from a disk and to a disk is carried out by a buffer manager.
- RAID are Redundant Array of Independent Disks that provide increased capacity, increased throughput, improved data integrity and fault tolerance. There are a number of levels of RAID technology. Level 5 is commonly used for database systems.
- Buffer management including policies for replacing buffer pages is described. The policies include LRU and FIFO.
- The efficiency of file structures is measured by their efficiency in update, insertion, deletion and search.

- Unstructured and unordered files are simple file organization techniques.
- Sequential or sorted files have records arranged in some order which must be maintained in insertions and deletions.
- File performance may be improved by using an index which may be of different types (including primary and secondary indexes).
- An indexed sequential file is an efficient structure.
- A B-tree can form the basis of an indexed sequential file since it is efficient for search, insertion and deletion.
- The algorithms for search, insertion and deletion are presented and performance of B-tree is analyzed.
- An indexed sequential file based on a B-tree is called the B^+-tree. This structure improves the efficiency of the B-tree index.
- Another data structure for storing files is called hashing. The basic concepts of static hashing, hash functions and collision resolution are described.
- Performance of static hashing techniques is discussed and it is shown, for example, that linear probe and double hashing do not perform well when the load is high.
- Static hashing is only suitable for tables in the main memory. For dynamic files on external storage, dynamic hashing may be used. Two types of dynamic hashing are extendible and linear hashing.
- Concept of inverted files in which records are listed for each keyword are suitable for searching large collections of documents and are used by search engines.

REVIEW QUESTIONS

1. Explain computer memory hierarchy; and primary, secondary and tertiary storage. Give examples of each and explain the major differences between them. (Section 7.2)
2. Describe the types of storage media that can be used for storing a database. (Section 7.2)
3. Where is database normally stored? What are the reasons for storing it there? (Section 7.2)
4. Given that large main memories are becoming available, could the whole database be stored in one large memory? Discuss. (Section 7.2)
5. How is data stored on a magnetic disk? What is a track, sector, cylinder, and block? (Section 7.3.1)
6. What is RAID? What are the major benefits of using RAID compared to an ordinary magnetic hard disk? (Section 7.3.2)
7. Describe RAID Level 0, Level 1 and Level 5. For each level RAID list at least one suitable application and give reasons for the suitability. (Section 7.3.2)
8. What are the basic processes needed in memory buffer management? (Section 7.4)
9. How should one measure the performance of a storage structure? (Section 7.5)
10. Describe unstructured and sequential files. When are they useful? Could a relational database be stored as a collection of unstructured files? Explain. (Section 7.6 and 7.7)
11. Explain the concept of index and index types. What type of indexes may be created in SQL? (Section 7.8)
12. Describe primary index, clustered index and secondary index. (Section 7.8)

13. Give an example showing when a composite index may be useful. (Section 7.8.5)
14. Explain the basic features of an indexed-sequential file. Why is it better than the sequential file? (Section 7.9)
15. Explain the concept of a B-tree and discuss why tree-structured files are efficient for some types of searching. (Section 7.10)
16. How can one determine a suitable order for a B-tree? Would an order of 100 be suitable for all type of large tables stored on disk? (Section 7.10)
17. Describe insertion, deletion and search algorithms for a B-tree. (Section 7.10.1, 7.10.2, 7.10.3)
18. Explain node splitting and node coalescing algorithms in a B-tree. When are they required? (Section 7.10.2, 7.10.3)
19. Derive the expressions for the worst case and the best case performance of a B-tree of order *m*. (Section 7.10.4)
20. Describe the B^+-tree and why it is used more commonly than the B-tree? (Section 7.11)
21. Can the B^+-tree be modified to improve performance? Give details. (Section 7.11)
22. How does a B-tree perform in terms of searching, insertion, deletion and storage utilization? What type of queries is the B-tree efficient for in large databases? (Section 7.12)
23. Explain the concepts of static hashing, hash functions and collision resolution. (Section 7.13)
24. Explain the concept of dynamic hash files in contrast to static hash files. What is the main motivation for using dynamic hashing? (Section 7.14, 7.15)
25. Describe the extendible hashing technique. Explain how search, insert and delete operations are carried out. Do we need node splits in this scheme? (Section 7.15.1)
26. Explain the concept of linear hashing. Explain how search, insert and delete operations are carried out. Do we need node splitting in linear hashing? (Section 7.15.2)
27. When is it advisable to use a hashed file in storing a database? (Section 7.14)
28. Describe an inverted file. What is it used for? (Section 7.16)

SHORT ANSWER QUESTIONS

1. Give two examples of primary storage.
2. Give two examples of secondary storage.
3. Give two examples of tertiary storage.
4. What operations are carried out on data that is stored in a database?
5. Where is a database resident most of the time?
6. What are the two major differences between the main memory and hard disk storage?
7. How much data can be stored on a single platter hard disk?
8. What is the main purpose of creating an index on a file?
9. What is a covering index?
10. What are the advantages of using an index and what are its disadvantages?
11. Give examples of two different types of indexes?
12. Give an example to show that ordering of a composite index can impact performance.

13. Explain the basic features of an indexed-sequential file.
14. How many keys would a B-tree of order 100 accommodate if its height was 2 and it was full?
15. Briefly explain how to search a B-tree.
16. What is node splitting in a B-tree?
17. What are the advantages of using the B-tree compared to an indexed sequential file?
18. Describe B^+-tree.
19. Explain the concept of static hashing.
20. What is collision resolution?
21. What are dynamic hash files designed to achieve?
22. What is the major difference between extendible hashing and linear hashing?
23. Give an example of an application of an inverted file.

MULTIPLE CHOICE QUESTIONS

1. Which one of the following is **not** a type of computer memory?
   (a) Primary storage (b) Secondary storage (c) Tertiary storage (d) Permanent storage
2. Which of the following are correct?
   (a) Cache is primary storage.
   (b) Flash memory is secondary storage.
   (c) Magnetic disk is tertiary storage.
   (d) Magnetic tape is secondary storage.
3. Which one of the following is correct?
   (a) RAID Level 2 has stripping but no redundancy.
   (b) RAID Level 0 uses error correction using Hamming Code.
   (c) RAID Level 1 is not suitable for database systems.
   (d) RAID Level 5 is suitable for most database systems.
4. Which of the following are features of RAID 3?
   (a) It is a high performance RAID system.
   (b) It involves striped segments written on two or more disks.
   (c) It involves an additional disk to store parity information.
   (d) It is not suitable for transaction-oriented databases.
   (e) All of the above are correct.
5. Which of the following are techniques used for replacing buffer pages?
   (a) Least recently used (b) Last in first out (c) First in first out (d) Most recently used
6. Which of the following are normally used to evaluate storage structures?
   (a) Cost of insertion (b) Cost of retrieval (c) Cost of deletion (d) Storage utilization
   (e) All of the above
7. Which one of the following is correct?
   (a) Sequential files provide fast access to the next record.
   (b) Sequential files provide fast access for random access.
   (c) Sequential files provide fast updates of file.
   (d) Sequential files are difficult to implement.

8. Which one of the following is **not** an indexing technique?
   (a) Primary index (b) Secondary index (c) Multilevel index (d) Sequential index
9. Which one of the following is correct about an indexed sequential file?
   (a) The file has a prime area. (b) The file has an overflow area.
   (c) The file has an index area. (d) All of the above
10. Which one of the following is **not** true about a B-tree?
   (a) All nodes including the root must be at least half full.
   (b) All leaf nodes must be at the same level.
   (c) All nodes with k keys except the leaves must have $k + 1$ descendents.
   (d) The height of the tree grows when the root splits.
11. The maximum number of keys in a B-tree of order 11 of height 2 is (the root is at level 0):
   (a) Less than 2,000 (b) Between 2,000 and 6,000
   (c) Between 6,000 and 18,000 (d) Between 18,000 and 50,000
   (e) More than 50,000
12. A B-tree of order 5 is built by inserting the keys 10, 20, 30, 25, 15, 5, 17, 27, 37, 35, 32 and 22 in that order to a null tree.
   Which one of the following statements is correct about the resulting tree?
   (a) The tree is of height 2. (b) The root node has keys 10, 20, 25 and 30.
   (c) The root node has keys 20 and 30. (d) Altogether there are 5 nodes in the tree.
13. A B-tree is built up by inserting the keys in the last question in the reverse order (that is, by first inserting 22 to a null tree, then inserting 32, 35, 37 and so on).
   Compare the resulting tree with the tree obtained in the last question. Which one of the following statements is correct?
   (a) The two B-trees are identical.
   (b) The number of nodes in the two trees is the same.
   (c) The storage utilization in the two trees is the same.
   (d) The height of the two trees is the same.
14. Which one of the following is **not** correct?
   (a) Composite indexes are larger than single indexes.
   (b) Composite indexes should be ordered from most distinct to least distinct.
   (c) Consider a composite index on the table *Batting* on *MID* and *PID*. Let there be 3000 matches and 50,000 batting records which include only 500 distinct *PIDs*. An index on (*PID*, *MID*) would be better than an index on (*MID*, *PID*).
   (d) There is not much advantage in having a composite index on the table *Player*.
15. Which one of the following is correct?
   (a) A binary tree is not suitable for storing information on a disk since disk read/write are by a block.
   (b) A binary tree is effective for small files on the disk since binary tree looks at one node at a time .
   (c) A binary tree is a special case of a B-tree.
   (d) B-tree is not suitable for secondary indexing.
16. Which one of the following is **not** true about extendible hashing?
   (a) It consists of an index as well as a data file.
   (b) The technique can be modified to make sequential access easier.

(c) If the index is kept in the main memory, retrieval from an extendible hashing structure requires only one access to the disk.
(d) Storage utilization is approximately 69%.
(e) None of the above

17. Which of the following statements is **not** true?
(a) Linear hashing does not need an index as extendible hashing does.
(b) Linear hashing has better storage utilization than extendible hashing.
(c) Linear hashing files shrink and grow as required.
(d) Linear hashing requires overflow buckets.
(e) None of the above

18. Linear hashing and extendible hashing have a number of common features. Which one of the following is **not** true?
(a) They both involve splitting the block where an overflow occurs.
(b) They both allow the file to expand or shrink dynamically as required.
(c) They both change the hashing function dynamically with the size of the file.
(d) They both combine the node-splitting feature of B-trees with that of conventional hashing.
(e) None of the above

19. Some B-tree structures store all the records in the leaves only. Such trees are called B^+-trees. Which one of the following statements is **not** correct?
(a) In a B^+-tree the records and the index are separate.
(b) Sequential processing of the records is facilitated in B^+-trees.
(c) If the order of the tree is large, say 100, storing all the records in the leaves does involve substantially more storage than used by conventional B-trees.
(d) Since the index needs to store only the keys and not the records, the order of the index can be higher and therefore searching can be faster in B^+-trees.
(e) None of the above

20. What is the maximum number of nodes in a B-tree (of order 5) of height 2?
(a) 128 (b) 255 (c) 129 (d) 127
(e) None of the above

21. Which of the following advantages apply to files organized using a B-tree scheme rather than using hashing?
(a) B-tree files can be sequentially processed with much lower cost than the hashed files.
(b) B-tree files are better because with such files it is possible to carry out range queries at much lower cost.
(c) B-tree files are better because with such files it is easier to have multiple attribute retrieval.
(d) B-tree files are better than hashed files because they do not involve using overflow buckets.

22. Which one of the following is **not** correct?
(a) The optimum node size for disk resident B-trees should be selected based on rotational delay time, seek time, transfer rate and key size.
(b) A file organization based on B-trees is guaranteed to have at least about 50% storage utilization.
(c) A file organization based on B-trees on average has storage utilization of about 69%.
(d) A B^*-tree guarantees storage utilization of at least about 75.0%.

23. The following tree only shows the keys present in each node. It does not show the size of the nodes or their capacity. All the nodes are of equal size.

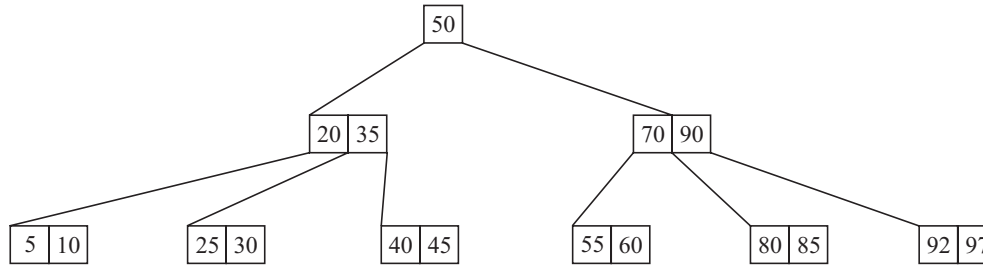

Figure 7.33

Which one of the following statements is correct?

(a) The above tree is not a B-tree since the storage utilization is less than 50%.
(b) The above B-tree must be of order 3.
(c) The above tree must be of order 3 or 4.
(d) The above tree could be of order 2, 3 or 4.
(e) The above tree could be of order 3, 4 or 5.

24. This question also refers to the B-tree displayed in the last question. Which one of the following statements is **not** correct?

(a) The above B-tree has the minimum number of nodes possible in a B-tree of order 5 and height 2.
(b) The above B-tree has the maximum number of nodes possible in a B-tree of order 3 and height 2.
(c) If the order of the above B-tree is 5, the height of the tree may not increase even if 100 keys are inserted in the tree.
(d) If the order of the above B-tree is 5, insertion of another 40 keys may increase the height of the tree.

25. Which one of the following statements is **not** correct?

(a) The height of a B-tree grows only when the root is split.
(b) In the worst case, only 4 accesses are necessary if a file of about 2 million records is stored as a B-tree of order 200.
(c) In the best case, about 16 million records could be stored in a B-tree of order 200 and height 3.
(d) In the worst case, a B-tree of height 2 may contain only half the number of records that can be stored in a B-tree, of the same height, in which storage utilization is 100%.

26. Which one of the following is **not** correct regarding $T1$ and $T2$ below?

(a) If key 30 is deleted from the tree $T2$, either 22 or 32 is moved to the root node.
(b) If key 32 is deleted from the tree $T1$, two rightmost nodes at level 1 and key 30 from the root node are combined.
(c) If key value 2 is inserted into the tree $T2$, the left node at level 1 is split and the key 10 is moved to the root.
(d) If key value 22 is deleted from the tree $T1$, 20 is moved to the leaf node which underflows and 17 is moved to the root.
(e) None of the above

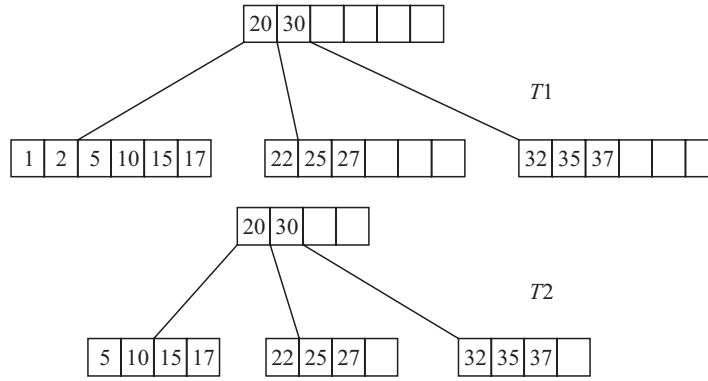

Figure 7.34

27. Given the B-tree of order 5 in Fig. 7.35, which one of the following sequence of operations is correct when the keys 25, 26 and 24 are deleted?
    (a) The key 25 is deleted from the tree, the root disappears and the nodes *B* and *C* are combined. Deletion of 26 results in borrowing of key 29 from the node *C* and moving of 32 to the node *C*. Deletion of 24 similarly involves moving of 19 down to the node *F* and of 18 up to the node *B*.
    (b) The key 25 is deleted and a key from the leaves is moved to the root. If 26 is moved to the root, the key 29 moves down to the node *G* while 32 moves up to the node *C*. Deletion of 26 then involves moving 27 to the root and concatenation of the nodes *G* and *H* and key 32. Deletion of 24 then causes 19 to be moved to the node *F* and 18 to be moved to the node *B*.
    (c) When the key 25 is deleted, 26 is moved to the root. The node *G* underflows and this results in key 29 moving down to the node *G* while 32 moves up to the node *C*. Deletion of 26 then involves moving 27 to the root and concatenation of the nodes *G* and *H* and key 32. This makes the node *C* underflow and this results in concatenation of the node *C*, node *B* and the root to form the new root. The height of the tree is therefore reduced. Deletion of 24 now involves moving 19 to the node *F* and 18 to the node *B*.
    (d) None of the above

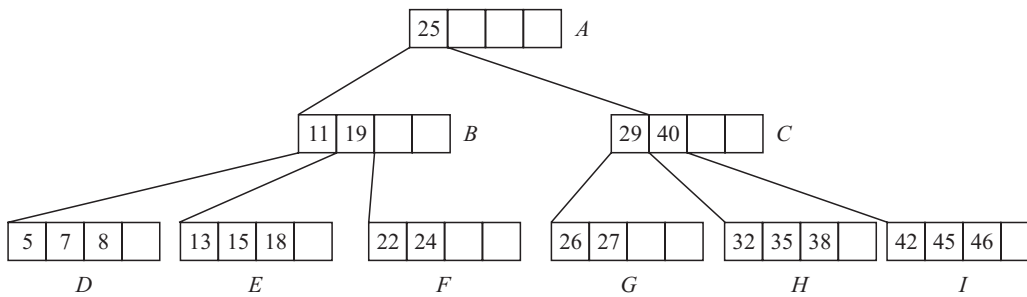

Figure 7.35

28. Which of the following are correct about conventional hashing?
    (a) It is difficult to find hashing functions that will uniformly distribute keys over the hash table.
    (b) Even when the hash table is of a reasonable size, overflow chains become long.

(c) If the file size grows beyond what was expected, the performance will decline rapidly.
(d) If the file shrinks much below what was expected, storage utilization will be low.
(e) None of the above

EXERCISES

1. What is the main storage media for databases? Why? Could a flash drive be used for storing a database? Explain.
2. Explain how reading from a disk and writing to the disk is carried out? What is a block? Why do we need a disk to be divided into blocks?
3. Explain the role of a buffer manager in a database system.
4. How are tables of a database stored in a computer system? How is data on a disk transferred to the main memory and vice versa?
5. What operations are carried out on data that is stored in a database?
6. List some different data structures that are possible for storing data on disk.
7. What is an index? Describe the different types of indexes and give examples to illustrate when each of them is useful. Given the cricket database in Chapter 3, give a set of queries and then discuss what data structure might be the best if the database is quite large.
8. Give an example of an indexed-sequential file and show how insertion, deletion and search are carried out in such a file.
9. Discuss the major differences between an indexed-sequential file and B-trees.
10. How does the B^+-tree differ from a B-tree in terms of search, insertion and deletion? Which of the two structures is preferred as an access structure in database management and why?
11. Suggest a modification to improve the performance of B^+-tree indexing. Write an algorithm to show how the modification will work.
12. How do dynamic hashing schemes differ from the classical hashing schemes? Why is the classical technique not suitable for storing information on disk?
13. Discuss extendible and linear hashing. How can they be used for constructing an index?
14. Implement one of the following schemes including modules for insertion, update, retrieval and deletion of records. The programs should be able to deal with alphanumeric keys and be able to store records that have several attribute values in addition to the key. You should also evaluate the performance of the technique that you have implemented. For example, you should be able to compute the average cost of insertion, deletion and retrieval as well as the average storage utilization.
    1. Indexed-sequential file
    2. B^+-tree
    3. Extendible Hashing
    4. Linear Hashing
15. Suggest two schemes that would improve the efficiency of a B^+-tree scheme.
16. How is the cost of different data structures for storing databases compared?
17. Show how an indexed-sequential file is maintained when rows are being inserted and deleted from a table.

18. A B-tree of order 5 is built by inserting the keys 1,2,3,4,...20 in some order.
    (a) Which order of insertion of the keys would lead to the resulting tree having as high storage utilization as possible?
    (b) Write the keys in that order.
19. Consider the B-tree shown in Fig. 7.11. Describe what will happen if the key 11 is deleted from the tree. Show the tree after the deletion.
20. Consider a file being built with bucket capacity of 4. The key values are integers and the hashing function to be used is (key value) mod 2^i (i.e., least significant i bits) for a suitable i. The following key values are added to a null file: 1, 2, 17, 20, 39, 40, 5, 6, 9, 11, 22 and 21.
    Using extendible hashing:
    (a) What is the size of the directory after these insertions?
    (b) How many buckets are in the file?
21. Consider the file built-up in the last question using extendible hashing. The following keys are now deleted from the file: 5, 9, 1 and 11. How many buckets have been removed from the file?
22. Consider a file being built with bucket capacity of 4. The key values are integers and the hashing function to be used is (key value) mod 2^i for a suitable i. The following key values are added to a null file: 1, 2, 17, 20, 39, 40, 5, 6, 9, 11, 22, 21, 19, 23, 7, 13 and 15.
    If the file is built-up using linear hashing with uncontrolled splitting (that is, a split occurs every time a record goes to an overflow bucket), answer the following:
    (a) How many buckets are in the file?
    (b) What was the maximum number of overflow buckets used at any one time?
23. Consider the file built-up in the last question using linear hashing. The following keys are now deleted from the file: 5, 7, 9, 1 and 11. How many buckets have been removed from the file?

PROJECTS

1. Given the decreasing cost of main memory, it is possible to store a whole database in the main memory. Explore the changes that would be necessary to make the database perform well. What physical storages should be used? Would B-tree and dynamic hashing schemes still be suitable? What bucket size would now be appropriate?
2. Implement the classical B-tree in C++. Carry out random searches after building it and carry out some random deletions and insertions in order to evaluate its performance.

LAB EXERCISES

1. Suppose we have a database in which two attributes are the latitude and longitude of the location of the entity. A lot of queries involve the latitude and longitude. Explore how the data should be stored. What index would be most effective?

2. Consider a database like that used in the search engines. Most queries involve one or two words but some queries include several search words. What data structure is most suitable for such a document database and the queries?

BIBLIOGRAPHY

For Students

Comer's paper is a good reference on B-trees.

For Instructors

We have presented some original papers, for example, by Fagin, Nievergelt, Pippenger and Strong. Knuth's Volume 3 book has a lot of detail about file structures. The book by Frake and Baeza-Yates is also useful and so is the book by Lightstone, Teorey and Nadeau.

Comer, D., "The Ubiquitous B-Tree", *ACM Computing Surveys*, Vol. 11, No. 2, June 1979.

Dippert, B., and M. Levy, *Designing with Flash Memory*, Annabooks, 1993.

Elmasri, R., and S. B. Navathe, *Fundamentals of Database Systems*, Addison-Wesley, Chapters 13 and 14, Fifth Edition, 2003.

Enbody, R. J., and H. C. Du, "Dynamic hashing schemes", *ACM Computing Survey*, Vol. 20, June 1988, pp 850-113.

Fagin, R., J. Nievergelt, N. Pippenger, and R. Strong, "Extendible Hashing – A fast access method for dynamic files", *ACM Transactions of Database Systems*, Vol. 4, pp 315-344, 1979.

Frake, W. B., and R. Baeza-Yates, *Information Retrieval: Data Structures and Algorithms*, Prentice Hall, 1992.

Knuth, D. E., "The Art of Computer Programming", *Sorting and Searching*, Vol. 3, Addison-Wesley.

Lightstone, S., T. Teorey, and T. Nadeau, *Physical Database Design: The Database Professional's Guide to Exploiting Indexes, Views, Storage, and More*, Morgan Kaufmann/Elsevier, 2007.

Litwin, W., "Virtual Hashing: A Dynamically Changing Hashing", *Proceedings 4th Conference on Very Large Data Bases*, Berlin, 1978, pp 517-523.

Ruemmler, C., and J. Wilkes, An Introduction to Disk Drive Modelling, *IEEE Computer*, 27:3, March 1994, pp. 17-27.

Vitter, J. S., "External Memory Algorithms and Data Structures: Dealing with Massive Data", *ACM Computing Survey*, Vol. 33, June 2001, pp 209-271.

Zobel, J. A., A. Moffat, and K. Ramamohanrao, "Inverted File Versus Signature Files for Databases", In *Proceedings International Conference on Very Large Data Bases*, Vol. 18, Vancouver, 1992.

CHAPTER 8

Transaction Management and Concurrency

OBJECTIVES

- Explain the concept of a transaction.
- Describe the four properties of DBMS transactions.
- Explain why concurrency is important.
- Discuss when concurrent execution of two transactions leads to conflict.
- Give examples of concurrency anomalies.
- Explain the concept of a schedule.
- Study the concepts of serializability, final state serializability (FSR), view serializability (VSR) and conflict serializability (CSR).
- Discuss how to test for CSR.
- Describe how serializability can be enforced using locking and timestamping.
- Discuss the concept of deadlocks, their prevention and resolution.
- Explain the concept of lock granularity, multiple granularity.
- Describe use of intention locking in multiple granularity.
- Describe timestamping concurrency control.
- Describe the optimistic concurrency control.
- Briefly look at evaluation of concurrency control techniques.

KEYWORDS

Concurrency, transactions, ACID, atomicity, consistency, isolation, durability, concurrency anomalies, lost update, inconsistent retrievals, schedule, serial schedule, serializability, final state serializability FSR, view serializability VSR, conflict serializability, CSR, testing for CSR, precedence graphs, recoverability, hierarchy of serializability, 2PL, locking, timestamping, deadlocks, prevention, resolution, detection, wait-for graph, granularity, fine granularity, coarse granularity, multiple granularity locking, intention locking, intention shared (IS), intention exclusive (IX), recoverability, optimistic concurrency control, evaluation.

A moment comes, which comes but rarely in history, when we step out from the old to the new; when an age ends; and when the soul of a nation long suppressed finds utterance.

Jawaharlal Nehru (on the independence of India, 15 August 1947)

8.1 INTRODUCTION

Most modern database systems, except personal databases, are multi-user systems. Some large systems support thousands of users. Multiple users are able to use a single system simultaneously because of multiprogramming in which the processor (or processors) in the system is shared amongst a number of users trying to access computer resources (including databases) simultaneously. The concurrent execution of programs is therefore interleaved, with each program being allowed access to the CPU at regular intervals. This also enables a program to access the CPU while another program is doing input/output. In this chapter, we first discuss the concept of a transaction which may be defined as execution of a user program in a database. Since many users are using a multiuser system concurrently, the transactions must be managed so that they do not interfere with each other. In this chapter we discuss the problem of synchronization of access to shared objects in a database while supporting a high degree of concurrency.

Concurrency control in a database system permits many users, assumed to be interactive, to access a database in a multiprogrammed environment while preserving the illusion that each user has sole access to the system. Control is needed to coordinate concurrent accesses to a DBMS so that the overall correctness of the database is maintained. Efficiency is also important since the response time for interactive users ought to be short. The reader may have recognized that the concurrency problems in database management systems are related to the concurrency problems in operating systems although there are significant differences between the two.

Clearly no problem would arise if all users were accessing the database only to retrieve information and no one was modifying it as, for example, when accessing census data or a library catalogue. If one or more users are modifying the database, for example, a bank account or airline reservations, an update performed by one user may interfere with an update or retrieval by another. For example, users A and B both may wish to read and update the same record in the database around the same time. The relative timing of the two transactions may have an impact on the state of the database at the end of the transactions. The end result can be an inconsistent database.

Our discussion of concurrency will be transaction based. A transaction T is a sequence of actions [t_1, t_2, … t_n]. A transaction is assumed to be a logical unit of work and a unit of consistency as it preserves database consistency. Transaction processing database systems are now ubiquitous. For example, they include e-commerce, billing systems like telecommunications accounting, payroll, airline and hotel reservations, banking and electronic stock market transactions, to name a few.

This chapter is organized as follows. The concept of a transaction is defined followed by a discussion of the properties of a transaction and the importance of concurrency control. Examples of concurrency anomalies, lost updates, inconsistent retrieval, and uncommitted dependency, are described in Section 8.3. Sections 8.4 and 8.5 present the major concepts of concurrency control. Section 8.6 defines serializability and deals with techniques of concurrency control by enforcing serializability. Section 8.7 deals with hierarchy of serializable

schedules. Section 8.8 describes concurrency control and enforcing serializability including the concepts of locking and two-phase locking (2PL). Definition of deadlocks, their detection, prevention and resolution are discussed in Section 8.9. Section 8.10 explains lock granularity while Section 8.11 explains multiple granularity and intention locking. Sections 8.12 and 8.13 describe nonlocking techniques of concurrency control. Section 8.12 deals with timestamping control and Section 8.13 with optimistic control. The chapter concludes with a brief look at transaction support in SQL in Section 8.14 and evaluation of concurrency control techniques in Section 8.15.

8.2 THE CONCEPT OF A TRANSACTION

We first need to define the concept of a *transaction*.

Definition—Transaction
A transaction is defined as a logical unit of work which involves a sequence of steps but which normally will be considered by the user as one action and which preserves consistency of the database.

For example, transferring an employee from department *A* to department *B* may involve updating several tables in a database but would be considered a single transaction. Transactions are straight-line programs devoid of control structures. The sequence of steps in a transaction may lead to an inconsistent database temporarily but at the end of the transaction, the database is expected to be in a consistent state. It is assumed that a transaction always carries out correct manipulations of the database.

The concept of a transaction is important since the user considers it as one unit and assumes that a transaction will be executed in isolation from all other transactions running concurrently that might interfere with the transaction. We must therefore require that actions within a transaction be carried out in a prespecified serial order and in full, and if all the actions do not complete successfully for some reason then the partial effects must be undone. A transaction that successfully completes all the actions is said to *commit*. It otherwise aborts and must be *rolled back*. For example, when a transaction involves transferring an employee from department *A* to department *B*, it would be unacceptable if the result of a failed transaction was that the employee was deleted from Department *A* but not added to Department *B*.

We summarize the ACID properties of a transaction now.

8.2.1 Atomicity

As noted above, although a transaction is conceptually atomic, it usually consists of a number of steps. It is necessary to make sure that the outside world and other transactions do not see partial results of a transaction execution and therefore either all actions of a transaction are completed without any errors or the transaction is abnormally terminated and has no effect on the database. Therefore, a transaction is either completed successfully and commits or is rolled back as if it had never been invoked. This is sometimes called *all-or-nothing*. In the section on recovery we will discuss how to recover from a crash while transactions are being executed.

8.2.2 Consistency

Although a database may become inconsistent during the execution of a transaction, it is assumed that a completed transaction preserves the consistency of the database. This consistency preservation cannot be completely automatic[1] and often requires the application program running the transaction to carry out a variety of checks which ensure that database consistency is preserved. For example, when a user carries out a bank transaction, it is assumed that the consistency of the bank accounts is not violated. When a transaction is run on an ATM, strange incidents can happen. For example, the power might go off after the card is returned but before the cash is dispensed. The transaction must maintain consistency in such unusual situations. Further, we will discuss how to recover from such database crashes while one or more transactions are being executed. Any transaction that does not pass the checks of the application program is abnormally terminated. Whether a transaction completes or is terminated, the consistency of the database is thus ensured.

8.2.3 Isolation

As noted earlier, no other transactions should be able to view any partial result of the actions of a transaction since intermediate states may violate consistency. It is essential that other transactions only view results of committed transactions. Also, no transaction should interfere with the actions of another transaction. Each transaction must be executed as if it was the only transaction being carried out and the user should feel as if all transactions are being carried out one after another. This requirement ensures that the detail of concurrency are hidden from the application developers making their task of writing application programs easier.

8.2.4 Durability

A transaction that is for some reason interrupted in the middle of its execution must be aborted so that the database is left consistent. Also, once a transaction has been completed successfully and has committed, its effects must persist and a transaction must complete before its effects can be made permanent. The results of a committed transaction are durable. A committed transaction cannot be aborted and its effects cannot be undone by a subsequent software or hardware failure. Also a committed transaction may have seen the effects of other transactions but we assume those transactions had committed before the present transaction saw the effects.

The result of all transactions following the ACID properties is that the users can rely on the consistency of the data they see whether the database is a centralized system or a distributed system. Furthermore, as noted above, these properties make it easier for application developers to develop applications. These ACID properties represent the first letters of the four properties above. A summary of these properties in presented in Fig. 8.1.

When an application program specifies a transaction we assume that it is specified in the format given in Fig. 8.2.

1. Some tasks of checking consistency can be automatically carried out by the DBMS since the DBMS will check a variety of constraints that are declared for the database including those declared in triggers. We discussed integrity constraints in Chapter 3 and 5 and will discuss them in more detail in Chapter 13.

	Property	What is it?
A	Atomicity	All or nothing.
C	Consistency	Each transaction leaves the database consistent; otherwise rolled back.
I	Isolation	Each transaction is executed as if it was the only one running.
D	Durability	A committed transaction cannot be aborted and its effects cannot be undone.

Figure 8.1 A summary of the ACID properties of transactions

All the actions between the Begin and Commit are now considered part of a single transaction. If any of these actions fail, all the actions carried out before the failure must be undone. We assume that transactions are not nested.

```
Begin Transaction
(details of the transaction)
Commit
```

Figure 8.2 Format of a DBMS transaction

We will use the classical example of a bank account transfer transaction to illustrate some of the concepts used in transaction management. The transaction is given in Fig. 8.3.

```
Begin Transaction
Transfer Rs. 100 from account A to account B
Commit
```

Figure 8.3 An example of a transaction

It is clear that there needs to be a transfer program in the system that will execute the transaction. The steps given in Fig. 8.4 are involved in carrying out the transaction.

Step	Action
1	Read account *A* from disk
2	If balance is less then Rs. 100, return with an appropriate message
3	Subtract Rs. 100 from balance of account *A*
4	Write account *A* back to disk
5	Read account *B* from disk
6	Add Rs. 100 to balance of account *B*
7	Write account *B* back to disk

Figure 8.4 Actions of an example transaction

We have ignored several small details. For example, the steps in Fig. 8.4 do not check if accounts *A* and *B* exist. Also it has been assumed that the application program has the authority to access the accounts and transfer the money.

In the above algorithm, a system failure at step 5 or step 6 would leave the database in an inconsistent state since both would result in Rs. 100 being subtracted from account *A* and not added to account *B*. A recovery from such failure would normally involve the incomplete transaction being rolled back.

We assume that all transactions complete successfully and any problem of transaction failures are resolved by the recovery mechanisms discussed in Chapter 8. The only detail of transactions that is of interest right now is the reads and writes although other computations would often be carried out between the reads and writes. We therefore assume that all actions t_i that form a transaction are either a read or a write. The set of items read by a transaction are called its *read set* and the set of items written by it are called its *write set*. Two transactions T_i and T_j are said to *conflict* if some action t_i of T_i and some action t_j of T_j access the same object and at least one of the actions is a write. Two situations are possible as presented in Fig. 8.5.

1.	The write set of one transaction intersects with the read set of another. The result of running the two transactions concurrently will clearly depend on whether the write is done first or the read. The conflict is called a *RW Conflict*.
2.	The write set of one transaction intersects with the write set of another. Again, the result of running the two transactions concurrently will depend on the order of the two writes. The conflict is called a *WW Conflict*.

Figure 8.5 Two conflict situations

A concurrency control mechanism must detect such conflicts and control them. Various concurrency control mechanisms are available. The mechanisms differ in the way they detect the conflict and the way they resolve it. We will consider concurrency control algorithms later in this chapter. First we discuss some examples of concurrency anomalies to highlight the need for concurrency control.

We have noted already that in the discussion that follows we will ignore many details of a transaction. For example, we will not be concerned with any computation other than the reads and the writes and whether the results of a read are being stored in a local variable or not.

We now discuss concurrency anomalies.

8.3 EXAMPLES OF CONCURRENCY ANOMALIES

There are three classical concurrency anomalies. These are lost updates, inconsistent retrievals and uncommitted dependency.

8.3.1 Lost Updates

Consider an example of two transactions *A* and *B* which simultaneously access an airline database wishing to reserve a number of seats on a flight. Let us assume that transaction *A* wishes to reserve five seats while transaction *B* wishes to reserve four seats. The reservation of seats involves booking the seats and then updating the number of seats available (N) on the flight. The two users read-in the present number of seats, store it in their local storage, and modify their local copy and write back resulting in the situation presented in Fig. 8.6.

Every row in Fig. 8.6 represents one time instance and one action of one of the transactions. We move down in time as we go down in the figure with time = 1 being the earliest time and time = 6 being the latest. No two actions of the two transactions are carried out at the same time. Also, we assume that these two transactions execute in isolation from all other transactions that might be running at the same time.

Transaction *A*	Time	Transaction *B*
Read *N*	1	–
–	2	Read *N*
N := *N* – 5	3	–
–	4	*N* := *N* – 4
Write *N*	5	–
–	6	Write *N*

Figure 8.6 An example of lost update anomaly

If the execution of the two transactions were to take place as shown in Fig. 8.6, the update by transaction *A* is lost although both users read the number of seats, get the bookings confirmed and write the updated number back to the database. This is called the *lost update anomaly,* since the effects of one of the transactions were permanently lost.

8.3.2 Inconsistent Retrievals

Consider another example of two transactions *A* and *B* accessing a department database simultaneously. Transaction *A* is updating the database to give all employees in the department a 3% raise in their salary while transaction *B* is computing the total salary bill of the department as shown in Fig. 8.7. The two transactions interfere since the total salary bill changes as the transaction *A* updates the employee records. The total salary retrieved by transaction *B* may be a sum of some salaries before the raise and others after the raise. This could not be considered an acceptable value of the total salary but the value before the raise or the value after the raise is acceptable.

Transaction *A*	Time	Transaction *B*
Read employee 100	1	–
–	2	Sum = 0.0
Update salary	3	–
–	4	Read employee 100
Write employee 100	5	–
–	6	Sum = sum + salary
Read employee 101	7	–
Update salary	8	–
Write employee 101	9	–
–	10	Read employee 101
–	11	Sum = sum + salary
etc.	–	etc.

Figure 8.7 An example of inconsistent retrieval

The problem illustrated in Fig. 8.7 is called the *inconsistent retrieval anomaly*. During the execution of a transaction, therefore, changes made by another transaction that has not yet committed should not be visible since that data may not be consistent.

8.3.3 Uncommitted Dependency

Consider an example of two transactions *A* and *B* running on the system as shown in Fig. 8.8.

Transaction *A*	Time	Transaction *B*
–	1	Read *Q*
–	2	Update *Q*
–	3	Write *Q*
Read *Q*	4	–
–	5	Read *R*
Update *Q*	6	–
Write *Q*	7	–
–	8	Failure (rollback)
–	9	–

Figure 8.8 An example of uncommitted dependency

Transaction *A* has read the value of *Q* that was updated by transaction *B* but was never committed. The result of transaction *A* writing *Q* therefore will lead to an inconsistent state of the database. Also if transaction *A* does not write *Q* but only reads it, it would be using a value of *Q* which never really existed. Yet another situation would occur if the rollback happened after *Q* was written by transaction *A*. The rollback would restore the old value of *Q* and therefore lead to the loss of updated *Q* by transaction *A*. This is called the *uncommitted dependency anomaly*.

We will not discuss the problem of uncommitted dependency any further since we assume that the recovery algorithms presented in the chapter will ensure that transaction *A* is also rolled back as transaction *B*. The most serious problem facing concurrency control is that of the lost update. The commonly suggested solutions are

1. Once transaction *A* reads a record *Q* for an update, no other transaction is allowed to read it until the transaction *A* update is completed. This is usually called *locking*.
2. Both transactions *A* and *B* are allowed to read the same record *Q* but once *A* has updated the record, *B* is not allowed to update it as well since *B* would now be updating an old copy of record *Q*. *B* must therefore read the record again and then perform the update.
3. Although both transactions *A* and *B* are allowed to read the same record *Q*, *A* is not allowed to update the record because another transaction (transaction *B*) has the old value of the record.
4. Segregate the database into several parts and schedule concurrent transactions such that each transaction uses a different portion. There is thus no conflict and the database stays consistent. Often this is not feasible since most databases have some parts, called *hot spots,* which most transactions want to access.

These are in fact the major concurrency control techniques. We discuss some of them in detail later in this chapter. We first need to consider the concept of *serializability* which deals with correct execution of transactions concurrently.

8.4 SCHEDULES

The concept of a transaction has already been discussed. A transaction must have the ACID properties of atomicity, consistency, isolation and durability. We have also noted earlier that each transaction normally consists of a number of steps or actions including reads and writes; an example is shown in Fig. 8.4.

When two or more transactions are running concurrently, the steps of the transactions would normally be *interleaved* because if one transaction is carrying out some I/O there is no reason for all the other transactions to wait for it to finish. Interleaving therefore improves the response time of the database system. The interleaved execution of transactions is decided by the database system concurrency control software called the *scheduler* which receives a stream of user requests that arise from the active transactions.

We now define a schedule.

Definition—Schedule
A particular sequencing (usually interleaved) of the actions (including reading and writing) of a set of transactions is called a schedule (or history). The order of actions of every transaction in a schedule must always be the same as they appear in the transaction.

We have discussed earlier that each transaction maintains the integrity of the database. It therefore follows that a serial execution of a number of transactions will also maintain the integrity of the database.

The serial schedule is defined as follows.

Definition—Serial Schedule
A serial schedule is a schedule in which all the operations of one transaction are completed before another transaction begins (that is, there is no interleaving).

For performance reasons, serial schedules are of little interest but they are useful as a measure of correctness because a serial schedule is always correct given our basic assumption that each transaction maintains the integrity of the database. That is what we will use them for.

We will soon present some interleaved schedules, for example, in Figs 8.11, 8.12 and 8.13. A serial transaction is presented in Fig. 8.10. We will show more different schedules of two transactions later in this chapter.

It is the responsibility of the scheduler to maintain database integrity while allowing maximum concurrency. The scheduler must make sure that only correct schedules are allowed. To have a mechanism or a set of conditions that define a correct schedule is unfortunately not possible since it is always very complex to define a consistent database. The assertions defining a consistent database (called *integrity constraints* or *consistency constraints*) could well be as complex as the database itself and checking these assertions, assuming one could explicitly enumerate them after each transaction commits, would not be practical. The

simplest approach to maintaining database consistency is based on the notion of each transaction being correct by itself transforming a consistent database to another consistent database and a schedule of concurrent executions being able to preserve their correctness.

8.5 SCHEDULES AND RECOVERABILITY

Normally we assume that if a transaction has been aborted or has failed for some reason then the transaction may be rolled back and all its action may be undone leaving the database in a condition as if the transaction never existed. In some situations aborting a transaction so that all its actions are undone may not be possible, perhaps because it had modified some data item which was subsequently modified by another transaction and that transaction has already committed. Such a schedule is called *unrecoverable* and hence should not be permitted.

On the other hand, *recoverable* transactions are such that another transaction is not able to modify contents that it has modified and commit. Recoverability is therefore used to enhance concurrency and maintain consistency when transactions abort.

We first define recoverability.

Definition—Recoverability
A schedule is said to be recoverable if it does not allow a transaction A to commit until every other transaction B that wrote values that were read by A has committed.

Therefore a schedule is recoverable as long as transaction *A* does not commit until every transaction that wrote values read by *A* has committed. In other words, a schedule is recoverable if it never requires a committed transaction to be rolled back.

Let us consider the example given in Fig. 8.8 where we discussed uncommitted dependency. A slightly modified version of Fig. 8.8 is reproduced in Fig. 8.9. Uncommitted dependency actually dealt with the problem of recoverability and as we show in Fig. 8.8 that transaction *A* commits before transaction *B* carries out a *dirty write* and *aborts*. Concurrency control therefore must ensure that all tractions are recoverable and situations like that shown in Fig. 8.9 are not permitted.

Transaction *A*	Time	Transaction *B*
–	1	Read *Q*
–	2	Update *Q*
–	3	Write *Q*
Read *Q*	4	–
–	5	Read *R*
Update *Q*	6	–
Write *Q*	7	–
Commit (*A*)	8	
–	9	Failure (rollback)
–	10	–

Figure 8.9 An example of uncommitted dependency

8.6 SERIALIZABILITY

As noted above, a serial schedule is always correct since serial transactions do not depend on each other. As noted earlier, we further assume that each transaction when run in isolation transforms a consistent database into a new consistent state and therefore a set of transactions executed one after another (i.e. serially) must also be correct. A database may not be consistent during the execution of a transaction, but it is assumed the database is consistent at the end of each transaction. Although using only serial schedules to ensure consistency is a possibility, it is not a realistic approach as noted earlier, since one transaction may be waiting for some input/output from secondary storage or from the user while the CPU remains idle, wasting a valuable resource. Another transaction could have been running while the first transaction was waiting and this would obviously improve the system efficiency. In some cases, a complex transaction if run serially could monopolize the CPU for a long period resulting in all other transactions waiting for it to finish. We therefore need to investigate schedules that are not serial but result in database consistency being maintained. However, since all serial schedules are correct, interleaved schedules that are equivalent to them must also be considered correct. There are in fact $n!$ different serial schedules for any set of n transactions. Note that not all serial schedules of the same set of transactions result in the same consistent state of the database. For example, an airline seat reservation system may result in different allocations of seats for different serial schedules although each schedule will ensure that no seat is sold twice and no request is denied if there is a free seat available on a desired flight. However, one serial schedule could result in 200 passengers on the plane while another could result in 202, as shown below.

Let us consider an airline reservation system. Let there be 12 seats available on flight IA32 at some stage. Three requests arrive, viz., transaction T_1 for 3 seats, transaction T_2 for 5 seats and transaction T_3 for 7 seats. Any of the six serial schedules of the three transactions presented in Fig. 8.10 are correct.

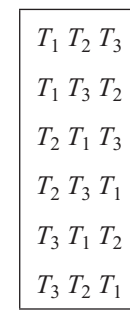

Figure 8.10 Six possible different schedules for three transactions

If the transactions are executed in the order $\{T_1, T_2, T_3\}$ then we allocate 8 seats but cannot meet the third request since there are only 4 seats left after allocating seats for the first two transactions. If the transactions are executed in the order $\{T_2, T_3, T_1\}$ then we are able to allocate all the 12 remaining seats but the request of transaction T_1 cannot be met. If the transactions are instead executed in the order $\{T_3, T_1, T_2\}$ then we allocate 10 seats but are unable to meet the request of transaction T_2 for 5 seats. In all there are 3! (that is 6) different serial schedules possible. The remaining three are $\{T_1, T_3, T_2\}$, $\{T_2, T_1, T_3\}$ and $\{T_3, T_2, T_1\}$. They lead to 10, 8 and 12 seats being sold respectively. All the above serial schedules must be considered correct although they produce different results and any one of the three possible results may be obtained as a result of running these transactions.

We have seen that some sets of transactions when executed concurrently without any controls may lead to problems, for example, the lost update anomaly. Of course, any set of transactions can be executed concurrently without leading to any difficulties if the read and write sets of these transactions do not intersect.

We want to discuss sets of transactions that do interfere with each other. To discuss correct execution of such transactions, we now define the concept of *final state serializability* which is a requirement when more than one transaction is being executed concurrently.

8.6.1 Final State Serializability

Before we define final state serializability, we define final state equivalence which is required in defining final state serializability.

Definition—Final State Equivalence
Two schedules S1 and S2 are called final state equivalent if both include the same set of operations and result in the same final state for any given initial state.

Let T be a set of n transactions $T_1, T_2, \ldots T_n$. If the n transactions are executed serially (call this execution S), we assume they terminate properly and leave the database in a consistent state.

Definition—Final State Serializability
A concurrent execution of n transactions $T_1, T_2, \ldots T_n$ (call this execution S) is called final state serializable (FSR) if the execution is final state equivalent to a serial execution of the n transactions.

The final state serializable schedule may be equivalent to any serial schedule. For example, Fig. 8.9 shows six possible different schedules of any three transactions. Therefore the concurrent execution C always produces exactly the same effect on the database as S does, where S is any serial execution of T, not necessarily the order $T_1, T_2, \ldots T_n$. As shown in a serializable schedule in Fig. 8.11, if a transaction T_i writes a data item A in the interleaved schedule C before another transaction T_j reads or writes the same data item, the schedule C must be equivalent to a serial schedule in which T_i appears before T_j. Therefore, in the interleaved transaction, T_i appears logically before T_j; the order is the same as in the equivalent serial schedule. The concept of serializability defined here is sometimes called *final state serializability*; other forms of serializability also exist. Final state serializability has sometimes been criticized for being too strict while at other times it has been criticized for being not strict enough!

If a schedule of transactions is not serializable, we may be able to overcome the concurrency problems by modifying the schedule so that it is serializable. The modifications are made by the database scheduler.

We now consider a number of examples of possible schedules of two transactions running concurrently. Consider a situation where a couple has three accounts (A, B, C) with a bank. The husband and the wife maintain separate personal savings accounts while they also maintain a joint loan account on their house. A is the husband's account, B is the wife's account and C is the housing loan. Each month a payment of $500 is made to account C. To make the payment on this occasion, the couple walks to two adjoining automatic teller machines and the husband transfers $200 from account A to C (Transaction 1) while the wife on the other machine transfers $300 from account B to C (Transaction 2). Several different schedules for these two transactions are possible. We present the following four schedules for consideration.

A careful look at these schedules will show that the third schedule (Fig. 8.12) is not serializable while the other three are.

Another Notation for Schedules

Although we will continue to use tabular representations for presenting schedules, a much more compact form is often used. In the tabular form, each schedule is ordered by time with time starting at the top increasing down to the bottom of the table. In the linear representation that we now describe, time goes from the left to the right.

A transaction is a sequence of steps. From the sequence, this representation first extracts the read and write steps from the schedules since they are the only steps that concern us here; ignoring all other actions. After extracting the reads and writes, these actions are represented by symbols like $r_1(A)$ for reading A by transaction number 1. Let us now look at two transactions as follows:

$$\text{Transaction 1} = r_1(A)w_1(A)r_1(C)w_1(C)c_1$$
$$\text{Transaction 2} = r_2(B)w_2(B)r_2(C)w_2(C)c_2$$

Note that we have added symbols c_1 and c_2 to indicate that transactions 1 and 2 commit respectively.

The schedule in Fig. 8.12 may then be represented as

$$\text{Schedule in Fig. 8.12} = r_1(A)w_1(A)\ r_2(B)w_2(B)\ r_1(C)w_1(C)r_2(C)w_2(C)$$

This list representation of the schedule indicates that a read and a write of item A by Transaction 1 takes place followed by a read and write of item B by Transaction 2 and so on.

We will continue to use the tabular representation for the moment since it is easier to see what is in the schedule at a glance, but we may use the compact notation later when convenient.

We first present a serial schedule in Fig. 8.11.

Transaction 1	Time	Transaction 2
Read A	1	–
$A := A - 200$	2	–
Write A	3	–
Read C	4	–
$C := C + 200$	5	–
Write C	6	–
–	7	Read B
–	8	$B := B - 300$
–	9	Write B
–	10	Read C
–	11	$C := C + 300$
–	12	Write C

Figure 8.11 A serial schedule without interleaving

Clearly all the actions of Transaction 1 are carried out first followed by actions of Transaction 2. There is no interaction between the transactions. This is not so in the next schedule in which the actions of the two transactions are interleaved.

We present an interleaved serializable schedule in Fig. 8.12.

Transaction 1	Time	Transaction 2
Read *A*	1	–
A := *A* – 200	2	–
Write *A*	3	–
–	4	Read *B*
–	5	*B* := *B* – 300
–	6	Write *B*
Read *C*	7	–
C := *C* + 200	8	–
Write *C*	9	–
–	10	Read *C*
–	11	*C* := *C* + 300
–	12	Write *C*

Figure 8.12 An interleaved serializable schedule

Figure 8.13 shows another interleaved schedule that is not serializable because it includes a lost update.

Transaction 1	Time	Transaction 2
Read *A*	1	–
A := *A* – 200	2	–
Write *A*	3	–
Read *C*	4	–
–	5	Read *B*
–	6	*B* := *B* – 300
–	7	Write *B*
–	8	Read *C*
–	9	*C* := *C* + 300
–	10	Write *C*
C := *C* + 200	11	–
Write *C*	12	–

Figure 8.13 An interleaved nonserializable schedule

Figure 8.14 shows yet another serializable schedule that has taken care of the lost update by writing *C* before the second transaction reads and writes it.

Transaction 1	Time	Transaction 2
–	1	Read *B*
–	2	$B := B - 300$
–	3	Write *B*
Read *A*	4	–
$A := A - 200$	5	–
Write *A*	6	–
Read *C*	7	–
$C := C + 200$	8	–
Write *C*	9	–
–	10	Read *C*
–	11	$C := C + 300$
–	12	Write *C*

Figure 8.14 Another interleaved serializable schedule

It is therefore clear that many interleaved schedules would result in a consistent state while many others will not. All correct schedules of the two transactions T_1 and T_2 are serializable only if they are equivalent to either serial schedule (T_1, T_2) or (T_2, T_1).

8.6.2 Testing for FSR

There is no simple way to test for FSR. Conceptually testing for FSR involves looking at the schedule that we want to test and comparing it with each of the permutations of the *n* transactions that are in it. For example, if there are three transactions in a schedule then we may compare the schedule with each of the six serial schedules of the three transactions. If the schedule matches one of them then the schedule is correct according to FSR. However, as the number of transactions grows, the number of permutations grows exponentially and then comparison becomes quite inefficient.

8.6.3 View Serializability

The final state serializability requires that a schedule be equivalent to a serial schedule as both result in the same final state for a given initial state. In FSR we do not know what happens at every step of the schedule. Does each transaction read the same values as the other? It is not unreasonable to require that they do, since it will ensure that both transactions have the same view of the database.

Definition—View Equivalence
Two schedules S1 and S2 are called view equivalent if both comprise the same set of operations and result in the same final state for any given initial state and in addition each operation has the same semantics in both schedules.

Given the definition of view equivalence, we can now define view serializability.

Definition—View Serializability
A concurrent execution of n transactions T_1, T_2, … T_n (call this execution S) is called view serializable (VSR) if the execution is view equivalent to a serial execution of the n transactions.

View serializability is a stricter condition than FSR since in addition to the requirement of FSR view serializability require that each corresponding read operation in each schedule return the same values. We now describe conflict serializability which is a stricter condition still.

8.6.4 Testing for VSR

There is no simple algorithm available for testing VSR. We could of course follow the same brute force approach as we did for FSR and compare a given schedule with each permutation of the transactions in it for view equivalence but this approach is not very practical. For this reason VSR has limited practical use.

8.6.5 Conflict Serializability

It should now be clear that there are a number of notions of serializability. We have discussed final state serializability and view serializability but FSR and VSR have limited practical value since it is difficult to test whether a given schedule is in FSR or VSR. We therefore discuss another concept of serializability called conflict serializability. Conflict serializability has a number of useful theoretical and practical properties and is therefore more widely used.

The definition of conflict serializability uses the concept of conflict equivalence. Two schedules are defined to be conflict equivalent if they have the same set of operations and the same order of conflicting operations. We first define conflict.

Conflict serializability is a stronger requirement than requirements for FSR and VSR.

Definition—Conflict
Two data operations belonging to two different transactions in a schedule are in conflict if they access the same data item and at least one of them is a write.

Based on this concept of conflict, we may now define conflict equivalence and conflict serializability.

Definition—Conflict Equivalence
Two schedules are called conflict equivalent if they have the same set of operations and the same conflict relations.

Definition—Conflict Serializability
A concurrent execution of n transactions T_1, T_2, … T_n (call this execution S) is called conflict serializable if the execution is conflict equivalent to a serial execution of the n transactions.

Conflict serializability is a stronger condition than view serializability and therefore all schedules that satisfy the requirements of CSR also satisfy the requirements of VSR.

8.6.6 Testing for Conflict Serializability—The Precedence Graph

In contrast to testing FSR and VSR, it is easier to test for CSR.

We now present an algorithm for testing conflict serializability. The algorithm involves constructing a *precedence graph* (also called *conflict graph* or *serialization graph*). The precedence graph is a directed graph in which there is a node for each transaction in the schedule and an edge between T_i and T_j exists if any of the conflict operations given in Fig. 8.15 appear in the schedule.

1. T_i executes WRITE(X) before T_j executes READ(X), or
2. T_i executes READ(X) before T_j executes WRITE(X), or
3. T_i executes WRITE(X) before T_j executes WRITE(X).

Figure 8.15 List of possible conflicts in two transactions

A schedule is serializable only if a cycle is not found in the precedence graph. We give an example of a precedence graph in Fig. 8.16. Each edge has a label which indicates the name of the object involved in the conflict. This graph shows that the transactions carry out one of the conditions 1–3 between T_1 and T_3, T_2 and T_3, T_1 and T_5 as well as T_3 and T_4, T_4 and T_6 and T_5 and T_6.

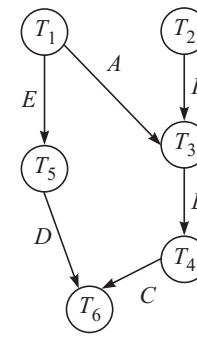

Figure 8.16 An example of a precedence graph

The precedence graph in Fig. 8.16 has no cycle and therefore the schedule shown in this figure is serializable. The reader is encouraged to write down a schedule that results in the given precedence graph.

It should be noted that all objects A, B, C, D, E and F are distinct objects because if say C was the same as A then there must have been edges between T_1 and T_4, and T_1 and T_6 as well since there would be additional conflicts because of transactions T_4 and T_6 operating on the object A.

Assuming a simple concurrent execution of two transactions T_i and T_j, it is clear that the graph can only have a cycle if there is an edge between T_i and T_j as well as an edge between T_j and T_i. For this to happen, one of the above conditions must be met for an object X followed by the schedule meeting one of the conditions given in Fig. 8.17 for an object Y, where the objects X and Y need not be distinct.

4. T_j executes WRITE(Y) before T_i executes READ(Y), or
5. T_j executes READ(Y) before T_i executes WRITE(Y), or
6. T_j executes WRITE(Y) before T_i executes WRITE(Y).

Figure 8.17 List of possible conflicts in two transactions

These two sets of conditions provide a total of nine different combinations of conditions which could lead to a cycle. It can be shown that none of these combinations are possible if 2PL discussed later in Section 8.8.2 is used. For example, assume the two conditions given in Fig. 8.18 have been satisfied by a schedule.

7. T_i executes WRITE(X) before T_j executes READ(X),
8. T_j executes WRITE(Y) before T_i executes WRITE(Y).

Figure 8.18 List of possible conflicts in two transactions

It can be shown that none of the eight combinations of conditions are possible if 2PL is used.

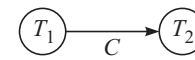

Figure 8.19 Precedence graph for the schedule in Fig. 8.11

It is interesting to draw precedence graphs for some schedules that we have presented. The schedule in Figs 8.11 and 8.12 are like that shown in Fig. 8.19.

The precedence graph for the schedule in Fig. 8.13 is presented in Fig. 8.20. Note that this graph has a cycle in it and is therefore not serializable. The precedence graph for the schedule in Fig. 8.13 is similar to that given in Fig. 8.20 and that schedule is also not serializable.

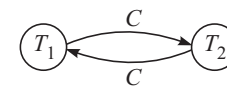

Figure 8.20 Precedence graph for the schedule in Fig. 8.13

Several versions of 2PL have been suggested. Two commonly used versions are called *static* two-phase locking and *dynamic* two-phase locking. The static technique basically involves locking all items of information needed by a transaction before the first step of the transaction is executed and unlocking them all at the end of the transaction. The dynamic scheme locks items of information needed by a transaction only immediately before using them. All locks are released at the end of the transaction.

8.7 HIERARCHY OF SERIALIZABLE SCHEDULES

We have explained the concepts of FSR, VSR and CSR and relationships between them. We illustrate them in Fig. 8.21.

Figure 8.21 shows that serial schedules are the most restrictive serializable schedules, since no interleaving of transactions is permitted in serial schedules. Amongst the serializability requirements we have discussed earlier, CSR is the next most restrictive, followed by VSR and FSR is the weakest requirement of serializability amongst the techniques discussed. There are a number of other serializability techniques presented in the literature that we have not discussed.

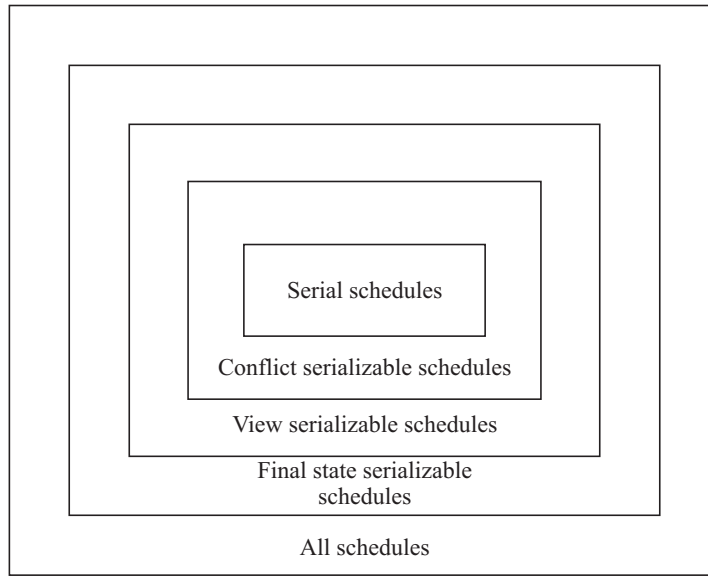

Figure 8.21 Hierarchy of serializable schedules

8.8 CONCURRENCY CONTROL AND ENFORCING SERIALIZABILITY

As noted earlier, a schedule of a set of transactions is serializable if computationally its effect is equal to the effect of some serial execution of the transactions. A DBMS must ensure that only serializable schedules are allowed and the durability property of transactions is also ensured. Therefore, any transaction that commits must not be allowed to be undone.

One way to enforce serializability is to insist that when two transactions are executed, one of the transactions is assumed to be older than the other. Now the only schedules that are accepted are those in which data items that are common to the two transactions are seen by the junior transaction only after the older transaction has written them back. The DBMS module that controls which schedules are permitted is called the *scheduler*.

The three basic techniques used in concurrency control (locking, timestamping and optimistic concurrency control) enforce this in somewhat different ways. The only schedules that these techniques allow are those that are serializable. We now discuss locking followed by these concurrency control techniques.

8.8.1 Locking

Locking is a common technique by which a database may synchronize execution of concurrent transactions. Using locking, a transaction can obtain exclusive or shareable access rights (called *locks*) to an object. If the access provided by the lock is shareable, the lock is often called a *shared lock* (sometimes called a *read* lock). On the other hand, if the access is exclusive, the lock is called an *exclusive lock* (sometimes called a *write*

lock). If a number of transactions need to read an object and none of them wishes to write that object, a shared lock would be the most appropriate. Locks may be acquired or released for data items by making requests to the scheduler. The locks are granted by database system software called the *lock manager.*

Of course, if any of the transactions were to write an object, the transaction must acquire an exclusive lock on the object to avoid any concurrency anomalies that we have discussed earlier. A number of locking protocols are possible. Each protocol consists of the following:

1. A set of locking types.
2. A set of rules indicating what locks can be granted concurrently.
3. A set of rules that transactions must follow when acquiring and releasing locks.

We will use the simplest approach that uses shared and exclusive locks, where a transaction would set a read lock on the data item that it reads and an exclusive lock on the data item that it needs to update. As the names indicate, a transaction may obtain a shared lock on a data item even if another transaction is holding a shared lock on the same data item at the same time. Of course, a transaction cannot obtain a shared or exclusive lock on a data item if another transaction is holding an exclusive lock on it. The shared lock may be upgraded to an exclusive lock (assuming no other transaction is holding a shared lock on the same data item at that time) for items that the transaction wishes to write. This technique is sometimes also called *blocking* because if another transaction requests access to an exclusively locked item, the lock request is denied and the requesting transaction is blocked.

A transaction can hold a lock on one or more items of information. The transaction must unlock these data item(s) before the transaction commits. As noted earlier, the locks are granted by the database system software called the *lock manager* which maintains information on all locks that are active in a lock table and controls access to the locks. The lock table contains a variety of information. For example, the identifier for each locked object, the number of locks on each object and their type and identifiers of transactions holding locks. As noted earlier, several modes of locks may be available. For example, in shared mode, a transaction can read the locked data item but cannot update it. In an exclusive lock mode, a transaction has exclusive access to the locked data item and no other transaction is allowed access to it until the lock is released. A transaction is allowed to update the data item only as long as an exclusive lock is held on it.

If transactions running concurrently are allowed to acquire and release locks as data items are read and updated, there is a danger that incorrect results may be produced in spite of the locks. Consider the example in Fig. 8.22.

In Fig. 8.22, exclusive lock (called XL) is a request for an exclusive lock and UL is releasing the lock (sometimes called unlock). SL is a request for a shared lock. In the example shown in Fig. 8.22, the result displayed by Transaction 2 is incorrect because it was able to read *B* before the update but read *A* only after the update. The above schedule is therefore not serializable. As discussed earlier, the problem with this schedule is inconsistent retrieval.

The problem arises because the above schedule involves two RW-conflicts. The first one involves *A* in which Transaction 1 logically appears before Transaction 2 since Transaction 1 reads *A* before Transaction 2. The second conflict involves *B* in which Transaction 2 logically appears before Transaction 1 since Transaction 2 reads *B* before Transaction 1. Therefore, the schedule is not serializable.

Transaction 1	Time	Transaction 2
XL(A)	1	–
Read(A)	2	–
$A := A - 50$	3	–
–	4	SL(B)
Write(A)	5	–
–	6	Read(B)
UL(A)	7	–
–	8	UL(B)
–	9	SL(A)
XL(B)	10	–
–	11	Read(A)
Read(B)	12	–
–	13	UL(A)
$B := B + 50$	14	–
–	15	Display ($A + B$)
Write (B)	16	–
UL(B)	17	–

Figure 8.22 Unserializable schedule using locks

Such problems may be overcome by using a technique called the two-phase locking (2PL), which we now discuss.

8.8.2 Two-Phase Locking (2PL)

To overcome the problem illustrated in Fig. 8.22 a two-phase locking (2PL) scheme is used. 2PL is the most commonly used concurrency control technique in commercial databases. In 2PL, a transaction cannot request a new lock after releasing a lock. The two-phase locking protocol involves the following two phases:

1. *Growing Phase* (*Locking* phase)—During this phase, locks may be acquired but not released.
2. *Shrinking Phase* (*Unlocking* phase)—During this phase, following the growing phase, locks may be released but not acquired. All locks are released when the transaction is completed.

In a summary, the main feature of two-phase locking is that conflicts are checked at the beginning of the execution and resolved by waiting. This leads to safe interleaving of transactions.

Using the 2PL method we can modify the schedule presented in Fig. 8.22. The modified schedule is given in Fig. 8.23. This schedule acquires all locks before any lock is released.

Unfortunately, although the above schedule would not lead to incorrect results, it has another difficulty, i.e., both the transactions are blocked waiting for each other. Transaction 1 in Fig. 8.23 is waiting (at Step 7) for Transaction 2 to unlock B while Transaction 2 cannot do so until it obtains a lock on A (at Step 9). A classical *deadlock* situation has occurred! Deadlocks arise because of circular wait conditions involving two or more

Transaction 1	Time	Transaction 2
XL(A)	1	–
Read(A)	2	–
$A := A - 50$	3	–
–	4	SL(B)
Write(A)	5	–
–	6	Read(B)
XL(B)	7	–
UL(A)	8	–
–	9	SL(A)
–	10	UL(B)
–	11	Read(A)
Read(B)	12	–
–	13	UL(A)
$B := B + 50$	14	–
–	15	Display ($A + B$)
Write (B)	16	–
UL(B)	17	–

Figure 8.23 A schedule using two-phase locking

transactions as above. A system that does not allow deadlocks to occur is called *deadlock free*. In two-phase locking, we need a deadlock detection mechanism and a scheme for resolving the deadlock once a deadlock has occurred. We will look at such techniques in Section 8.9.

The attraction of the 2PL algorithm derives from a theorem which proves that 2PL always leads to serializable schedules that are equivalent to serial schedules in the order in which each transaction acquires its last lock. This is a sufficient condition for serializability although it is not necessary.

We now discuss how deadlocks can be detected, prevented and resolved.

8.9 DEADLOCKS

We first define a deadlock.

Definition—Deadlock
A deadlock can be defined as a situation in which each transaction in a set of two or more concurrently executing transactions is blocked circularly waiting for another transaction in the set.

Therefore, none of the transactions will become unblocked unless there is some external intervention.

As noted earlier, the locks are granted and managed by the database system software called the lock manager and the lock manager maintains a lock table to manage the locks. This table maintains a variety of information about each lock including how many locks are held on each object, assuming these are shared locks, and which locks are held by which transaction. A queue of lock requests that are waiting is also maintained.

One experimental study has suggested that deadlocks are not very common. Of course the instance of deadlocks goes up as more and more transactions try to access the same object. For example, booking tickets on a train or a flight at the time of holidays like Deepawali. Such objects are called *hot spots*. Normally though, it has been found that less than 1% of the transactions are involved in deadlocks.

Deadlock detection requires a technique to detect a deadlock. Usually deadlocks are detected by building and analyzing a waits-for graph (WFG) as shown in Fig. 8.24. Deadlock detection may take place every time a transaction's lock request is not granted. This is called *continuous detection*. On the other hand one may prefer a *periodic detection* scheme, for example, once every second, although results of experiments seem to indicate that the continuous scheme is more efficient.

8.9.1 Wait-For Graph (WFG)

A wait-for graph is presented in Fig. 8.24. When a transaction T_i requests a data item that is currently being held by T_j, then the edge T_i to T_j is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.

The system in Fig. 8.24 is in a deadlock state because the wait-for graph given in the figure has a cycle since T_1 is waiting for T_2, which is waiting for T_3 and T_4 while T_3 is waiting for T_5, which in turn is waiting for T_2 which is already waiting. The letters on the edges in the figure indicate the names of the objects that transactions are waiting for. For example, T_1 is waiting for T_2 for object b. Therefore, there is a cyclical wait between transactions T_2, T_3 and T_5.

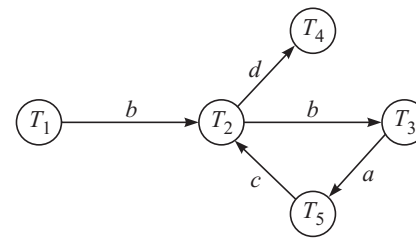

Figure 8.24 An example of a wait-for graph (WFG)

As noted earlier, a deadlock can only be detected by periodically checking the WFG for cycles. We will soon discuss how to resolve deadlocks once a deadlock has been detected.

A deadlock is clearly undesirable but deadlocks are often unavoidable in concurrency controls based on locking. Two-phase locking is not deadlock free. We therefore must deal with deadlocks when they occur. There are several aspects of dealing with deadlocks, viz., one should prevent them if possible (deadlock *prevention*), detect them when they occur (deadlock *detection*) and resolve them when a deadlock has been detected (deadlock *resolution*). Deadlocks may involve two, three or more transactions. To resolve deadlocks, one must keep track of what transactions are waiting and which transaction they are waiting for.

Once a deadlock is detected, one of the transactions in the deadlock must be selected and rolled back (such a transaction is called a *victim*) thereby releasing all the locks that the transaction held and thus breaking the deadlock. We first discuss deadlock prevention followed by discussion on deadlock detection and resolution.

8.9.2 Deadlock Prevention

Deadlocks occur when some transactions wait for other transactions to release a resource and the wait is circular (for example, as already discussed, in Fig. 8.24, T_1 is waiting for T_2, which is waiting for T_3 and T_4 while T_3 is waiting for T_5, which in turn is waiting for T_2 which is already waiting). Deadlock prevention does not involve deadlock detection. Instead deadlocks are prevented by ensuring that circular waits do not occur. This can be done either by defining an order on who may wait for whom or by eliminating all waiting. We first discuss two techniques that define an ordering on transactions.

Wait-die Algorithm

When a conflict occurs between T_1 and T_2 (T_1 being the older transaction), if T_1 possesses a lock then T_2 (the younger one) is not allowed to wait for the resource that T_1 has a lock on. T_2 must roll back and restart. If, however, T_2 possessed the lock at conflict, the senior transaction is allowed to wait. The technique therefore avoids cyclic waits and generally avoids *starvations* (a transaction's inability to secure resources that it needs). An older transaction may however find that it has to wait for every resource that it needs, and it may take a while to complete.

Wound-die Algorithm

To overcome the possibility of a senior transaction having to wait for every item of data that it needs, the wound-die scheme allows a senior transaction to immediately acquire a data item that it needs, even if it is being controlled by a younger transaction by having a lock on it. The younger transaction is then restarted. A younger transaction waiting for a lock that is held by a senior transaction however is allowed to wait.

Immediate Restart

In this scheme, no waiting is allowed. If a transaction requests a lock on a data item that is being held by another transaction (younger or older), the requesting transaction is restarted immediately. This scheme can lead to starvation if a transaction requires several popular data items since every time the transaction restarts and seeks locks on a number of popular items, it is likely to find that one or more of those data items have been locked by other transactions resulting in it being rolled back and restarted.

We now illustrate these techniques in Fig. 8.25. We assume that transactions arrive in the sequence 1, 2 and 3. Therefore, *T*1 is the oldest and *T*3 is the youngest.

The figure shows that the three techniques result in different actions. For example, when a senior transaction wishes to lock an object that has a lock by a junior transaction then wait-die allows it to wait, wound-die aborts the younger transaction while immediate restart aborts the senior transaction.

8.9.3 Deadlock Detection and Resolution

Once a deadlock has been detected using the WFG as shown in Fig. 8.24, it must be resolved. The most common resolution technique requires that one of the transactions in the waits-for graph be selected as a *victim* and be rolled back and restarted. The victim may be selected on the basis of any of the following:

1. *Randomly*—A transaction from the cycle is picked randomly.
2. *Last blocked*—The transaction that blocked most recently.

Trans. *T*1	Trans. *T*2	Trans. *T*3	Comments	Wait-die	Wound-die	Immediate Restart
XL(*A*)	1	–	*A* locked			
	XL(*C*)	–	*C* locked			
UL(*A*)	3	–	*A* unlocked			
XL(*C*)	4	–	Conflict	*T*1 waits for *C*	*T*1 locks *C*; *T*2 aborted	*T*1 aborted
–	5	XL(*B*)	*B* locked			
–	XL(*A*)		*A* locked			
–	XL(*B*)		Conflict	*T*2 waits for *B*	*T*2 locks *B*; *T*3 Aborted	*T*2 aborted
–	8	XL(*C*)	Conflict	*T*3 aborted	*T*3 waits	*T*3 aborted

Figure 8.25 Example illustrating deadlock prevention techniques

3. *Youngest*—Since this transaction is likely to have done the least amount of work.
4. *Minimum work*—The transaction that has written the least data back to the database, since all the data written must be undone in the rollback.
5. *Least impact*—The transaction that is likely to affect the least number of other transactions, that is, the transaction whose rollback will lead to least other rollbacks (in cascade rollback).
6. *Fewest locks*—The transaction that has the smallest number of locks.

It has been found that a technique that selects the victim with the fewest locks to resolve a deadlock provides the best performance. Also, it has been suggested that if deadlock prevention with immediate restart is to be used, database performance suffers. Furthermore, it has been suggested that locking smaller objects improves performance because chances of a deadlock occurring are reduced.

8.10 LOCK GRANULARITY

Lock granularity was first studied by Jim Gary and his colleagues. Our discussion is based on their 1976 paper cited at the end of this chapter.

In addition to selecting a locking mechanism, the nature and size of the individual data item that may be locked must be decided upon. A lockable data item could be as large as a table or as small as a column value of a row. The lockable item could also be of a size in between. For example, a page of a relation or a single row. The larger the size of the items that are locked, the smaller is the concurrency that is possible, because when a transaction wants to access one or more rows of a table, it is allowed to lock the whole table and another transaction wanting to access some other rows of the same table is denied access. On the other hand, coarse granularity reduces the overhead costs of locking since the lock table that maintains information on the locks is smaller since large objects are being locked and the number of transactions holding locks is smaller. Fine granularity of the locks involving individual row locking or locking of even smaller items allows greater

concurrency at the cost of higher overheads. Therefore, there is a trade-off between the level of concurrency and overhead costs.

We now define lock granularity.

Definition—Lock Granularity
The size of the data unit that is locked is called lock granularity. The granularity is fine if the data unit is small and coarse if it is large.

Typically the finest granularity involves locking a row at a time. A coarser granularity is locking a block or a page of the table. A coarse granularity involves locking the whole table.

8.10.1 Fine vs Coarse Granularity

The following points are worth noting:

1. Fine granularity reduces contention and therefore increases concurrency since granularity is fine, and therefore each transaction is locking only a small part of the database. This helps performance but is suitable only for simple transactions which access only a few rows of a table. Fine granularity may not be suitable for complex transactions which access a large number of rows because such a transaction would need to acquire many locks to get its work done.
2. Since fine granularity is locking only small data units, the number of locks at any one time is larger than it would be with a coarser granularity since each transaction is likely to hold many locks. Therefore, lock management overhead including space required by the lock manager to maintain information about locks is higher.
3. Coarse granularity results in fewer locks and lower overheads of maintaining the locks but usually leads to lower concurrency since each transaction is holding on to a larger chunk of the database. This is particularly true if there are some hot spots in the database and transactions are allowed to lock large data units. Coarse granularity is more suitable for complex transactions that access many rows at a time.

Because of this tension between the benefits and disadvantages of fine and coarse granularity, both types of units of locks or even more than two levels of granularity may be offered by a database system. We discuss this approach now.

8.11 MULTIPLE GRANULARITY AND INTENTION LOCKING

We earlier suggested that a database could use several different granularities not just within the database but also within an application, perhaps depending on the nature of each transaction. Multiple granularity locking solves some problems but leads to other problems because now locked items with different granularities must be compared to check conflicts. For example, if one user has been allocated a fine granularity lock, for instance, on one row, and another user requests to lock the whole table then the system needs to efficiently check that there is no conflict between the fine granularity lock and the coarse granularity lock.

8.11.1 Hierarchy of Lock Units

Lock units may be organized in a hierarchy, starting from the whole database going down to a single row or even a single cell as follows:

1. The database
2. Database areas
3. Tables
4. Blocks or Pages
5. Rows

This hierarchy may be considered as a tree with the whole database as its root and other data units as its descendents. The database then is a superset of areas which are supersets of tables. Tables contain blocks or pages and each page contains rows.

Each item in the hierarchy can be locked. It should be noted that as a transaction acquires an exclusive lock on an item then it also acquires exclusive locks on all its descendents.

8.11.2 Intention Locks

To overcome the problem of checking locks in a multiple granularity locking system, a new type of lock called the *intention lock* is used. When a fine granularity lock is taken, a weak coarse granularity lock, the intention lock, is also taken. Hence, an intention lock is used to lock all ancestors of a data unit to be locked either in a shared or exclusive mode. An intention lock is a flag to the lock manager that locking is being done at a finer level thus preventing conflicts.

We now show how an intention lock works.

1. When a transaction *A* wishes to obtain a fine granularity lock on a table *P*, it must first request an intention lock on *P*. This essentially flags to the system lock manager that *A* is going to request a fine granularity lock on table *P*.
2. The intention lock requested is an intention shared (IS) lock if the transaction wishes to request a shared lock on say a row in table *P*. If *A* needs an exclusive lock on a row in *P* then it requests an intention exclusive (IX) lock on the table *P*.
3. A special case of intention lock called shared intention lock (SIX) is available for transactions that read a lot of rows of a table but change only a small number. This resolves the problem of exclusively locking the whole table by granting a shared lock as well as an exclusive lock.

The conflict matrix for the locks is given in Fig. 8.26. It shows the following:

- *S Lock*—If transaction *A* has acquired a share lock on a table *P* then another transaction is allowed to lock descendents of data unit locked by *A* in *S* or IS mode.
- *X Lock*—If transaction *A* has been granted an exclusive lock on a table *P* then no other transaction is allowed to lock descendents.
- *IS Lock*—If transaction *A* has acquired an intention share IS lock on a table *P* then another transaction is allowed to lock descendents of the data unit locked by *A* in S, IS or IX mode.

- *IX*—If transaction *A* has been granted an intention exclusive IX lock on a table *P* then another transaction is allowed to lock descendents in IS or IX mode.
- *SIX Lock*—If transaction *A* has been granted an intention share and exclusive lock SIX on a table *P* then another transaction is allowed to lock descendents in IS mode.

	Requested Lock Type					
		S	X	IS	IX	SIX
Lock Type Granted	S	Y	N	Y	N	N
	X	N	N	N	N	N
	IS	Y	N	Y	Y	Y
	IX	N	N	Y	Y	N
	SIX	N	N	Y	N	N

Figure 8.26 Conflict matrix using intention locking

Locks are requested from the highest level to the lowest level and released from the lowest to the highest. Therefore, a higher level lock cannot be released without a lower level lock being released first.

The table in Fig. 8.27 illustrates how request sequences would work.

Locking Request	Database	Area	File	Block	Row
Lock row *R* for read	IS	IS	IS	IS	S
Lock row *R* for write-exclusive access	IX	IX	IX	IX	X
Lock a file *P* for read and write access	IX	IX	X		
Lock a file *P* for complete scan and occasional update	IX	IX	SIX		

Figure 8.27 Request sequence using intention locking

Clearly intention locks involve additional locks that must be managed by the system but it is possible to achieve better performance since intention locks allow other transactions much more freedom to lock database objects.

8.12 NONLOCKING TECHNIQUE– TIMESTAMPING CONTROL

The two-phase locking technique relies on locking to ensure that each interleaved schedule that is executed is serializable and that makes 2PL attractive. Now we discuss a technique that does not use locks and is therefore deadlock free, and works quite well when the level of contention between transactions running concurrently is not high. It is called *timestamping*. It should be noted that this technique works well only under specific conditions. Its performance can degrade when the conditions are very different.

The timestamping technique gets rid of locks by assigning a unique timestamp, which is a number indicating the time of start of the transaction or just a counter that is incremented every time a new timestamp is requested, for each transaction at the start of the transaction. It then insists that the schedule executed is always serializable to the serial schedule in the chronological order of their timestamps. This is in contrast to two-phase locking where any schedule that is equivalent to some serial schedule is acceptable.

Since the scheduler will only accept schedules that are serializable to the serial schedule in the chronological order of the timestamps, the scheduler must insist that in case of conflicts the junior transaction processes information only after the older transaction has written it. Note that the transaction with the smaller timestamp is the older transaction. For the scheduler to be able to carry out this control, the data items also need to have read and write timestamps. The read timestamp of a data item X is the timestamp of the youngest transaction that has read it and the write timestamp is the timestamp of the youngest transaction that has written it. Let the timestamp of a transaction T be TS(T).

Consider a transaction T_1 with timestamp = 100. Suppose the smallest unit of concurrency control is a table and the transaction wishes to read or write some rows from a table named R. Let the read timestamp of R be RT(R) and the write timestamp of R be WT(R). The following situations may then arise:

1. T_1 wishes to read R. The read by T_1 will be allowed only if WT(R) < TS(T_1) or WT(R) < 100. That is, the last write was by an older transaction which had a smaller timestamp. This condition ensures that a data item read by a transaction has not been written by a more recent transaction. If the write timestamp of R is larger than 100 (that is, a younger transaction has written it), then the older transaction is rolled back and restarted with a new timestamp.
2. T_1 wishes to read R. The read by T_1 will be allowed if it satisfies the above condition that R has been last written by an older transaction even if RT(R) > TS(T_1), that is, R has been read by a younger transaction. Therefore, reading a data item that has been read by a younger transaction is quite acceptable as long as it has not been written by a younger transaction.
3. T_1 wishes to write some rows of table R. The write will be allowed only if the last read of the transaction was by an older transaction and no younger transaction has read the table R. Therefore, the write is allowed if RT(R) < TS(T_1).
4. T_1 wishes to write some rows of table R. The write need not be carried out if the last write was by a younger transaction, that is, WT(R) > TS(T_1). The reason this write is not needed is that in this case the younger transaction has not read R, since if it had, the older transaction T_1 would have been aborted. If R had been read by a younger transaction, the older transaction would have been rolled back and restarted with a new timestamp. It is not necessary to check the write since if a younger transaction has written the relation, the younger transaction would have read the data item, and the older transaction operated on an old data item can be ignored. It is possible to ignore the write if the second condition is violated since the transaction is attempting to write an obsolete data item.

Now we illustrate these points in Fig. 8.28. A schedule with two transactions is shown in this figure. We assume transaction T_1 to be older and therefore TS(T_1) < TS(T_2).

Let us first explain the information given in the figure. The first column presents the actions carried out by transaction 1. In addition, we have presented the read and write timestamp values of the objects that the transaction reads or writes. For example, the first value in the first row "Read A (TS=95)" tells us that the

transaction wishes to read object A which has a read timestamp (RT) of 95 before the object is read. The second column gives all values to be equal to the timestamp of the transaction which is 100 after the transaction has read or written the object values, since that is the last transaction to have accessed those objects.

Transaction 1 (Old timestamp)	New Timestamps TS(T_1)=100	Time	Transaction 2 (Old timestamp)	Timestamps TS(T_2)=150
Read A (TS=95)	100	1	–	
$A := A - 200$		2	–	
Write A (TS=97)	100	3	–	
–		4	Read B (TS=102)	150
–		5	$B := B - 300$	
–		6	Write B (TS=105)	150
Read C (TS=98)	100	7	–	
$C := C + 200$		8	–	
Write C (TS=99)	100	9	–	
–		10	Read C (TS=110)	150
–		11	$C := C + 300$	
–		12	Write C (TS=115)	150

Figure 8.28 Example of concurrency control using timestamping

We note that if the RT(C) value at time 7 was 102, the transaction will be allowed to read C but if WT(C) is 105 then the transaction is rolled back.

Read (A) by transaction T_1 is permitted since A has not been written by a younger transaction. Write (A) by transaction T_1 is also permitted since A has not been read or written by a younger transaction. Read (B) by transaction T_2 is permitted since B has not been written by a younger transaction. Read (B) by transaction T_1 however is not allowed since B has been read by (younger) transaction T_2 and therefore transaction T_1 is rolled back.

8.13 NONLOCKING TECHNIQUE– OPTIMISTIC CONTROL

The locking concurrency control and the timestamping concurrency control both implicitly assume that conflicts are frequent between the transactions and therefore concurrency controls are needed to control the conflicts and maintain database consistency. If on the other hand we assume that our database environment is such that conflicts are uncommon then we might be able to use a different type of concurrency control mechanism. Such an environment could well exist for a database system like that of an Internet trader amazon.com or eBay where most visitors to the site only access parts of the database to check the price or read product

reviews and only a few visitors, as a percentage of visitors, buy products or modify data in other ways (for example, by writing book reviews).

The optimistic concurrency control approach is suitable for database applications where the number of conflicts between the transactions is expected to be low. The technique is not suitable for intensive applications like the airline reservation system where write operations frequently occur at 'hot spots'. For example, status of the flight for the next few days or for holiday periods. In the optimistic control technique, transactions are allowed to execute unhindered and are validated only after they have reached their commit points. If the transaction validates, it commits, otherwise it is restarted.

The optimistic control may be summarized as involving checking at the end of the execution and resolving conflicts by rolling back. One possible algorithm involves the following three phases:

1. *The Read Phase*—Each active transaction is allocated a private space which cannot be accessed by other transactions or by the outside world. Each transaction is then allowed to execute such that all writes are carried out only in the private space. At the end of this phase the transaction is ready to commit and the concurrency controller knows the read and write sets of each transaction.
2. *The Validation Phase*—The transaction is now validated by the scheduler which checks if the transaction violates any serializability conditions. If the transaction is found to be correct, it is validated otherwise aborted. The correct transaction goes to the next phase.
3. *The Write Phase*—The private space data of the validated transaction is now transferred to the database and the transaction has committed.

Optimistic concurrency control does not involve any checking before or during the execution of the transaction. Usually any updates carried out by the transaction are carried out to a local copy of the objects and not carried out directly to the database objects. At the completion of the transaction, validation is carried out. Validation requires certain information be retained by the system. This information may be transaction timestamps as well as a list of objects that are read and written by each transaction. We do not provide further details of validation.

If at the time of validation, it is discovered that the transaction wanting to commit had read a data item that has already been written by another transaction, the transaction attempting to commit would be restarted. These techniques are also called validation or certification techniques.

It is possible to combine two or more concurrency control techniques, for example, very heavily used objects could be locked, and others could follow optimistic control.

8.14 TRANSACTION SUPPORT IN SQL

SQL provides a number of facilities for transaction management. Three characteristics of a transaction may be specified in a SET TRANSACTION command by the user. These characteristics are as follows:

- Access mode
- Diagnostic area size
- Isolation level

1. *Access Mode*—A transaction may be specified not to modify the database by specifying READ ONLY. It may otherwise be specified to be READ WRITE which is also the default.
2. *Diagnostic Area Size*—The diagnostic area size specifies the number of error conditions that can be held simultaneously.
3. *Isolation Level*—This transaction characteristic specifies the extent to which the transaction is exposed to the actions of the other transactions. The default value of this characteristic is SERIALIZABLE which provides the highest level of isolation. This isolation level ensures that the transaction reads the results of only committed transactions. Locks are obtained before reading or writing objects and released at the completion of the transaction thus ensuring 2PL.

The choices available under isolation level are as follows:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

READ UNCOMMITTED allows the transaction to read changes made by another transaction that has not committed. This is called dirty read and is the lowest isolation level. READ COMMITTED is often the default isolation in DBMS. It ensures that the transaction only reads the changes made by committed transactions and no data that has been changed by another transaction but not yet committed will be read. Shared locks are used in READ COMMITTED. REPEATABLE READ also ensures that the transaction reads only the changes made by committed transactions and dirty data is not read. SERIALIZABLE is the most restrictive isolation level. It ensures that no other user is able to insert or update rows into the data set until the first transaction is completed.

Some problems (for example, dirty read) can occur if an isolation level lower than serializable is specified. Therefore, serializable is the safest isolation to use in transaction processing.

8.15 EVALUATION OF CONCURRENCY CONTROL MECHANISMS

A number of researchers have evaluated the various concurrency control mechanisms that have been discussed here. It was found that the level of resources available has significant impact on which concurrency method performs the best. It is generally accepted that a technique that minimizes the number of restarts (like the two-phase locking technique) is superior when the physical resources are limited and number of conflicts is high. Two-phase locking performed much better than timestamping and optimistic algorithms under medium to high conflict levels since timestamping and optimistic methods tend to waste the most resources through restarts. When the conflict level is low and there are sufficient resources, then restart-oriented methods are better since they can provide a higher degree of concurrency. The optimistic algorithm was found to perform the best in such an environment. Deadlock prevention techniques like those based on immediate restart tend to make the database slow down somewhat. It has been found that deadlock detection techniques help. At low loads, it does not really matter much which technique is used, although some studies show that the optimistic

technique then is more efficient. It is of course possible to use more than one technique in a system. For example, two-phase locking may be combined with optimistic techniques to produce hybrid techniques that work well in many situations.

SUMMARY

In this chapter we have considered the concept of transaction and its ACID properties and have discussed the importance of concurrency control. The following topics have been covered:

- The concept of transaction and its ACID properties (atomicity, consistency, isolation and durability) are important in transaction management and concurrency control.
- Conflicts may arise during concurrent execution of two transactions including RW conflict and WW conflict.
- The conflicts can lead to concurrency anomalies (lost update, inconsistent retrievals and uncommitted dependency).
- An important concept in concurrency control is the concept of a schedule and a serial schedule.
- For concurrent execution, a schedule must be a serializable schedule.
- For a schedule to be serializable, it must meet the requirements of serializability.
- Three types of serializability are described, viz., final state serializability (FSR), view serializability (VSR) and conflict serializability (CSR).
- CSR may be tested using a precedence graph.
- A number of concurrency control methods depend on the concept of locks, shared locks and exclusive locks to enforce serializability.
- Simple (or single-phase) locking is not always satisfactory and therefore the concept of two-phase locking (2PL) has been proposed.
- 2PL can lead to the problem of deadlocks.
- A number of methods are available to prevent deadlocks or for detecting and resolving these using wait-for graphs.
- It is necessary to deal with the issue of lock granularity, should lock granularity be fine or coarse.
- Another approach is multiple granularity using intention locks.
- It is possible to use concurrency control methods that do not use locks; two such methods are timestamping and optimistic concurrency control.
- Research has been carried out in performance evaluation of timestamping and 2PL to find when timestamping performs better than 2PL and when 2PL performs better.

REVIEW QUESTIONS

1. List the properties that a transaction must have. Briefly explain them. (Section 8.2)
2. Why is the concept of a transaction important in concurrency? (Section 8.2)

3. When does concurrent execution of two transactions lead to conflict? (Section 8.1 and 8.6.5)
4. What concurrency anomalies (lost update, inconsistent retrievals, uncommitted dependency) are possible if there is no concurrency control? (Section 8.3)
5. Give an example of a schedule of three transactions. How many different serial schedules are possible? List them. (Section 8.4)
6. What is a recoverable schedule? Define it. (Section 8.5)
7. When do two transactions running concurrently conflict? Explain RW and WW conflicts. (Section 8.6.5)
8. What is final state serializability? (Section 8.6.1)
9. Explain view serializability (VSR). Is it possible to test for VSR? (Section 8.6.2)
10. What is conflict serializability and how it differs from VSR? (Section 8.6.3)
11. What is a precedence graph? What does it show? (Section 8.6.4)
12. What is a serializable schedule? Why are serializable schedules considered correct? (Sections 8.6 and 8.7)
13. What are the issues in determining locking granularity? What is multiple granularity? (Section 8.10)
14. What is intention locking? How is it used in multiple granularity? (Section 8.11)
15. What is two-phase locking and how does it guarantee serializability? Why simple (or single-phase) locking is not satisfactory? (Section 8.8.2)
16. Explain how do you determine that a given schedule will lead to deadlock. What is wait-for graph? (Section 8.9.1)
17. How can deadlocks be prevented and how can deadlocks be resolved? (Section 8.9.2 and 8.9.3)
18. What is the basis of the timestamping technique? How does the timestamping technique differ from 2PL? (Section 8.12)
19. When does timestamping perform better than 2PL and vice versa? (Section 8.12)
20. Explain the basis of optimistic concurrency control. When does it work well? (Section 8.13)
21. Explain how SQL supports transaction. (Section 8.14)
22. Explain how different concurrency control techniques are evaluated. (Section 8.15)

SHORT ANSWER QUESTIONS

1. What is concurrency?
2. What problems may arise if concurrency control is not used in a database system?
3. Is concurrency important in a library catalogue database? Why?
4. What are the ACID properties of a transaction?
5. Why is the concept of transaction important in database systems?
6. Give an example of a schedule of three example transactions.
7. Give an example of a schedule that is not recoverable.
8. What is serializability?
9. Give an example of a serial schedule.
10. Define final state serializability (FSR).

11. Why do we need other types of serializability?
12. What is VSR?
13. What is CSR?
14. Which one of the three—FSR, VSR and CSR, is the strictest?
15. What does 2PL do?
16. What are locks?
17. Give an example of a precedence graph with a cycle.
18. What is meant by granularity?
19. Give an example of coarse granularity.
20. What is intention locking?
21. What is the main feature of timestamping concurrency control?
22. What is the main feature of optimistic concurrency control?
23. Give one example of deadlock prevention.
24. How can one detect a deadlock?
25. Give an example of a wait-for graph.
26. What is the aim of multiple granularity locking?
27. What may be specified in SET TRANSACTION mode in SQL?
28. What isolation levels may be specified in SQL?
29. What was discovered by evaluating concurrency control methods?

MULTIPLE CHOICE QUESTIONS

1. Which of the following problems does concurrency controls deal with?
   (a) lost updates (b) inconsistent retrievals
   (c) uncommitted dependency (d) All of the above
2. Which of the following are correct?
   (a) If two transactions are accessing the same object and writing them at the same time without any controls then one of the updates can be lost.
   (b) If one transaction is updating a number of items and another transaction is reading them without any controls then that may result in inconsistence retrievals.
   (c) If one transaction reads something that is written by another transaction and commits before the other transaction commits then it is called uncommitted dependency.
   (d) All of the above
3. In the following list, mark all properties that a transaction must have.
   (a) Serializability (b) Consistency (c) Atomicity (d) Isolation
4. Which one of the following is **not** correct?
   (a) A transaction is an atomic unit of work, all its subtasks must be processed together one after another.
   (b) A transaction may not be nested.

(c) A transaction is a logical unit of work which preserves the consistency of the database.
(d) A transaction is usually a serial program devoid of loops.

5. Three transactions attempt to book seats on a flight that has 12 seats available. The transactions are transaction T_1 for 3 seats, transaction T_2 for 5 seats and transaction T_3 for 7 seats. If a schedule that is serializable is executed, the number of seats sold cannot be
   (a) 7 (b) 8 (c) 10 (d) 12
   (e) None of the above
6. Which of the following are correct?
   (a) Although transactions appear atomic, they consist of a sequence of more elementary steps.
   (b) Although operations may be happening concurrently in the middle of a transaction, it is the role of concurrency control that each transaction is isolated from other operations.
   (c) To manage concurrency, the database must be partitioned into items which are the units of data to which access is controlled.
   (d) All of the above
7. Which of the following is **not** correct?
   (a) A schedule for a set of transactions is a specification of the order in which the elementary steps of the transactions are carried out.
   (b) A schedule is serial if all the steps of each transaction occur consecutively.
   (c) A schedule is serializable if its effect is equivalent to that of some serial schedule.
   (d) Each schedule that is serializable is allowed by 2PL.
8. Which of the following is correct? Two transactions running concurrently and accessing the same data item do **not** conflict if
   (a) both transactions are reading the data item but neither is writing it.
   (b) the first transaction is reading the data item and the second writing it.
   (c) the first transaction is writing the data item and the second reading it.
   (d) both transactions are writing the data item.
9. Consider the following schedule:

Transaction 1	Time	Transaction 2
Read (A)	1	–
$A := A - 50$	2	–
Write (A)	3	–
–	4	Read (B)
–	5	Read (A)
Read (B)	6	–
$B := B + 50$	7	–
–	8	Write ($A + B$)
Write (B)	9	–

Which of the following is correct?

(a) The schedule is serializable.
(b) A deadlock happens if 2PL is used to execute the above schedule.
(c) If timestamping is used to execute the above schedule then assuming that Transaction 1 is the older transaction, Transaction 1 is rolled back.
(d) None of the above

10. Which of the following schedules are serializable schedules? $r_i(K)$ means a read by transaction i of data item K. w indicates a write.
(a) $r_1(A)w_1(A)\ r_2(B)w_2(B)\ r_3(D)w_3(D)\ r_1(C)w_1(C)\ r_2(C)w_2(C)$
(b) $r_3(D)w_3(D)\ r_1(A)w_1(A)\ r_1(C)w_1(C)\ r_2(B)w_2(B)\ r_2(C)w_2(C)$
(c) $r_1(A)w_1(A)\ r_2(B)w_2(B)\ r_1(C)w_1(C)\ r_2(C)w_2(C)\ r_3(D)w_3(D)$
(d) All of the above

11. Which one of the following schedules is **not** a recoverable schedule? In the schedules below Y indicates a transaction commit while X denotes a transaction abort. The subscripts show the transaction numbers.
(a) $r_1(C)w_1(C)\ r_2(B)w_2(B)\ r_2(C)w_2(C)(Y_2)\ r_3(D)w_3(D)\ r_1(A)(X_1)$
(b) $r_1(A)w_1(A)\ r_2(B)w_2(B)\ r_1(C)w_1(C)\ r_2(C)w_2(C)\ r_3(D)w_3(D)$
(c) $r_1(A)w_1(A)\ r_3(D)w_3(D)(Y_3)\ r_2(B)w_2(B)\ r_1(C)w_1(C)\ r_2(C)w_2(C)(X_1)$
(d) All of the above

12. Which one of the following is true?
(a) There are no schedules that are acceptable to a two-phase locking scheme but not to a timestamping scheme.
(b) A timestamping schedule is always equivalent to the serial schedule in the order of the timestamps of the transactions.
(c) Timestamping concurrency control works well even when the level of contention in the database is high.
(d) Timestamping allows no schedules that would not be permitted by 2PL.

13. Which one of the following is true?
(a) 2PL schedules are equivalent to serial schedules in the order of the acquisition of the last locks.
(b) 2PL schedules are equivalent to serial schedules in the order of the acquisition of the first locks.
(c) Timestamping is a more strict concurrency control mechanism (in that it allows less concurrency).
(d) A timestamping technique allows fewer sets of schedules than a dynamic two-phase locking scheme.

14. Which of the following are true?
(a) All serial schedules are serializable.
(b) All FSR schedules are serial schedules.
(c) All VSR schedules are FSR schedules.
(d) All of the above are true.

15. Which one of the following is true?
(a) All schedules are FSR schedules.
(b) CSR is the most stringent serializability requirement we have discussed.
(c) All FSR schedules are VSR schedules.
(d) All VSR schedules are CSR schedules.

16. Which one of the following is true?
    (a) If a schedule is not at least FSR then it is not serializable.
    (b) A serializable and correct schedule must be CSR.
    (c) All FSR schedules can be easily tested for FSR.
    (d) All serializable schedules are correct.
17. Which one of the following techniques is **not** guaranteed to prevent deadlocks?
    (a) The younger transaction is not allowed to wait for a lock if the data item is held by an older transaction.
    (b) The older transaction is not allowed to wait for a lock if the data item is held by a younger transaction.
    (c) Some transactions are selected randomly and not allowed to wait for a data item that is already locked.
    (d) No transaction is allowed to wait for a lock if the data item is held by another transaction.
18. In timestamping, consider the smallest unit of concurrency control to be a relation and let transaction T at time t wish to read or write some rows from the relation R. Let the read and write timestamps of R be r and w respectively.
    Which of the following is *not* correct?
    (a) If $r < t$, the read by T will be allowed only if $w < t$.
    (b) If $r > t$, the read by T will not be allowed.
    (c) If $w < t$, the write by T will be allowed only if $r < t$.
    (d) If $w > t$, the write by T need not be carried out if $r < t$.
    (e) All of the above are correct.

EXERCISES

1. Three transactions A, B and C arrive in the time sequence A, then B and then C. The transactions are run concurrently on the database. Can we predict what the result would be if 2PL is used? Explain briefly.
2. Show whether the given schedule is possible if 2PL is used or if timestamping is used.
3. Can a schedule that is not serializable be correct? Give an example.
4. Show an example of a schedule that is final state serializable but not conflict serializable.
5. Explain the use of intention locks in multiple granularity and describe lock conflicts when intention locking is used.
6. Is the following schedule serializable? Show what happens if 2PL is used to execute the schedule.

Transaction 1	Time	Transaction 2
Read (A)	1	–
$A := A - 50$	2	–
Write (A)	3	–

Contd.

Contd.

–	4	Read (B)
–	5	Read (A)
Read (B)	6	–
$B := B + 50$	7	–
–	8	Write ($A + B$)
Write (B)	9	–

7. Explain the hierarchy of serializable schedules. Why are view serializable schedules a subset of FSM schedules and CSR schedules are a subset of the VSR schedules? Can you think of any serializable schedules that are a subset of the CSR schedules?
8. What is a recoverable schedule? Discuss what happens when a schedule is not recoverable. Explain the situation when two transactions have committed before a transaction having carried out a dirty write aborts.
9. Can a schedule that is not serializable be correct? Give an example.
10. Find out what other types of serializability has been proposed. How do they compare to final state serializability and conflict serializability?
11. Give an example of a schedule that is final state serializable but not conflict serializable.
12. Illustrate what happens if timestamping is used to execute the above schedule. Assume that Transaction 1 is the older transaction.
13. Explain why deadlocks occur and the role of the lock manager.
14. Briefly explain one deadlock prevention and one deadlock detection algorithm. Compare the efficiency of the two.
15. Does timestamping use locking? What is the motivation for timestaming and in what type of database environment it works best? Explain briefly.
16. What is the single most significant difference between two-phase locking and the timestamping technique of concurrency control? Explain briefly.
17. Under what conditions do you think deadlock prevention is better than deadlock detection and resolution?
18. If deadlock is prevented, is starvation still possible? Explain clearly.
19. Give an example schedule for two transactions that is allowed under timestamping control but not under 2PL.
20. Consider the following transactions:

T_1	T_2
read(x)	read(x)
$x := x - n$	$x := x + m$
write(x)	write(x)
read(y)	
$y := y + n$	
write(y)	

How many different schedules of the two transactions are possible? How many of these are serializable?

PROJECTS

1. Consider an airline booking system and let us assume we have implemented the following schema which is a rather primitive system:

   Customer(CustomerID, Title, First name, Last name, Gender, Age, Address, TelephoneBH, TelephoneAH, Email)

   Flight(FlightNUM, DepartureDate, DepartureTime, ArrivalDate, ArrivalTime)

   Seat(FlightNUM, CustomerID, DepartureDate, SeatNUM, Class)

   Ticket(SerialNUM, FlightNUM, DepartureDate, SeatNUM, CustomerID)

There is little doubt that this schema has a number of weaknesses but we will ignore them at the moment. The table *Customer* includes the details of each customer. No customer can make a booking unless the person is in the *Customer* table. There is a separate row in the table *Flight* for each flight and departure date and time. Seats are allocated when booking is made. The ticket is assumed to be for only one flight and only one customer.

Which tables in the above schema are likely to be hot spots? How should the concurrency be controlled for the system to work efficiently? Discuss the issues of serializability, granularity, deadlocks and recoverability. Suggest how the database design could be modified to deal with concurrency more effectively.

LAB EXERCISES

1. Consider a bank, say ICICI, which, let us assume, has more than 1000 branches and more than 5000 ATMs all over India. What time of the day is the peak traffic time for the ATMs? Make an assumption on how many ATM transactions are carried out at peak time and then make an assumption of a simple schema for the bank customer database. What concurrency problems might arise? Discuss.
2. It has been estimated that close to four trillion SMS will be sent all over the world in 2011 and there will be 700 million mobile customers in India in 2011. Annual SMS traffic in India might be estimated to grow to 500 billion messages per year. Estimate how many messages per second might be sent on Diwali or 1st January all over India. Discuss the concurrency problems of such high traffic for mobile phone databases in India.
3. Are the ACID properties of transactions always required? Discuss a situation when it may be possible not to worry about one or more of them.
4. The conditions in Figs 8.15 and 8.17 provide a total of nine combinations of conditions which can lead to a cycle in the precedence graph. Show how each of these combination may be avoided if 2PL is used.

BIBLIOGRAPHY

For Students

The paper by Thomasian is good. The book by Bernstein et al., is quite good for an introduction. An older version is available free on the Web.

For Instructors

There are a number of books that deal exclusively with transaction processing and these books provide a much more detailed discussion on concurrency control and other transaction processing topics. Some of these books are listed below including the classic book by Gray and Reuter and the one by Papadimitriou. The book by Kifer et al., is also good. The book edited by Kumar has 21 papers in it with a focus on performance of concurrency control techniques. The paper by Agarwal, Carey and Livny deals with performance evaluation of concurrency control methods in detail.

Agarwal, R., and D. DeWitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation", *ACM Transactions of Database Systems*, Vol. 10, pp 529-564, 1985.

Agarwal, R., M. J. Carey, and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications", *ACM Transactions on Database Systems,* Vol. 12, December 1987.

Agarwal, R., M. J. Carey and L. McVoy, " The performance of alternative strategies for dealing with deadlocks in database management systems", *IEEE Transactions on Software Engineering*, Vol. 13, Issue 12, pp 1348-1363 , 1987.

Bernstein, P. A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, 1987 version. http://research.microsoft.com/~philbe/ccontrol/

Bernstein, P. A., and E. Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann Series in Data Management Systems, Second Edition, 2009, pp 400.

Eswaran, K. P., J. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system", *Communications of the ACM*, Vol. 19, pp 624-633.

Gray, J., "The Transaction Concept: Virtues and Limitations", *Proceedings of the 7th International Conference on Very Large Database Systems*, September 1981.

Gray, J., and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Series in Data Management Systems, 1993.

Gray, J. N., R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of locks and degrees of consistency in a shared data base", *Modeling in Data Base Management Systems*, edited by G. M. Nijssen, North Holland, 1976, pp 365-394.

Kumar, V. (Ed.), *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, Prentice Hall, 1996.

Kung, H. T., and C. H. Papadimitriou, "An Optimality Theory of Concurrency Control in Databases", *Proc. ACM SIGMOD Conference*, pp 116-126, 1979.

Kifer, M., A. Bernstein and P. M. Lewis, *Databases and Transaction Processing: An Application-Oriented Approach*, Second Edition, Addison-Wesley, 2006.

Papadimitriou, C. H., "The Serializability of Concurrent Database Updates", *Journal of the ACM*, Vol. 26, 1979, pp 631-653.

Papadimitriou, C. H., *The Theory of Database Concurrency Control*, Computer Science Press, New York, 1986.

Thomasian, A.,"Concurrency Control: Methods, Performance, and Analysis", *ACM Computing Surveys*, Vol. 30, No. 1, 1998, pp. 70-119.

Weikum, G., and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*, Morgan Kaufmann, 2002.