



ALAGAPPA UNIVERSITY

[Accredited with 'A+' Grade by NAAC (CGPA:3.64) in the Third Cycle
and Graded as Category-I University by MHRD-UGC]

(A State University Established by the Government of Tamil Nadu)

KARAIKUDI – 630 003



Directorate of Distance Education

B.Sc. [Computer Science]

V - Semester

130 52

**RELATIONAL DATABASE
MANAGEMENT SYSTEMS (RDBMS)**

Authors:

Manas Ghosh, *Lecturer, RCC Institute of Information Technology, Kolkata*

Sudipta Pathak, *System Consultant, COGNIZANT*

Units: (1.0-1.3, 2.5-2.6, 3.0-3.4, 3.6-3.10, 5.2, 7.0-7.2.7, 7.5-7.9, 8, 9.0-9.2, 9.4-9.9, 10, 11.0-11.2, 11.4-11.8, 13.0-13.2, 13.4-13.8)

Dr Preety Khatri, *Assistant Professor, Institute of Management Studies (IMS), Noida*

Units: (2.4, 3.5, 4, 5.4, 6, 7.2.8-7.4, 9.3, 12, 13.3, 14)

Vikas Publishing House, Units: (1.4-1.9, 2.0-2.3, 2.7-2.12, 5.0-5.1, 5.3, 5.5-5.9, 11.3)

"The copyright shall be vested with Alagappa University"

All rights reserved. No part of this publication which is material protected by this copyright notice may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the Alagappa University, Karaikudi, Tamil Nadu.

Information contained in this book has been published by VIKAS Publishing House Pvt. Ltd. and has been obtained by its Authors from sources believed to be reliable and are correct to the best of their knowledge. However, the Alagappa University, Publisher and its Authors shall in no event be liable for any errors, omissions or damages arising out of use of this information and specifically disclaim any implied warranties or merchantability or fitness for any particular use.



Vikas is the registered trademark of Vikas Publishing House Pvt. Ltd.

VIKAS PUBLISHING HOUSE PVT. LTD.

E-28, Sector-8, Noida - 201301 (UP)

Phone: 0120-4078900 • Fax: 0120-4078999

Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44

• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

Work Order No. AU/DDE/DE-12-27/Preparation and Printing of Course Materials/2020 Dated 12.08.2020 Copies -

SYLLABI-BOOK MAPPING TABLE

Relational Database Management Systems (RDBMS)

Syllabi	Mapping in Book
BLOCK - I: INTRODUCTION	
UNIT 1: Data Base System Applications: Data Base System VS File System – View of Data – Data Abstraction – Instances and Schemas – Data Models – the ER Model	Unit 1: Database System Applications (Pages 1-15)
UNIT 2: Model : Relational Model – Other Models – Database Languages – DDL –DML – DataBase Access for Applications Programs – Data Base Users and Administrator – Transaction Management – DataBase System Structure – Storage Manager – the Query Processor.	Unit 2: Models (Pages 16-44)
UNIT 3: History of Data Base Systems: Data Base Design and ER Diagrams – Beyond ER Design Entities, Attributes and Entity Sets – Relationships and Relationship Sets – Additional Features of ER Model – Concept Design with the ER Model – Conceptual Design for Large enterprises.	Unit 3: History of Database Systems (Pages 45-65)
BLOCK - II: RELATIONAL MODEL	
UNIT 4: Introduction: Integrity Constraint Over Relations – Enforcing Integrity constraints – Querying Relational Data – Logical DataBase Design –Introduction to Views – Destroying / Altering Tables and Views.	Unit 4: Introduction to Constraints and Views (Pages 66-80)
UNIT 5: Relational Algebra: Selection and Projection Set Operations – Renaming –Joins – Division – Examples of Algebra Overviews –	Unit 5: Relational Algebra (Pages 81-97)
UNIT 6: Relational Calculus: Tuple Relational Calculus – Domain Relational Calculus – Expressive Power of Algebra and Calculus.	Unit 6: Relational Calculus (Pages 98-107)
BLOCK - III: SQL QUERY	
UNIT 7: Form of Basic SQL Query: Examples of Basic SQL Queries – Introduction to Nested Queries – Correlated Nested Queries Set – Comparison Operators – Aggregative Operators – NULL Values – Comparison using Null Values – Logical Connectivity's – AND, OR and NOT – Impact on SQL Constructs – Outer Joins – Disallowing NULL Values – Complex Integrity Constraints in SQL Triggers and Active DataBases. Schema Refinement	Unit 7: Form of Basic SQL Query (Pages 108-216)
UNIT 8: Normal Forms : Problems Caused by Redundancy – Decompositions – Problem Related to Decomposition – Reasoning about FDS – FIRST, SECOND, THIRD Normal forms – BCNF –	Unit 8: Normal Forms (Pages 217-243)
UNIT 9: Join: Lossless Join Decomposition – Dependency Preserving Decomposition – Schema Refinement in DataBase Design – Multi Valued Dependencies – FORTH Normal Form.	Unit 9: Join (Pages 244-255)

BLOCK - IV: TRANSACTION

UNIT 10: Introduction : Transaction Concept- Transaction State- Implementation of Atomicity and Durability – Concurrent – Executions – Serializability- Recoverability – Implementation of Isolation – Testing for Serializability

UNIT 11: Protocols : Lock Based Protocols – Timestamp Based Protocols- Validation- Based Protocols – Multiple Granularity.

UNIT 12: Recovery and Atomicity: Log – Based Recovery – Recovery with Concurrent Transactions – Buffer Management – Failure with Loss of Nonvolatile Storage-Advance Recovery Systems- Remote Backup Systems

Unit 10: Basic Concept of Transaction
(Pages 256-281)

Unit 11: Protocols
(Pages 282-306)

Unit 12: Recovery and Atomicity
(Pages 307-324)

BLOCK - V: STORAGE

UNIT 13: Data on External Storage: File Organization and Indexing – Cluster Indexes, Primary and Secondary Indexes – Index Data Structures – Hash Based Indexing– Tree Base Indexing – Comparison of File Organizations – Indexes.

UNIT 14: Performance Tuning: Intuitions for Tree Indexes – Indexed Sequential Access Methods (ISAM) – B+ Trees: A Dynamic Index Structure.

Unit 13: Data on External Storage
(Pages 325-344)

Unit 14: Performance Tuning
(Pages 345-354)

CONTENTS

INTRODUCTION**BLOCK I: INTRODUCTION****UNIT 1 DATABASE SYSTEM APPLICATIONS****1-15**

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Database System vs File System
 - 1.2.1 File-Based Systems
 - 1.2.2 Database Management System—A Better Alternative
- 1.3 Basics of Database Management System
 - 1.3.1 Data Abstraction
 - 1.3.2 Logical and Physical View of Data
 - 1.3.3 Schemas, Subschema Instances or State of a Database
- 1.4 Data Models
- 1.5 ER Model
- 1.6 Answers to Check Your Progress Questions
- 1.7 Summary
- 1.8 Key Words
- 1.9 Self-Assessment Questions and Exercises
- 1.10 Further Readings

UNIT 2 MODELS**16-44**

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Types of Data Models
 - 2.2.1 Hierarchical Data Model
 - 2.2.2 Relational Data Model
 - 2.2.3 Network Data Model
 - 2.2.4 Comparison of three Models
- 2.3 Database Languages
- 2.4 Database Access for Application Programs
- 2.5 DBMS Users
- 2.6 Transaction Management
- 2.7 Database Systems Structure
 - 2.7.1 DBMS Structure
 - 2.7.2 Applications of DBMS
 - 2.7.3 Disadvantages of DBMS
 - 2.7.4 Storage Manager
 - 2.7.5 Query Processor
- 2.8 Answers to Check Your Progress Questions
- 2.9 Summary
- 2.10 Key Words
- 2.11 Self Assessment Questions and Exercises
- 2.12 Further Readings

UNIT 3 HISTORY OF DATABASE SYSTEMS**45-65**

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Database and Concept Design Using Entity–Relationship Model

- 3.2.1 Entity
- 3.2.2 Attribute
- 3.2.3 Types of Attributes
- 3.2.4 Relationship
- 3.2.5 Classifying Relationships
- 3.2.6 Unary Relationship
- 3.2.7 Ternary Relationship
- 3.2.8 Multiplicity
- 3.3 Entity–Relationship Diagram
 - 3.3.1 E–R Diagram Notation
 - 3.3.2 Attributes of a Relationship
 - 3.3.3 Representing Participation in E–R Diagram
- 3.4 Additional Features of E–R Model
 - 3.4.1 Attribute Inheritance
 - 3.4.2 Constraints on Specialization and Generalization
 - 3.4.3 Aggregation
- 3.5 Conceptual Design for Large Enterprises
- 3.6 Answers to check your progress Questions
- 3.7 Summary
- 3.8 Key Words
- 3.9 Self Assessment Questions and Exercises
- 3.10 Further Readings

BLOCK II: RELATIONAL MODEL

UNIT 4 INTRODUCTION TO CONSTRAINTS AND VIEWS

66-80

- 4.0 Introduction
- 4.1 Objectives
- 4.2 Integrity Constraints and Enforcing Integrity Constraints Over Relations
- 4.3 Querying Relational Data
- 4.4 Logical Database Design
- 4.5 Views
- 4.6 Answers to Check Your Progress Questions
- 4.7 Summary
- 4.8 Key Words
- 4.9 Self Assessment Questions and Exercises
- 4.10 Further Readings

UNIT 5 RELATIONAL ALGEBRA

81-97

- 5.0 Introduction
- 5.1 Objectives
- 5.2 Basic Operations of Relational Algebra
- 5.3 Different Types of Join
 - 5.3.1 Natural Join
 - 5.3.2 Θ -Join and Equijoin
 - 5.3.3 Semijoin
 - 5.3.4 Antijoin
 - 5.3.5 Outer Join
- 5.4 Division-Examples of Algebra Overviews
- 5.5 Answers to Check Your Progress Questions
- 5.6 Summary
- 5.7 Key Words
- 5.8 Self Assessment Questions and Exercises
- 5.9 Further Readings

UNIT 6 RELATIONAL CALCULUS**98-107**

- 6.0 Introduction
- 6.1 Objectives
- 6.2 Relational Calculus
 - 6.2.1 Tuple Relational Calculus
 - 6.2.2 Domain Relational Calculus
- 6.3 Expressive Power of Algebra and Calculus
- 6.4 Answers to Check Your Progress Questions
- 6.5 Summary
- 6.6 Key Words
- 6.7 Self Assessment Questions and Exercises
- 6.8 Further Readings

BLOCK III: SQL QUERY**UNIT 7 FORM OF BASIC SQL QUERY****108-216**

- 7.0 Introduction
- 7.1 Objectives
- 7.2 Relational Database Query Language
 - 7.2.1 Forms of SQL
 - 7.2.2 Types of SQL Commands
 - 7.2.3 SQL Statements
 - 7.2.4 Data Definition Language (DDL)
 - 7.2.5 Data Manipulation Language (DML)
 - 7.2.6 Join in SQL
 - 7.2.7 Nested Query/Subquery in SQL
 - 7.2.8 Impact on SQL Constructs
- 7.3 Integrity Constraints in SQL Triggers and Active Databases
- 7.4 Introduction to Schema Refinement
- 7.5 Answers to Check Your Progress Questions
- 7.6 Summary
- 7.7 Key Words
- 7.8 Self Assessment Questions and Exercises
- 7.9 Further Readings

UNIT 8 NORMAL FORMS**217-243**

- 8.0 Introduction
- 8.1 Objectives
- 8.2 Problems Caused by Redundancy and Normalization
- 8.3 Decomposition
- 8.4 Reasoning about FDS
 - 8.4.1 Types of Dependency
 - 8.4.2 Armstrong Axioms—Inference Rules for Functional Dependencies
 - 8.4.3 Key and Functional Dependency
- 8.5 First Normal Form
- 8.6 Second Normal Form
- 8.7 Third Normal Form
- 8.8 BCNF
- 8.9 Answers to Check Your Progress Questions
- 8.10 Summary
- 8.11 Key Words
- 8.12 Self Assessment Questions and Exercises
- 8.13 Further Readings

UNIT 9 JOIN**244-255**

- 9.0 Introduction
- 9.1 Objectives
- 9.2 Lossless Join and Dependency Preservation
 - 9.2.1 Dependency Preservation and Multi-Valued Dependencies
- 9.3 Schema Refinement in Database Design
- 9.4 Fourth Normal Form (4NF)
- 9.5 Answers to Check Your Progress Questions
- 9.6 Summary
- 9.7 Key Words
- 9.8 Self Assessment Questions and Exercises
- 9.9 Further Readings

BLOCK IV: TRANSACTION**UNIT 10 BASIC CONCEPT OF TRANSACTION****256-281**

- 10.0 Introduction
- 10.1 Objectives
- 10.2 Transaction Management
 - 10.2.1 Transaction
 - 10.2.2 Transaction Processing Steps
 - 10.2.3 Interleaved vs Simultaneous vs Serial Transaction
 - 10.2.4 Transaction Action
 - 10.2.5 Transaction States and Additional Operations
 - 10.2.6 Concept of System Log
 - 10.2.7 Commit Point of a Transaction
 - 10.2.8 Acid Properties of a Transaction
 - 10.2.9 Concurrent Execution of Transactions
 - 10.2.10 Motivation for Concurrent Execution of Transactions
- 10.3 Concurrency Control
 - 10.3.1 The Lost Update Problem
 - 10.3.2 Uncommitted Dependency—The Dirty Read Problem
 - 10.3.3 Unrepeatable Read or Inconsistent Retrievals Problem
 - 10.3.4 Phantom Reads; 10.3.5 Schedule
 - 10.3.6 Serializability; 10.3.7 Recoverability
- 10.4 Answers to Check Your Progress Questions
- 10.5 Summary
- 10.6 Key Words
- 10.7 Self Assessment Questions and Exercises
- 10.8 Further Readings

UNIT 11 PROTOCOLS**282-306**

- 11.0 Introduction
- 11.1 Objectives
- 11.2 Concurrency Control Mechanism and Locking Protocols
 - 11.2.1 Locking Protocol for Concurrency
 - 11.2.2 Other Concurrency Control Protocols
 - 11.2.3 Two-Phase Locking Protocols
 - 11.2.4 Concurrency Control Protocols based on Timestamp
 - 11.2.5 Concurrency Control using Multi-Versioning
 - 11.2.6 Validation (Optimistic) Concurrency Control Techniques
- 11.3 Multiple Granularity Locking Protocol
- 11.4 Answers to Check Your Progress Questions
- 11.5 Summary

- 11.6 Key Words
11.7 Self Assessment Questions and Exercises
11.8 Further Readings

UNIT 12 RECOVERY AND ATOMICITY

307-324

- 12.0 Introduction
12.1 Objectives
12.2 Types of Failure and Storage Structure
12.3 Database Recovery Techniques
12.4 Log Based Recovery
12.5 Recovery with Concurrent Transactions
12.6 Buffer Management
12.7 Failure with Loss of Non Volatile Storage
12.8 Advanced Recovery Techniques in a Database
12.9 Remote Backup Systems
12.10 Answers to Check Your Progress Questions
12.11 Summary
12.12 Key Words
12.13 Self Assessment Questions and Exercises
12.14 Further Readings

BLOCK V: STORAGE

UNIT 13 DATA ON EXTERNAL STORAGE

325-344

- 13.0 Introduction
13.1 Objectives
13.2 File Organization and Indexing
 13.2.1 Indexed Files; 13.2.2 Single Level Ordered Index
 13.2.3 Primary Index; 13.2.4 Clustering Index
 13.2.5 Secondary Index; 13.2.6 Non-Key Secondary Index
 13.2.7 Multi-Level Index
13.3 Index Data Structures
 13.3.1 Hash Based Indexing; 13.3.2 Tree Based Indexing
 13.3.3 Comparison of File Organization
13.4 Answers to Check Your Progress Questions
13.5 Summary
13.6 Key Words
13.7 Self Assessment Questions and Exercises
13.8 Further Readings

UNIT 14 PERFORMANCE TUNING

345-354

- 14.0 Introduction
14.1 Objectives
14.2 Intuitions for Tree Indexes
14.3 B⁺ Tree
 14.3.1 Structure of B⁺ Tree
 14.3.2 B⁺ Trees: A Dynamic Index Structure
14.4 Indexed Sequential Access Method (ISAM)
14.5 Answers to Check Your Progress Questions
14.6 Summary
14.7 Key Words
14.8 Self Assessment Questions and Exercises
14.9 Further Readings

NOTES

*Self-Instructional
Material*

INTRODUCTION

Rapid globalization coupled with the growth of the internet and information technology has led to a complete transformation in the way organizations function today. Organizations require those information systems that would provide them a ‘Competitive Strength’ by handling online operations, controlling operational and transactional applications, and implementing the management control tools. All this demands the Relational DataBase Management System or RDBMS which can serve both the decision support and the transaction processing requirements.

Technically, the present RDBMS handles the distributed heterogeneous data sources, software environments and hardware platforms. Precisely, RDBMS is a DataBase Management System or DBMS which is a digital database based on the relational model of data introduced by E. F. Codd in 1970. SQL stands for Structured Query Language. It is specific software system uniquely designed for managing data in a Relational DataBase Management System (RDBMS). The prototype for SQL was originally developed by IBM based on E. F. Codd’s paper ‘*A Relational Model of Data for Large Shared Data Banks*’.

The most widely used commercial and open source databases are based on the relational model. Characteristically, a RDBMS is a DBMS in which data is stored in tables and the relationships among the data are also stored in tables. A relation is defined as a set of tuples that have the same attributes. A tuple usually represents an object and information about that object. Objects are typically physical objects or concepts. Relations can be modified using the insert, delete, and update operators. New tuples can supply explicit values or be derived from a query. Similarly, queries identify tuples for updating or deleting. Tuples by definition are unique. If the tuple contains a candidate or primary key then obviously it is unique; however, a primary key need not be defined for a row or record to be a tuple.

This book *Relational Database Management Systems (RDBMS)*, is divided into five blocks, which are further subdivided into fourteen units. The topics discussed include database system applications, view of data, data abstraction, instances and schemas, data models, relational model, database languages (DDL, DML), database access for applications programs, data base users and administrator, transaction management , storage manager, the query processor, database design and ER diagrams, relationships and relationship sets, querying relational data, logical database design, views, relational algebra, joins, relational calculus, SQL queries, outer joins, normal forms, multi valued dependencies, transaction concept, transaction state, implementation of atomicity and durability, serializability, protocols, recovery and atomicity, data on external storage, file organization and indexing, performance tuning, and Indexed Sequential Access Methods (ISAM).

This book follows the Self-Instructional Mode (SIM) wherein each unit begins with an ‘Introduction’ to the topic. The ‘Objectives’ are then outlined before going on to presentation of the detailed content in a simple and structured format. ‘Check Your Progress’ questions are provided at regular intervals to test the student’s understanding of the subject. ‘Answer to Check Your Progress Questions’ a ‘Summary’, a list of ‘Key Words’, and a set of ‘Self-Assessment Questions and Exercises’ are provided at the end of each unit for effective recapitulation.

NOTES

BLOCK - I
INTRODUCTION

**UNIT 1 DATABASE SYSTEM
APPLICATIONS**

Structure

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Database System vs File System
 - 1.2.1 File-Based Systems
 - 1.2.2 Database Management System—A Better Alternative
- 1.3 Basics of Database Management System
 - 1.3.1 Data Abstraction
 - 1.3.2 Logical and Physical View of Data
 - 1.3.3 Schemas, Subschema Instances or State of a Database
- 1.4 Data Models
- 1.5 ER Model
- 1.6 Answers to Check Your Progress Questions
- 1.7 Summary
- 1.8 Key Words
- 1.9 Self-Assessment Questions and Exercises
- 1.10 Further Readings

1.0 INTRODUCTION

The DataBase Management System (DBMS) is the system for the processing and review of data that communicates with end users, applications, and the database itself. In addition, the DBMS programme covers the key facilities offered to manage the database. A “database system” may be considered the sum total of the database, the DBMS and the related applications. The term “Database” is also sometimes used to refer loosely to any database-related DBMS, database system, or programme.

DataBase management systems can be categorised by computer scientists according to the database models that they support. In the 1980s, relational databases became dominant. In a set of tables, these model data as rows and columns, and the vast majority use SQL for writing and querying data. Non-relational databases, referred to as NoSQL because they use various query languages, became popular in the 2000s.

Database means a place where data can be stored in a structured manner. It is a shared collection or batch of data that is logically related. It has various

NOTES

objectives varying from organization to organization. Data is an asset for any enterprise or entity. The basic objective of a database is to provide security, safety and storage to data. In a DataBase Management System (DBMS), all files are integrated into one system, making data management more efficient by providing centralized control on the operational data. A DBMS is requested to carry out various operations, such as insert, delete, update and retrieval, on the database by the user. There are several parts of a DBMS to carry out the requested operations on the database and provide required data to the users.

In this unit, you will study about the basic concepts of DataBase Management System or DBMS applications, database system vs. file system, views of data, data abstraction, instance and schemas, data models and the ER model.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the various advantages of database system over file based system
- Understand the significance of data abstraction
- Discuss the views of data
- Define data models
- Elaborate on the ER model.

1.2 DATABASE SYSTEM VS FILE SYSTEM

In this section, we will discuss the various advantages of database system over file based systems.

1.2.1 File-Based Systems

Conventionally, before the database systems were introduced, data in software systems was maintained using files. In this approach, the needed data is stored locally and programs are developed for each type of application. The following terms are related to a file processing system:

- **Data**, which is raw facts, is provided to the computer in terms of bytes.
- **Data item** is the smallest named unit of data that has meaning in the real world. Examples include employee name, address and employee code. Traditionally, the term ‘data item’ is called **field** in data processing and is the smallest unit of data that has meaning to its users. **Record** is a set of related data items (fields) organized in a well-defined structure treated as a unit by an application program. The structure of a record depends on:
 - o Items to be included: their characteristics, variability, prevalence in file
 - o The intended use(s) of the information: time requirements, overlapping or conflicting requirements

NOTES

- o Update characteristics, for example frequency, growth, deletion and modification, time constraints, and so on
 - o Links to other records/files/systems
 - o Computing environment, such as operating system, languages, hardware, and so on
- **File** is a collection of all occurrences (instances) of similar types of records, i.e., it is a group of records of a single type.

In file-based systems, an organization stores information as groups of records in separate files. These file-based systems consist of a few data files and many application programs. Data files are organized to facilitate access to records and ensure their efficient storage. In a file-based system, there are two types of records—logical records and physical records. A logical record is a collection of related data items treated as a conceptual unit independent of how or where the information is stored. A physical record is a contiguous area of storage space defined by the characteristics of storage devices and operating systems, and includes all the bytes which will be read/written by a single command to the I/O (Input Output) device. A file processing system relies on the basic access methods of the host operating system for data management.

In a file processing system, it is possible for the application program to only make a request for data from the operating system by demanding a specific logical record. The relationship between physical and logical records, and the location of the physical records in a specific file is kept track of by the operating system. Location of fields within the logical records is an activity that is taken care of by the application program.

In a file processing system, a program takes less time to execute than an equivalent program written in a high-level language as algorithms for sort, merge and report generation are already in-built in the file management software. Besides, the cost of software development is less.

Such file-based approaches provide an improved automated data processing than earlier manual paper record-based systems. However, in view of demand for increased efficiency and speed, a file processing system suffers some significant disadvantages, which are as follows:

- **Data Redundancy and Inconsistency:** A major drawback in the traditional file system environment is non-sharing of data. It means if different systems of an organization are using some common data, even then instead of storing it once and sharing it, each system stores data in separate files. Often, the same information is stored in more than one file. This redundancy in storing the same data in multiple places leads to several problems. First, this leads to the **wastage of storage space** and poses a serious problem if the file processing system has to manage a large amount of data. Second, it leads to the **duplication of effort**, as one needs to perform several updates.

NOTES

Third, **data inconsistency** leads to a number of problems, including loss of information and incorrect results.

- **Difficulty in Accessing Data:** A file-based system does not provide users with ad hoc information requests as most of the information retrieval possibilities are implemented based on pre-determined requests for data. In today's competitive and fast business environment, other than such regularly scheduled requests, there is also a need for responding to unexpected queries. A new application program often requires an entirely new set of file definitions. Even though an existing file may contain some of the needed data items, the application often requires a number of other data items. As a result, the programmer has to recode definitions of the needed data items from the existing file as well as definitions of all new data items, thereby, requiring excessive programming effort. Therefore, conventional file processing systems do not let needed data to be retrieved conveniently and efficiently.
- **Data Isolation:** Data are scattered in various files, in different formats. In a file processing system, it is difficult to determine relationships between isolated data in order to meet user requirements. To tackle such situations, first, the files have to be evaluated by analysts and programmers to determine the specific data requirement from each file and the relationships between the data. Then applications have to be written in a third generation language for processing and extracting the needed data. Imagine the work involved if data from several files was needed.
- **Program–Data Interdependency:** In a file-based system, data is organized in the form of file and records. Files and records are described by specific formats and access strategy, which are coded into application program by the programmer. Such application programs are data-dependent. Consequently, in such an environment, any change of data structure or format requires appropriate changes to all the concerned application programs. This is something that makes a change very painful or problematic for the designers or developers of the system.
- **Atomicity Problem:** In many applications, which have already been implemented, it is not easy to see to it that the data is restored to the last consistent state following the detection of a failure.
- **Integrity Problem:** Data integrity means correctness or accuracy of data. Integrity ensures accuracy of data. The data values stored in a file system may have to satisfy certain types of constraints. Programmers enforce these types of constraints in the system by adding appropriate code in the various application programs. However, when new constraints are to be added, it is difficult to change the program to enforce them with small effort and time.

NOTES

- **Security Problems:** In conventional systems, applications are developed in an ad hoc manner. At times, different components of the operational data are accessed by different parts of a firm. In an environment of this type, it can be quite difficult to ensure/enforce security.
- **Concurrent Access anomalies:** If the system allows multiple users to update the data simultaneously, the interaction of concurrent updates may result in inconsistent data. In the traditional file system, such concurrent access anomalies cannot be avoided without huge programming effort.

The above-mentioned limitations can be attributed to the following factors:

- Definition of data is embedded in application programs rather than stored separately. In such an environment, any change in data structure or format requires appropriate changes to the application programs.
- There is no control over the access and manipulation of data beyond that imposed by application programs.

1.2.2 Database Management System—A Better Alternative

With a Database Management System or DBMS, as is referred to in short, the scenario is totally different. Programs do not deal with stored data by its location but they are provided with a software by a DBMS. This software allows application programs to deal with data field names irrespective of the location of the fields within the records, the location of the records within a file and the file within a device. In a DBMS, all files are integrated into one system, thus, making data management more efficient by providing centralized control on the operational data. Database management systems are not only used in the commercial applications but also in many of the scientific/engineering applications.

Database

Database means a place where data can be stored in a structured manner. It is a shared collection or batch of data that is logically related, along with their descriptions designed to meet the information requirements of an organization.

A database is a complex data structure. It is stored in a system of mutually dependent files. Those files contain the following information:

1. The set of data available to the user, the so-called ‘end-user data’. Those are the real data, which can be read, connected and modified by the user (if he has the corresponding rights).
2. The so-called ‘metadata’ (the data describing the end-user data). Here, the properties (for example, their type) and the relative relations of the end-user data are described.

NOTES

DataBase Management System (DBMS)

DataBase Management System (DBMS) is a software system that allows users to not only define and create a database but also maintain it and control its access. A database management system can be called a collection of interrelated data (usually called database) and a collection or set of programs that helps in accessing, updating and managing that data (which form part of a database management system).

The primary benefit of using a DBMS is to impose a logical and structured organization on data. A DBMS provides simple mechanisms for processing huge volumes of data because it is optimized for operations of this type. The two basic operations performed by the DBMS are as follows:

- Management of data in the database
- Management of users associated with the database

Management of the data means specifying how data will be stored, structured and accessed in the database. This includes the following:

- **Defining:** Specifying data types and structures, and constraints for data to be stored
- **Constructing:** Storing data in a storage medium
- **Manipulating:** Involves querying, updating and generating reports
- **Sharing:** Allowing multiple users and programs to access data simultaneously

Further, the database management system must offer safety and security of the information stored, in case unauthorized access is attempted or the system crashes. If data is required to be shared among many users, the system must ensure that possible anomalous results are avoided.

Management of database users means managing the users in such a way that they are able to perform any desired operations on the database. A DBMS also ensures that a user cannot perform any operation for which he is not authorized.

In short, a DBMS is a collection of programs performing all necessary actions associated with a database. There are many DBMSs available in the market, such as Access, dBase, FileMaker Pro, Foxpro, ORACLE, DB2, Ingress, Informix, Sybase, and so on.

A database application is a program or a set of programs that interacts with the database at some point in its execution. It is used for performing certain operations on data stored in the database. These operations include inserting data into a database or extracting data from a database based on certain conditions, updating data in a database, producing data as output on any device such as screen, disk or printer.

A database system is a collection of application programs that interacts with the database along with DBMS and database itself (and sometimes the users

who use the system). Database systems are designed in a manner that facilitates the management of huge bodies of information.

A database clearly separates the physical storage of data from its use by an application program to attain program–data interdependence. For using a database system, the user or programmer is unaware of the details of how the data are stored. Data can be changed or updated without making any effect on other components of the system.

1.3 BASICS OF DATABASE MANAGEMENT SYSTEM

In this section, we will discuss the basics of database management system.

1.3.1 Data Abstraction

A DBMS must have some means of representing the data in a way that users can easily understand. A DBMS provides users with the conceptual representation of data. The system hides certain details regarding data storage and maintenance, and data is retrieved efficiently. This is performed by defining levels of abstraction at which the database may be viewed.

1.3.2 Logical and Physical View of Data

Separating the logical and physical structures of data clearly is one of the main features of the database approach. The term ‘logical structure’ indicates the manner in which the programmers view it, whereas the physical structure refers to the manner in which data is actually stored on the storage medium.

A **logical view** of data expresses the way a user thinks about data. Usually, it is expressed in terms of a **data model**.

A **physical view** of data is the way data are handled at low level, i.e., the storage and retrieval of it. Specifically stated, it is expressed in terms of specific locations on storage devices plus techniques used to access it.

A set of logical constructs that can help describe the structure of a database, that is, its data types, constraints and relationships, is referred to as a *data model*. It is also a set of basic operations that specify updates and retrievals on the database.

A data model is used to refer to a set of general principles for handling data (Tsitchizris and Lochovsky, 1982). The set of principles that defines a data model may be divided into the following three major parts:

- **Data Definition:** A set of principles concerned with how data is structured
- **Data Manipulation:** A set of principles concerned with how data is operated upon

NOTES

NOTES

- **Data Integrity:** A set of principles concerned with determining which states are valid for a database

1.3.3 Schemas, Subschema Instances or State of a Database

The overall description of a database is called **database schema**, which is specified during database design and is expected not to be changed very frequently. The values of a data item can be fitted into a framework. A database schema includes such information as:

- Characteristics of data items
- Logical structure and relationship among those data items
- Format for storage representation
- Integrity parameters, authorization and backup policies

A **subschema** is its proper subset designed to support ‘views’ belonging to different classes of users in order to hide or protect information. It refers to the same view as schema but for the data types and record types, which are used in a particular application or by a particular user.

Database changes over time, as information is inserted, deleted or updated. The collection of data or information stored in the database at a particular moment of time is called an **instance, state or a snapshot of the database**. Database schema and database state are two different things. While a new database is being defined, only the database schema is specified to the DBMS. The existing state of the database, with no data, is the *empty state*. We get the initial state of the database when data in the database is first inserted. The DBMS is responsible for ensuring that every state is a **valid state** satisfying the structure and constraints specified in the schema. Sometimes, the schema is referred to as the **intension**, and a database state as an **extension** of that schema.

1.4 DATA MODELS

A characteristic of the database approach is that it provides a level of data abstraction by hiding superfluous details, while highlighting the details that are of interest to the application. A data model is a mechanism to provide this abstraction for different database applications.

A Database Management System or DBMS can choose from several approaches to manage data. Each approach constitutes a data model, which can be defined as an integrated collection of concepts or tools used to describe and manipulate data, relationships between data and semantics and constraints on data in an organization. Constraints imply a set of rules that imposes restriction on data in a database. The data model provides the necessary means to achieve the abstraction. Most DBMS provide mechanisms to structure data in the database being modelled, allow the set of operations to be defined on them and enforce

certain constraints to maintain the integrity and security of data. Therefore, a data model is a mechanism for specifying the schema of a database.

Data model is a collection of conceptual tools for describing data, relationship between data and consistency constraints. Figure 1.1 illustrates the structure of data model.

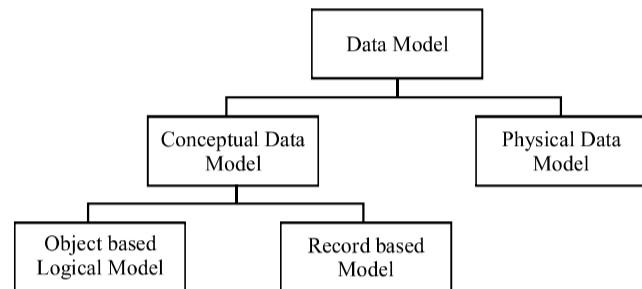


Fig. 1.1 Structure of Data Model

Data models help in describing the structure of data at the logical level. Data model describes the structure of the database. It is a set of conceptual constructs available for defining a schema. The data model is a language for describing the data and database, and it may consist of abstract concepts which must be translated by the designer.

Advantages of a Data Model

The advantages of a data model are as follows:

- It ensures that all data objects provided by the functional team are represented with accuracy and in complete form.
- It contains enough details to be used by the technical team who build the physical database.
- It can be used to communicate information within and across business organizations.

Traditional Data Model

A data model is primarily concerned with the logical database design and the rules for it. To use a data model, one needs only *paper and pencil* on which to put one's thoughts about data and their usage. A data model should provide a set of formalisms to organize the thoughts and put it on paper in an abstract notation.

Data models may be categorized into the following types:

• Physical Data Model

Physical data model provides concepts that describe the details of how data is stored in a computer. These concepts are generally meant for specialists and not end-users.

NOTES

NOTES

Physical data models define the record structure, file structure and ordering and accessing paths (structure that makes the search for a particular records efficient). These models use files, indices, pointers, links, records, fields, etc. Professional programmers use these types of models.

• Logical/Conceptual Data Model

It provides concepts regarding users' perception of data. Logical data models use objects, entities, relationships and attributes. They hide details about data storage/retrieval. They target end-users.

Logical data models are classified into the following two categories:

- Object-based logical models
- Record-based logical models

In an object-based data model, data is viewed as sets of entities (objects) that represent things in the real-world. Entities (objects) in the system are distinct and uniquely identifiable. New types of objects (entities) can be constructed from old types. There are two types of common object-based logical data models:

- Entity–Relationship (E–R) model
- Object Oriented (OO) model

In a record-based logical data model, data is viewed as fixed-format records of various types, for example, one record type for customers, another for checking accounts, etc. There are mainly three types of record-based logical models:

- Hierarchical model
- Network model
- Relational model

As already mentioned, several popular data models have been developed over the years. However, the relational data model is currently the most popular and most widely used one.

• Representational Data Model

It provides concepts that can be easily understood by end-users, but it is not dependent on the way data is organized.

Representational data models are used most frequently in commercial DBMS. They include relational data models and legacy models, such as network and hierarchical models. Each data model has its own strengths and weaknesses in terms of commercial applications.

Data modelling tools are the only way by which we can create powerful data models.

There are innumerable data modelling tools that convert business requirements into a logical data model and a logical data model into a physical

data model. These tools, in a physical data model, can be told to generate Structured Query Language or SQL codes for database creation. Some of the popular data modelling tools are given in Table 1.1.

Table 1.1 Popular Data Modelling Tools

Tool Name	Company Name
Erwin	Computer Associates
Embarcadero	Embarcadero Technologies
Rational Rose	IBM Corporation
Power Designer	Sybase Corporation
Oracle Designer	Oracle Corporation
Xcase	RESolution LTD

NOTES

The record-based logical data models are discussed in detail as follows:

1.5 ER MODEL

In a particular area of science, an Entity-Relationship model (or ER model) describes interrelated items of interest. A simple ER model consists of categories of entities (which define items of interest) and defines relationships between entities that may exist (instances of those entity types).

An ER model is generally developed in software engineering to reflect items an organisation needs to remember in order to execute business processes. The ER model thus becomes an abstract data model, which describes a system of data or information that can be applied in a database, usually a relational database.

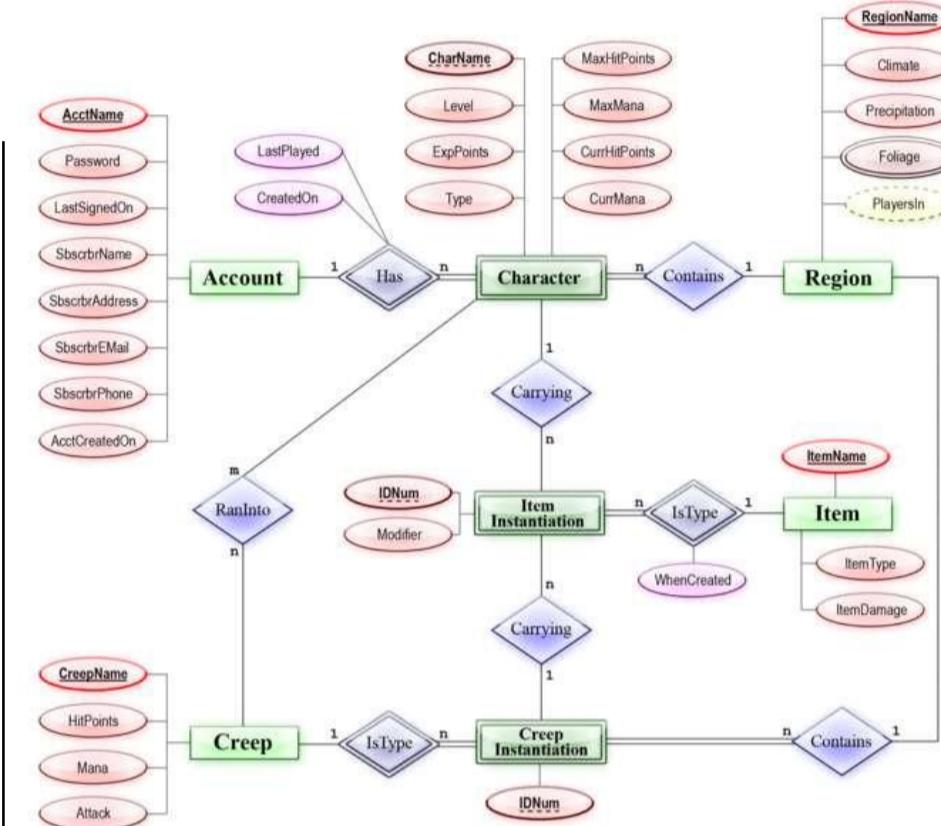
The Entity-Relationship (E-R) model facilitates database design by enabling the designer to express the logical properties of the database in an enterprise schema. Identification of real-world objects referred to as entities, forms the basis of this model. These entities are described by their attributes and are connected by relationships among them. The E-R model has its own set of symbols for drawing the E-R diagram which depicts the logical model of the database.

Regardless of the database system selected, the enterprise schema will be valid. It is capable of remaining constant even if the DBMS is modified.

The E-R model enables you to express restrictions or constraints on the entities or relationships.

Entity-relationship modelling was developed by Peter Chen for database and design and published in a 1976 paper with variants of the previously existing idea. Some ER models display super and subtype entities linked by generalization-specialization relationships, and it is also possible to use an ER model in the domain-specific ontology specification.

NOTES



Check Your Progress

1. What is a file?
2. Define the term database.
3. What do you understand by abstraction?
4. What is a data model?
5. What is physical data model.

1.6. ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. File is a collection of all occurrences (instances) of similar types of records, i.e., it is a group of records of a single type.
2. Database means a place where data can be stored in a structured manner. It is a shared collection or batch of data that is logically related, along with their descriptions designed to meet the information requirements of an organization.

NOTES

3. The system hides certain details regarding data storage and maintenance, and data is retrieved efficiently. This is performed by defining levels of abstraction at which the database may be viewed.
4. A characteristic of the database approach is that it provides a level of data abstraction by hiding superfluous details, while highlighting the details that are of interest to the application. A data model is a mechanism to provide this abstraction for different database applications.
5. Physical data model provides concepts that describe the details of how data is stored in a computer. These concepts are generally meant for specialists and not end-users.

Physical data models define the record structure, file structure and ordering and accessing paths (structure that makes the search for a particular records efficient). These models use files, indices, pointers, links, records, fields, etc. Professional programmers use these types of models.

1.7 SUMMARY

- Conventionally, before the database systems were introduced, data in software systems was maintained using files.
- File is a collection of all occurrences (instances) of similar types of records, i.e., it is a group of records of a single type.
- In a file-based system, there are two types of records—logical records and physical records. A logical record is a collection of related data items treated as a conceptual unit independent of how or where the information is stored. A physical record is a contiguous area of storage space defined by the characteristics of storage devices and operating systems, and includes all the bytes which will be read/written by a single command to the I/O (Input Output) device.
- Database means a place where data can be stored in a structured manner. It is a shared collection or batch of data that is logically related, along with their descriptions designed to meet the information requirements of an organization.
- Database Management System (DBMS) is a software system that allows users to not only define and create a database but also maintain it and control its access.
- A database application is a program or a set of programs that interacts with the database at some point in its execution.
- The system hides certain details regarding data storage and maintenance, and data is retrieved efficiently. This is performed by defining levels of abstraction at which the database may be viewed.

NOTES

- A logical view of data expresses the way a user thinks about data. Usually, it is expressed in terms of a data model.
- A physical view of data is the way data are handled at low level, i.e., the storage and retrieval of it.
- The overall description of a database is called database schema, which is specified during database design and is expected not to be changed very frequently.
- A characteristic of the database approach is that it provides a level of data abstraction by hiding superfluous details, while highlighting the details that are of interest to the application. A data model is a mechanism to provide this abstraction for different database applications.
- The Entity–Relationship (E–R) model facilitates database design by enabling the designer to express the logical properties of the database in an enterprise schema. Identification of real-world objects referred to as entities, forms the basis of this model. These entities are described by their attributes and are connected by relationships among them. The E–R model has its own set of symbols f

1.8 KEY WORDS

- **DataBase Management System (DBMS):** It is a software system that allows users to not only define and create a database but also maintain it and control its access.
- **Database Schema:** The overall description of a database which is specified during database design and is expected not to be changed very frequently.
- **Instance, State or a Snapshot of the Database:** The collection of data or information stored in the database at a particular moment of time.
- **ER Model:** The Entity–Relationship (E–R) model facilitates database design by enabling the designer to express the logical properties of the database in an enterprise schema.

1.9 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What are the disadvantages of file processing system?
2. What is data abstraction?
3. What are the various views of data?

4. Define schema and instances of a database.
5. What is ER model?

*Database System
Applications*

Long-Answer Questions

1. Explain the advantages of database system over file processing system.
2. What are the different types of data models? Explain.
3. Explain the basics of Data Base Management System (DBMS).

NOTES

1.10 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

UNIT 2 MODELS

NOTES

Structure

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Types of Data Models
 - 2.2.1 Hierarchical Data Model
 - 2.2.2 Relational Data Model
 - 2.2.3 Network Data Model
 - 2.2.4 Comparison of three Models
- 2.3 Database Languages
- 2.4 Database Access for Application Programs
- 2.5 DBMS Users
- 2.6 Transaction Management
- 2.7 Database Systems Structure
 - 2.7.1 DBMS Structure
 - 2.7.2 Applications of DBMS
 - 2.7.3 Disadvantages of DBMS
 - 2.7.4 Storage Manager
 - 2.7.5 Query Processor
- 2.8 Answers to Check Your Progress Questions
- 2.9 Summary
- 2.10 Key Words
- 2.11 Self Assessment Questions and Exercises
- 2.12 Further Readings

2.0 INTRODUCTION

The development process of a database, including the relationships and constraints that dictate how data can be processed and accessed, is illustrated by a database model. The design of individual database models is based on the rules and principles of whichever larger data model is adopted by the designers. An accompanying database diagram will represent most of the data models.

Different models are used at various stages of the design process of the database. High-level conceptual data models are best used to map data relationships in ways that people interpret that knowledge. On the other hand, logical models based on records more closely represent the ways in which data is stored on the server.

A database model uses database which determines in which manner data can be stored, organized and manipulated in a database system. It then defines the infrastructure offered by a particular database system. In a hierarchical model, data is organized into a tree like structure which provides link in each record and a sort field to keep the records in a particular order at the same level list. The basic data structure of the relational model is represented in terms of rows, also known

as tuples, and columns. Thus, ‘relation’ in the ‘relational database’ refers to the various tables in the database representing a relation as a set of tuples.

Models

In this unit, you will study about the relational model, other models, traditional database model over relational database model, database languages (DDL and DML), database access for application programs, database users and administrator, transaction management, database system structure, storage manager and query processor.

NOTES

2.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the various types of data models
- Discuss the types of database languages and DBMS users
- Define about the transaction management.
- Understand the database system structure

2.2 TYPES OF DATA MODELS

Following are the different types of data models.

2.2.1 Hierarchical Data Model

Data models can be defined as a collection of various concepts used to describe the structure of a database. Implementing a data model includes specifying data types, relationships among data types and constraints on the data. In the hierarchical model, also called hierarchical schema, data is organized in the form of a tree structure. The hierarchical model supports the concept of data independence. Data independence is the ability to change the representation of data at one level of a database system without the compulsion of changing the data representation at the next higher level.

The hierarchical model uses two types of data structures, records and parent-child relationship to define the data and relationship among data. Records can be defined as a set of field values, which are used to provide information about an entity. An entity is a collection of objects in a database, which can be described by using a set of attributes. Records that have the same type can be easily grouped together to form a record type and assigned a name. The structure of a record type can be defined by using a collection of named fields or data items. Each data item or field has a certain data type such as character, float or integer. The Parent-Child Relationship (PCR) can be defined as a 1:N relationship between two different record types. The record type on the 1-side is called the parent record type and the record type on the N-side is called the child record type. Occurrence of the PCR type, also called instance, consists of one record of the

parent record type and a number of records of the child record type. Figure 2.1 shows an example of 1:N relationship between a finance department and its employees.

NOTES

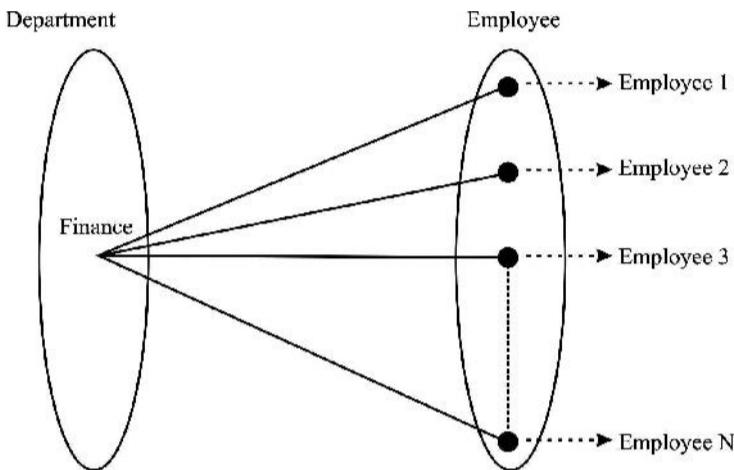


Fig. 2.1 1:N Relationship between a Finance Department and Its Employees

Hierarchical schema consists of a number of record types and PCR types. In the hierarchical schema, record types are represented by rectangular boxes and PCR types are represented by the lines, which are used to connect a parent record type to a child record type. Figure 2.2 represents a hierarchical schema, which has three record types and two PCR types. Department, Employee and Project are the record types in Figure 2.2.

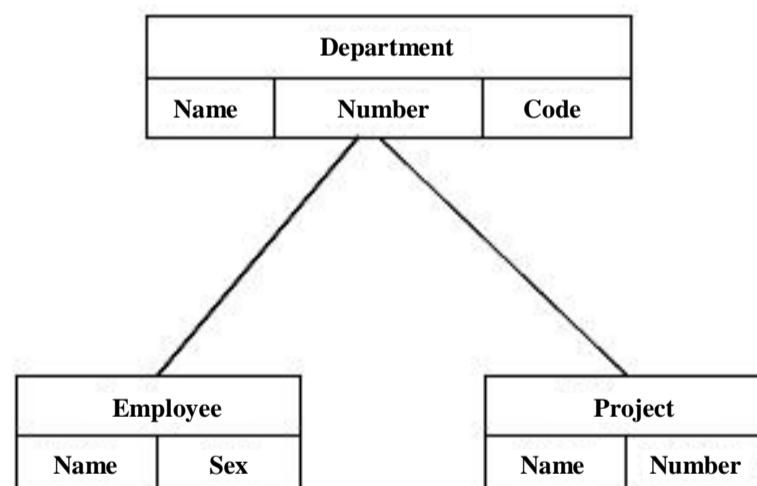


Fig. 2.2 Hierarchical Schema

Construct of Hierarchical Data Model

Each record type can have a set of data items or fields. For example, the record type Department can have department name, department number and department

code as the fields or data items. PCR type can be represented by listing pair in parentheses. For example, in Figure 2.2, there are two PCR types, which can be represented as (Department, Employee) and (Department, Project). In the Figure 2.3, each occurrence of the (Department, Employee) PCR type relates one department record to the records of many employees, who work in that department. The occurrence of (Department, Project) PCR type relates a department record to the records of projects controlled by that department. Figure 2.3 represents the tree-like structure of the hierarchical schema shown in Figure 2.2.

NOTES

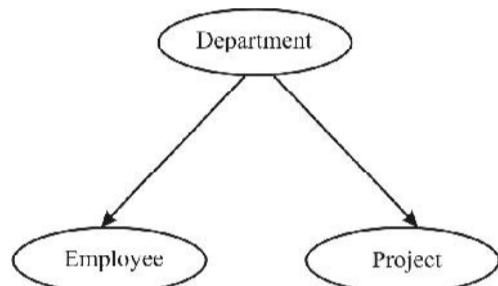


Fig. 2.3 Tree Representation of Hierarchical Schema

In a tree-like structure, a record type is represented by the node of the tree and PCR type is represented by the arc of the tree. The following are the properties of the hierarchical schema, which contains the numbers of record types and PCR types:

- One record type, called the root of the hierarchical schema, does not participate as a child record type in any PCR type.
- In the hierarchical model, each record can have only one parent record but can have many child records.
- Every record type except the root participates as a child record type in only one PCR type.
- A record type can participate as a parent record type in a number of PCR types.
- A record type which does not participate as a parent record type in any PCR type is called leaf node in hierarchical schema.
- If a record type participates as a parent node in more than one PCR type, then its child record types must be in a left to right ordered sequence.

The advantages of the hierarchical data model are as follows:

- It is simple to construct and operate on data in the hierarchical model.
- It involves hierarchically organized domains, such as product info in manufacturing and employee information in organization.
- It uses constructs, such as GET, GET UNIQUE and GET NEXT.

NOTES

The disadvantages of the hierarchical data model are as follows:

- It requires the navigational and procedural processing of data.
- It provides less scope of query optimization.

2.2.2 Relational Data Model

In 1970, E. F. Codd formally introduced the relational model. Predicate logic and set theory form the basis of the relational model for database management. This model provides a simple, yet rigorously defined, concept of the manner in which data is perceived by users.

Strengths

Some of the strengths of relational models are given in Table 2.1.

Table 2.1 Strengths of Relational Model

Simplicity	End-users' requests are formulated in terms of the information content. These requests do not reflect any complexities due to system-oriented aspects. A relational data model is what one sees, and not necessarily, what will be implemented physically.
Non-procedural Request	Requests focus on 'What is to be done' rather than 'How it is done'.
Data Independence	Removes the details of storage structure and access strategy from the user interface. Structural flexibility is provided by relational databases; It is easier to maintain applications written for those databases. It also allows retrieval of combinations of data that may not have been anticipated as required or needed when the database was designed. To be able to make use of this characteristic, however, the design of the relations must be complete and accurate.
Mathematical Backbone	It is based on a formal theoretical model and is not only studied extensively but proven in practice. Almost every known aspect of it is actually proven in the form of mathematical theorems.

Components

The main principle of the relational model is the information principle—all information is represented by data values in relations. The three components—structural, manipulative and integrity—make up the relational model. These components are described as follows:

- The structural component is concerned with how data is represented. A set of relations represents the conceptual view of the database.
- The manipulative component is concerned with how data is operated upon. It comprises a set of high level operations, which produces whole tables and acts upon them.
- The integrity component is concerned with determining which states are valid for a database. It is a cluster of rules for the maintenance of the integrity of the database.

The relational data model represents a database as a collection of relation values or relations where a relation resembles a two-dimensional table of values presented as rows and columns. A relation has a heading, which is a tuple of attribute names,

and a body, which is a set of tuples having the same heading. The heading of a relation is also referred to as **relation schema** or **intension** and the body of the relation is referred to as **extension**. Thus, the intension of the EMP relation would be:

EMP (EmpCode, EmpName, Salary, Date_of_join, Deptno)

Intensions provide a convenient way of describing a database depicting the schema of the database. An extension refers to the rows of data values.

Relational Terminologies

Domain: A domain is the set of defined atomic values for an attribute. It is a pool of values from which specific relations draw their actual values. A domain is specified in terms of data type (possibly system-defined or user-defined) and, optionally, in terms of size, range, etc.

Attribute: Attribute is the name of a role played by a domain in the relation. Each attribute A_i is defined over a domain D_i (the set of values that A_i can take on) and is the name of a feature of the real world entity or relationship that the relation is representing. Formally, it is an ordered pair (N, D) , where N is the name of the attribute and D is the domain that the named attribute represents, e.g., Emp Name, M GHOSH.

Relational Schema: A relational schema is made up of a relation name and a list of attributes. A relation schema R is denoted by $R(A_1, A_2, \dots, A_n)$, where R is the name of the relation and A_1, A_2, \dots, A_n is a list of attributes. A relation schema is used to describe a relation.

Relational Database Schema: It is a set of relation schema, each with a distinct name. If R_1, R_2, \dots, R_n are a set of relation schemas, then we can write the relational database schema R as $R = \{R_1, R_2, \dots, R_n\}$.

Relation: A relation (or relation state) r of the relation schema $R(A_1, \dots, A_n)$ is a set of n -tuples,

$$\text{i.e., } r = \{t_1, t_2, \dots, t_n\}.$$

Tuple: Each row in a relation is a set of related data values and is called a tuple. Formally, an **n -tuple**, is an ordered list of values $t = \langle v_1, \dots, v_n \rangle$ where each v_i is an element of D_i where D_i is the domain of A_i .

Degree (of a Relation Schema): The degree of the relation is the number of attributes (n).

Cardinality (of a Relation State): The cardinality m is the number of tuples in a particular relation state.

Formally, a **relation** is defined as the subset of the subset of the Cartesian product of domains. In order to do so, first we define the Cartesian product of two sets and then the expanded Cartesian product. The Cartesian product of two sets A and B , denoted by $A \times B$ is:

$$A \times B = \{(a, b) : a \in A \text{ and } b \in B\}$$

NOTES

NOTES

The expanded Cartesian product of n sets A_1, A_2, \dots, A_n is defined by,

$(A_1, A_2, \dots, A_n) = \{(a_1, a_2, \dots, a_n) : a_j \in A_j, 1 \leq j \leq n\}$. The element (a_1, a_2, \dots, a_n) is called an n-tuple.

A relation $r(R)$ is a subset of the Cartesian product of the domains $D(A_i)$ that define R . Therefore,

$$r(R) \subseteq D(A_1) \times D(A_2) \times \dots \times D(A_n).$$

A relation state r of the relation schema $R(A_1, \dots, A_n)$ is a set of n-tuples, i.e., $r = \{t_1, t_2, \dots, t_m\}$.

Construct of Relational Data Model-Set Heading

Let us consider the following relational schema EMPLOYEE describing the employee information of a company. The relation EMPLOYEE can be shown as follows:

EMPLOYEE	ECODE	ENAME	ADDRESS	DT_JN	BASIC	DEPT
Tuple	E01	M GHOSH	107 PRATAP GARH	10-JAN-85	6000	PROJECT

Fig. 2.4 Relational Schema EMPLOYEE

Characteristics of Relations

Following are the characteristics of relations:

- A relation has a name that is distinct from all other relation names in the relation schema.
- Each attribute value of a tuple is atomic. Hence, composite and multi-valued attributes are not allowed in a relation.

According to this property, repeating groups or arrays should not form columns in a relational table. Such tables are said to be in the 'First Normal Form' (1NF). The foundation of the relational model is the atomic value property of relational tables and there it is important. The primary advantage of the one value property is that it makes the data manipulation logic simple.

- A distinct name is given to each attribute.
- In a relation, all the values of an attribute come from the same domain.
- There is no semantic significance in the order of attributes as long as correspondence between the attributes and their values in the relation is maintained.

NOTES

This property is derived from the fact that the heading of the relation is a mathematical set (of attribute). According to this property, the ordering of the columns in the relational table is meaningless. Columns can be retrieved in various sequences and in any order. The advantage of this property is that it allows multiple users to share the same table without any concern for the manner in which it is organized. It also allows the physical structure of the database to alter without any impact on the relational tables.

- Each tuple is distinct; there are no duplicate tuples.

This property is based on the fact that the body of the relation is a mathematical set (of tuples). In mathematics, sets do not include duplicate elements. Therefore, theoretically, this property makes sure that two rows are never identical in a relational table; the values of at least one column, or set of columns, uniquely identify each row in the table. Such columns are referred to as primary keys.

- The order of tuples has no semantic significance.

This property is based on the fact that in mathematics, a set is not ordered. Since the body of the relation is represented following the set theory, this property is analogous to the one mentioned earlier. However, it applies to rows rather than columns. The primary advantage is that in a relational table, the rows are retrievable in varying sequences and orders. Addition of information to a relational table becomes simple and does not impact the existing queries.

- Derived attributes are not captured in a relation schema.

In an SQL schema, only two types of relation schema may be defined, that is VIEWS and BASE RELATION. These are called NAMED RELATIONS. Other tables, called UNNAMED RELATIONS, may be derived from these using relational operations, such as join and projection.

- **Base Relation:** This implies a named relation which corresponds to an entity in the conceptual schema whose tuples are physically stored in the database. A relational system must provide a means for creating the base relations (specifically tables) in the first place. In SQL, this function is performed by the CREATE TABLE command. Base tables have independent existence.
- **View:** It is a virtual or derived relation. It is a named relation that does not necessarily exist in its own rights, but may be dynamically derived from one or more base relations. Its purposes may be cited as follows:
 - o It provides a powerful and flexible security by hiding parts of the database from certain users.
 - o It permits users to access data in a way that is customized to their needs so that the same data can be seen in different ways at the same time.
 - o It can simplify complex operations on the base relations.

NOTES

Disadvantages of Relational Model

As already mentioned, of all data models, the relational model is the most dominant. However, it suffers from certain limitations. Like the hierarchical and network models, the relational model has been developed to meet the requirements of business information processing. While applying the relational model to the application areas, such as Computer Aided Design (CAD), simulation and image processing, many shortcomings have been noticed in this model. It is being suggested that a more sophisticated data model should be developed. The various shortcomings of this model may be discussed as follows:

- **Difficulty in Modelling Complex Objects:** In certain circumstances, the strength of the relational model—its simple tabular data-model—becomes its weakness. The reason for this is that compressing some of the complex relationships, that exist in the real world, into tables is a cumbersome exercise. Thus, the modelling of such complex, nested entities in a relational data model is not easy.
- **Lack of Semantic Knowledge:** ‘Semantic knowledge’ refers to knowledge about the meaning of data, i.e., how to interpret data, and the legitimate processes for which the data may be used. In the relational database model, this knowledge is scarce. Only the domain, entity and referential integrity rules possess semantic information. Moreover, many Relational Database Management System or RDBMS do not fully support the domain concept. In such circumstances, application programmers are left with no other option but to compensate for the inability of the basic relational model to carry semantic knowledge, by building such knowledge into application programs.
- **Limited Data Types:** This limitation is also related to the two limitations just mentioned. An RDBMS can recognize only simple atomic data types, such as integers, characters, etc. It is one of the most critical disadvantages of RDBMS.

Properties of Relational Model

Relational Database Schema: A relation schema R , denoted by $R(A_1, A_2, \dots, A_n)$, is made up of a relation name R and a list of attributes A_1, A_2, \dots, A_n . Each attribute A_i is the name of a role played by some domain D in the relation schema R . D is called the domain of A_i and is denoted by $\text{dom}(A_i)$. A relation schema is used to describe a relation; R is called the name of this relation. The degree or arity of a relation is the number of attributes n of its relation schema. A relation schema is sometimes called a relation scheme.

A relation or relation state r of the relation schema $R(A_1, A_2, \dots, A_n)$, also denoted by $r(R)$ is a set of n -tuple t is an ordered list of n values $t = \langle v_1, v_2, \dots, v_n \rangle$, where each value v_i , $1 \leq i \leq n$, is an element of $\text{dom}(A_i)$ or is a special Null value. The i th value in tuple t which corresponds to the attribute A_i is referred to

as $t[A_i]$. The terms ‘relation’ for the schema R and relation extension for a relation state $r(R)$ are commonly used. The properties of relation are as follows:

- Relational schemas have named and typed attributes and the relational instances are finite.
- Relation model is based on (finite) set theory in which attribute ordering is not strictly necessary.
- All attribute values are atomic in which degree (arity) supports of attributes in schema and cardinality supports of tuples in instance.

Relational model has some important properties. Each relation has a name, cardinality and a degree. Some of the properties are described below:

- **Name:** The first property is that a relation has a name which identifies it, for example the Student relation.
- **Cardinality:** The second property of a relation is its cardinality. This refers to the number of tuples in the relation.
- **Degree:** The third and final property of a relation is its degree. The degree of a relation refers to the number of attributes in each tuple.
- **Supports Relations:** A database consists of sets of records or (equivalently) sets of tuples (relations) or (equivalently) tables; no links allowed in the database. Every tuple is an element of exactly one relation and is identified uniquely by a primary key.

An intuitive, predictable and orderly approach for the organization, manipulation and viewing of data is presented by the relational model.

Relational data are comprised of relations. A *relation* (or relational table) can be defined as a two dimensional table that possesses certain special properties. It contains a random number of rows and a set of named columns. The former are known as *tuples* or *records* and the latter are referred to as *attributes* or *fields*. All attributes are related to pools of values known as *domains* and draw a value from them. More than a single attribute of a table can be associated with a particular domain. The following are the six primary properties that a table must satisfy to be categorized as relational:

- **Each Column has a Distinct Identity:** Columns are identified by their names and not positions. Every column in the table must have a unique name.
- **No Importance is Attached to the Order of the Rows:** Any row can be retrieved by the user in any order.
- **No Importance is Attached to the Order of the Attributes:** Any column can be retrieved by the user in any order.
- **No Two Rows should be Identical:** The identity of each row in the table must be distinct. The values in a specific column known as the *primary key* are responsible for ensuring this uniqueness.

NOTES

NOTES

- The Attribute Entries are of the Same Data Type or Kind:** All the entries in a column should be of a similar data type or kind. A column that is meant for storing emp_age of an employee must not be storing the name. For instance, all the values of an attribute MGR in a table EMPLOYEE must contain only digits and not characters.
- The Attribute Entries are Single Valued or Atomic:** All the entries in all the column or row positions of a table should be single valued. In other words, no column should contain repeating groups. For instance, an attribute in an EMPL table called Emp_name, should not contain a value like Sunil John as it is not an atomic or a single value. Instead, the attribute Emp_name can be decomposed to First_name, Middle_name and the Last_name. Now, each of these attributes will contain a single value.

The relational database model is based on set theory of mathematics. Relation is a central concept in relational model that is borrowed from the concept of set theory. Such concepts are widely applied in designing the relational database model. Viewing from the application side, this model is based on first-order predicate logic. E.F. Codd first proposed the concept of this model in the year 1969.

Access to data in relational model is made via relations. Relations storing data are known as base relations; ‘table’ is an implementation of this base relation. There are other relations too that are derived using operations such as selection, projection, union and intersection, and hence, they are known as ‘derived relations’. Figure 2.5 shows a relation (table) derived from two tables. The first table shows few activities coded for easy structuring of work. Every activity has few routes which started on some specific date. This is shown in another table, showing date, activity code and route number. Here, overlaying is done by following two routes I – 19 and I – 12. The third table is a result of query that displays the required route number, activity code and the date.

Relational Model		
Activity Code	Activity name	
230	Patchworking	
240	Overlaying	
250	Sealing of cracks	

Activity Code	Date	Route number
240	10/01/02	I-19
240	08/02/02	I-12

Date	Activity Code	Route number
10/01/02	240	I-19
15/01/02	230	I-40
08/02/02	240	I-12

Fig. 2.5 The Relational Model

Thus, the relational model describes database as a set of predicates having a finite set of variables, known as predicate variables. The model also describes constraints for possible values of their combinations. Database content is thought to be a logical model of the database, containing a set of relations, with one relation for every predicate variable, satisfying all predicates. Queries from such a database are made as predicates and hence, request for information is also a predicate.

The two principal rules for the relational model are known as entity integrity and referential integrity.

In a relational model, data is presented by mathematical n -ary relation that is a subset of the Cartesian product that has n number of domains. While dealing with a mathematical model, such data is analysed with two-valued predicate logic in which any proposition has only two possible outcomes, either *true* or *false*. There is no place for a third outcome such as ‘not applicable’, or ‘not available’. Some favour use of such a two-valued logic for relational model, whereas others favour three-valued logic and this will be still called relational. In such a model, Relational Algebra or relational calculus is used for operation on data.

The relational model is consistent, which is achieved by using constraints while designing database. This design is also known as logical schema.

The various concepts associated with the relational model is shown in Figure 2.6. Attributes in a relational model are put as columns of a table in this model. The name of the relation is shown as R , but it can assume any name according to the context. For example, if we are dealing with details of employees, we may write the relation name as Emp or EMP, as may appeal to one who designs the database. Row is known as tuple. A relation is stored as a table. Attributes may be in any order in the table. A set of attributes is heading entries which are put under rows and columns for the table body. An attribute must have some specific values.

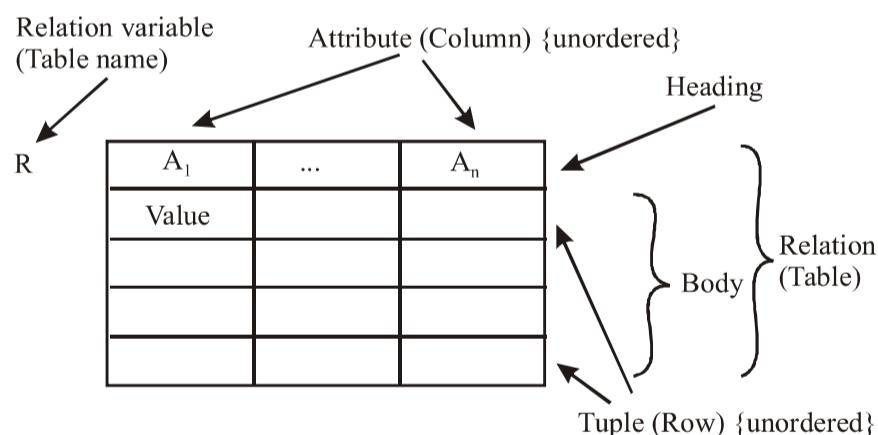


Fig. 2.6 Concepts of the Relational Model

Thus, a relation has a tabular structure with definition for every column and data put in this structure. The structure is defined by the heading and the data is entered in the body containing a set of rows. In a database, a relvar stands for a

NOTES

NOTES

named variable of a specific relation type in which some relation of that type has been assigned.

A database relvar that stands for relational variables is called a base table. Update operators, namely `INSERT`, `DELETE` or `UPDATE` are used to make changes in the database entry. To retrieve data some queries are made using expression according to the definitions of operators. In making queries using Structural Query Language (SQL), the heading may not always be taken as a set of column definitions always. This is so since a column may not have any name in certain cases and also, the same name may appear in two or more columns. Also, the body may not always be a set of rows since the same row may appear in the same body more than once.

2.2.3 Network Data Model

The network data model can be defined as a database model used to represent objects and the relationships among these objects. In this model, a record can have any number of parent records and it can also have multiple child records. Like the hierarchical model, the network model also supports the concept of data independence, which can be defined as the ability to change the representation of data at one level of a database system without the compulsion of changing the data representation at the next higher level. In the network data model, Data Manipulation Language (DML) is used for searching and retrieving records from the database. DML can also be used for connecting records from the set of instances, deleting and modifying records.

The network data model uses two types of data structures, records and set type, to define the data and relationship among data. Figure 2.7 represents a record type Employee that has three data items: Name, Sex and Birth Date.

Employee		
Name	Sex	Birth Date

Fig. 2.7 Employee Record Type

Set type is a description of a 1:N relationship between two record types. Each set type definition has the following elements:

- Name for set type
- Owner record type
- Number record type

Figure 2.8 represents a set type R_Dept as an arrow. This representation is known as Bachman diagram. In the figure, Department is the owner record type and Employee is the child record type. This represents a 1:N relationship between the department of the company and the employees that are working in that department.

In a database, there are set occurrences, also called set instances, corresponding to a set type. Each instance is used to relate one record from owner record type, i.e., Department to the set of records of member record types, i.e., Employee. Owner serves as parent node and member serves as a child node. Each set occurrence consists of the following elements:

- One owner record from owner record type.
- A member of related member records from the member record type.

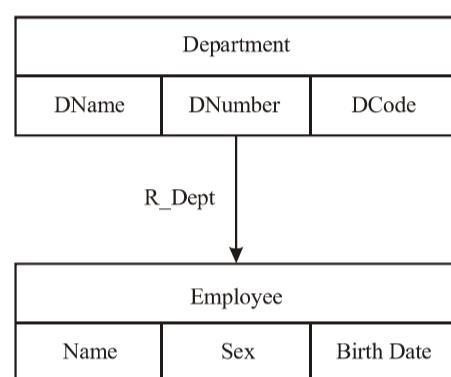


Fig. 2.8 Set Type R_Dept

A record from the member record type cannot belong to more than one set occurrence of a particular set type. This represents a 1:N relationship. A set occurrence can be easily identified by the owner record or by any number of records. The following are the differences between the set instance of a database and the set in mathematics:

- The set instance in a database has one distinguished element called owner record, whereas in mathematics, there is no such type of distinction among set elements.
- In a database, all member records of a set instance are ordered. On the other hand, in mathematics, the elements of a set are not ordered.

Construct of Network Model

The most commonly used implementation of a set type in a network model is the system-owned set. A system-owned set can be defined as a set that does not have any owner record type. In this set, the system can be regarded as an owner record type. It provides the following services to the network model:

- System-owned sets provide entry points into the database through the records of a specified member record type. Processing can be performed through the fields or data items of the member record type.
- System-owned sets can be used to order the records of a given record type by using set ordering specifications. By specifying the number of system-owned sets on the same record type, you can access your records in a different order.

NOTES

Figure 2.9 shows a network model.

NOTES

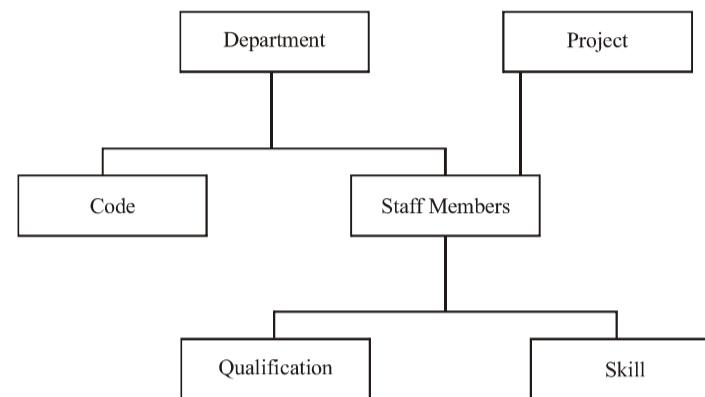


Fig. 2.9 Network Model

In Figure 2.9, Department, Project and Staff Members are the owner record types and Code, Qualification and Skill are the member record types.

The advantages of the network data model are as follows:

- It enables the representation of complex relationships and effect of operations, such as add and delete, on the relationships.
- It uses constructs, such as FIND, FIND OWNER and FIND NEXT, within a set that allows the users to navigate through the database.
- It can inherit the advantages of the hierarchical model.
- Many-to-many (M:N) relationships are easier to implement in a network model as compared to a hierarchical model.
- It ensures data integrity.

The disadvantages of the network data model are as follows:

- It provides a complex array of pointers, that thread through a set of records, that are not dealt with easily.
- It provides less scope for query optimization.

In the network model, collections of records represent data and links which are visible as pointers and represent data relationships. Database records are organized as sets of arbitrary graphs. This is illustrated in Figure 2.10.

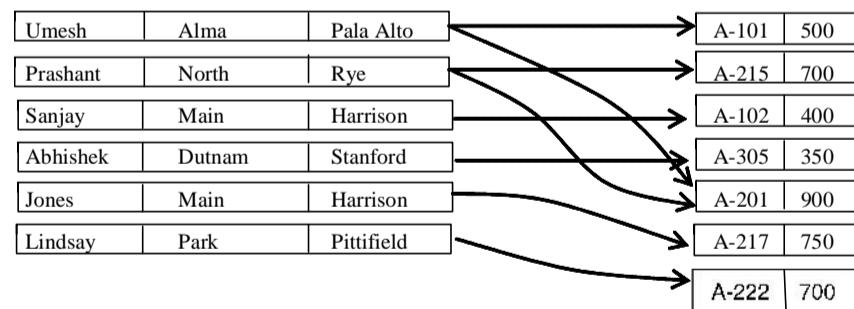


Fig. 2.10 A Sample Network Database

2.2.4 Comparison of three Models

Models

Table 2.2 summarizes a list of comparison of three models.

Table 2.2 List of Comparison of Three Models

Factor	Network Model	Hierarchical Model	Relational Model
Data Independence	Yes	Yes	Yes. But, logical data independence is more difficult to achieve than physical data independence.
Structural Independence	No. Changes in the database structure require to be made in all related application programs.	No. Changes in the database structure require to be made in all related application programs.	Yes. The relational model does not depend on the navigational data access system thus freeing the database designers, programmers and end users from learning the curves of data storage. Changes in the database structure do not affect the data access. When it is possible to make change to the database structure without affecting the DBMS's capability to access data then that the structural independence has been achieved. Relational model has structured independence.
Programming	Extensive programming required, as network model is implemented using linked list	Difficult to design, as you need to implement it using a tree.	One of the biggest advantages of the relational model is its conceptual simplicity and the ability to link records in a way that is not predefined, i.e., they are not explicit as those in the hierarchical and network models. This capability provides great flexibility particularly for end users. The relational model of data can be used in different applications and can be easily visualize the relational model as a table.
Data Definition	Network Data Definition Language (NDDL) is used to define data in network models.	Hierarchical Data Definition Language (HDDL) is used to define data in hierarchical models.	Queries uses DDL which comprises SQL commands that specify the definition of database objects that store or index data and SQL commands that control user access to database objects.
Data Manipulation	Network Data Manipulation Language (NDML) is used to modify data in network models.	Hierarchical Data Manipulation Language (HDML) is used to define data in hierarchical models.	In relational data model, queries use DML which comprises SQL commands to specify how data from existing database objects using relation is combined and manipulated to produce the data results that you want.
Constraint	A link depends on its start node and end node. If a start node or end node is deleted, the link is also deleted.	A record can only be occur if it is related to a parent record and not to a root record.	In relational model, the constraints restrict the data that can be stored in relations. These are defined using expressions that result in a Boolean value indicating whether or not the data satisfies the constraint. Constraints are applied to single attributes to a tuple or to an entire relation.

NOTES

NOTES**Check Your Progress**

1. What are the different types of data model?
2. What do you understand by network data model?

2.3 DATABASE LANGUAGES

These languages are used to define and query a database.

Data Definition Language or DDL: A database scheme is specified by a set of definitions which are expressed by a special language called DDL. DDL allows the creation and deletion of structures of database objects and provides facilities for defining and altering defined physical data structures. CREATE, DROP and ALTER are the most frequently used DDL statements. The definition also includes any constraints that are set of rules to be maintained for the integrity of a database.

A DDL statement is given as follows:

```
CREATE TABLE EMPLOYEE
  (FNAME      VARCHAR (15),
   LNAME      VARCHAR (15),
   ECODE      CHAR(5) PRIMARY KEY,
   DATE_JOIN  DATE,
   SEX        CHAR,
   SALARY     NUMBER (10,2),
   DNO        VARCHAR (5) REFERENCES DEPARTMENT (DNUMBER));
```

In most DBMS, the DDL also defines user views and sometimes storage structures. In other DBMS, separate languages, such as View Definition Language (VDL), Storage Definition Language (SDL), etc., may exist for specifying views and storage structures.

In databases where there is a separation between the conceptual and internal schemas, the DDL is used to specify the conceptual schema and the SDL is used to specify the internal schema.

An SDL statement in Oracle is written as follows:

```
CREATE TABLESPACE payroll
  DATAFILE 'C:/ACTS/payroll.tsp' SIZE 10M
  DEFAULT STORAGE (
    INITIAL 10K
    NEXT 50K
    MAXEXTENTS 999
    PCTINCREASE 10);
```

For true three-schema architecture, View Definition Language (VDL) is used to specify the user views and their mappings to the conceptual schema.

An example of a VDL statement is as follows:

```
CREATE VIEW sales AS SELECT * FROM employee WHERE dno = 'D04';
```

However, in most DBMS, the DDL is used to specify both the conceptual and external schemas.

Data Manipulation Language or DML: Once the schemas are compiled and the database is populated with data, users need to manipulate the database. DML is a language that allows users to access as well as manipulate data. Retrieving data from the database, inserting new data into the database and deleting or modifying the existing data are the activities that comprise data manipulation. A query refers to a statement in the DML that is used for data retrieval from the database. A query language is a subset of the DML used to generate a query. However, the terms DML and query language are used synonymously.

Example of DML statements are as follows:

```
SELECT ECODE, ENAME, DNO, SEX FROM EMPLOYEE;  
DELETE FROM EMPLOYEE WHERE ECODE ='E01';  
UPDATE EMPLOYEE SET DNO ='D04' WHERE ECODE='E03';
```

DML can be used in an interactive mode or embedded in conventional programming languages, such as Assembler, COmmon Business Oriented Language or COBOL, C, C++ Pascal or PL/I. Whenever DML statements are embedded in a general purpose programming language, that language is called the host language and the DML is called the data sublanguage.

There are two types of DML as follows:

- **Low-Level or Procedural DML:** This requires a user to specify what data is needed and how to get it. Examples are SQL, QUEL.
- **High-Level or Non-Procedural DML:** Here, the user is required to specify the data needed without specifying the manner of retrieval, for example, datalog, Query By Example or QBE, etc.

In most existing DBMS, the external view of data is defined outside the application program or interactive session. Data is manipulated by procedure calls to subroutines provided by a DBMS or through preprocessor statements. A uniform collection of constructs forming part of the user's programming environment, is used to define and manipulate, in an integrated environment.

Note: In most DBMS, VDL, DDL and DML are not considered separate languages but a comprehensive integrated language for conceptual schema definition, view definition and data manipulation. Storage definition is kept separate to fine tune the performance, usually done by the DBA staff.

An example of a comprehensive language is SQL which represents a VDL, DDL, DML as well as statements for constraint specification, etc.

NOTES

NOTES

As mentioned earlier, when DML commands are included in a general purpose programming language, that programming language is called the host language and the DML is called data sublanguage.

Table 2.3 shows the comparison of Data Definition Language (DDL) and Data Manipulation Language (DML).

Table 2.3 *Data Definition Language (DDL) vs. Data Manipulation Language (DML)*

Data Definition Language (DDL)	Data Manipulation Language (DML)
Data definition language defines the schema for the database by specifying entities and the relationship among them.	Data manipulation language allows the user to access or modify the data in accordance to the required data model.
DDL even defines certain security constraints.	The different operations performed using DML include, insertion of new data, deletion of stored data, retrieval of stored data and modification of data.
The execution of DDL statements results in new tables which are stored in 'system catalog' also called data dictionary or data directory. Data dictionary maintains the information of all the definitions of database schema, low-level data structures (files and records) and the relationship among various records. Data dictionary provides faster access to database by maintaining certain indexes. It also contains information regarding metadata, i.e., data about data, stored in database.	Data manipulations are applied at internal, conceptual and external levels of schemas. However, the level of complexity at each schema level varies from one another. For instance, we define complex low-level procedures to allow efficient access, but procedures focus on ease of usage, thus involve low complexity.
Data definition language allows the user to specify the storage and access methods using a special language called data storage and definition language.	Data manipulation language that involves in retrieval of data is called 'query language'. DML are classified based on the retrieval constructs as: <ul style="list-style-type: none"> • Procedural DML: Using procedural DML, the user can specify 'what' data needs to be retrieved and 'how' to retrieve the required data. • Non-Procedural DML: Non-procedural languages are also called declarative language that specifies 'what' data is to be retrieved but does not specify 'how' the data is to be retrieved.
DDL commands are CREATE, ALTER, DROP, TRUNCATE.	DDL commands are INSERT, SELECT, UPDATE, DELETE.

2.4 DATABASE ACCESS FOR APPLICATION PROGRAMS

The main purpose of **database application program** is to insert and retrieve the information from the main database like warehousing system, inventory system etc. So the data which is stored in the database is accessed by the data manager. Data manager accepts the user request and after accepting the request, data is retrieved.

Application programs are written in C, C++ Java etc. and to communicate with database application programs, DML statements needs to be executed by providing an application program interface (API) that can be used to send DML and DDL statements and retrieve the result. The DDL defines the structure in which data is placed and DML is used to extract the data from the database. So DML and DDL have the main role to access the data for application program.

NOTES

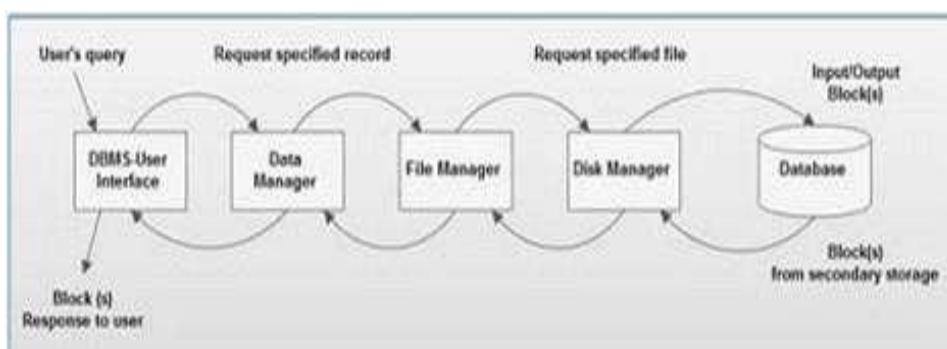


Fig. 2.11 Requested Record Requested Blocks

As shown in the Figure 2.11, the different steps of database accessing are given below:

1. DBMS-user interface (user's query) request from data manager for accessing the database.
2. The data manager sends the request for particular physical record to the file manager.
3. The file manager requests specific files from disk manager. The file manager decides which physical block of secondary storage devices contains the required record.
4. After finding the specific record, file manager sends the request for the appropriate block (a unit of physical input/output operations between primary and secondary storage) to the disk manager.
5. The disk manager retrieves the block from the database and sends it to the file manager, which sends the specific record to the data manager.

2.5 DBMS USERS

Some important DBMS users include the following:

- **Database Designer:** Before implementing a database and populating the database with data, it is required to identify the data to be stored in the database and the appropriate structure to represent and store the data. A database designer does the aforementioned tasks before the database is actually implemented and populated with data. They analyse and integrate all external views from different groups of users, and their data and processing

NOTES

requirements. It is the responsibility to identify the integrity constraints during the database design.

- **Data Administrator (DA):** A non-technical person who is responsible for managing the data resource of an organization and decides on policies.
- **Application Programmer:** A computer professional who writes application program through which a user can interact with the system. Application programmers use several tools such as RAD (Rapid Application Development) tools to construct forms and reports without application programs. Sometimes, they interact with the system through DML calls, which are embedded in a program written in host languages. This category of database users has clear idea of the structure of the database and knows clearly about the needs of the organizations.
- **Database Administrator (DBA):** The Database Administrator (DBA) is the person (or a group of people) responsible for overall control of the database system. Once database is installed and is functioning properly in a production environment of an organization, the database administrator takes over the charge and performs specific DBA-related activities including:
 - Database maintenance
 - Database Backup
 - Grant of rights to database users
 - Managing print jobs
 - Ensuring quality of service to all users
- **Routine Work:** It includes the following:
 - Acting as liaison with users to ensure that the data they require is available and to write the necessary external schemas and conceptual/external mapping (again using DDL)
 - Responding to modifications in requirements, monitoring performance and modifying details related to storage and access, thereby, organizing the system in a manner that will provide the best performance for the organization
- **End-Users:** People who use the DBMS for querying, updating generating report. End-users can be classified into the following categories:
 - **Naive users** are users who interact with the system by invoking one of the permanent application programs that have been written previously by the application programmer. Tellers in a bank, reservation clerks for airlines, hotels, and so on, are the native users. The main interface that a native user uses is form interface using GUI (Graphical User Interface).
 - **Sophisticated users** interact with the system without writing a program. Instead they form their requests using database language especially

DML. Other tools they use are OLAP (Online Analytical Processing) and data mining tools, which help them summarize data in different patterns, occasionally accessing the database using database query language.

Models

- o **Casual users** are those who access the database occasionally and have different needs each time; they use a sophisticated query language.

A database administrator can interact with the database designer during the database design phase so that he has a clear idea of the database structure for easy reference in future.

- This helps the DBA perform different tasks related to the database structure.
- The DBA also interacts with the application programmers during the application development process and provides his services for better design of applications.
- End-users also interact with the system using application programs and other tools as specified in the description above.

NOTES

2.6 TRANSACTION MANAGEMENT

At times, a single logical unit of work is formed by numerous operations on the database. Transaction is an action that is used to perform some manipulation on data stored in the database. A DBMS is responsible for supporting all the required operations on the database; it also manages the execution of transactions so that only the authorized and allowed actions are performed. The execution of transactions requires ACID properties (Atomicity, Consistency, Isolation and Durability). All operations of a transaction will be executed or none of the operations will take effect (*Atomicity*). As a result of a transaction, data records are accurate (*Consistency*). When two or more transactions run concurrently, their effects must be isolated from one another. If a transaction has completed its operations, its effect should not be lost even if the system fails immediately after the transaction completes (*Durability*). In case of failure, abandoning the partial transaction and re-applying it becomes necessary. Also, in the event of failure, the database should be able to restore itself to a consistent state.

2.7 DATABASE SYSTEMS STRUCTURE

A database system is portioned into modules that deal with each of the responsibilities of the overall system.

2.7.1 DBMS Structure

In database system the OS or Operating System provides the basic service and DBMS is built on that base. The database system structure is shown in Figure 2.12.

*Self-Instructional
Material*

37

NOTES

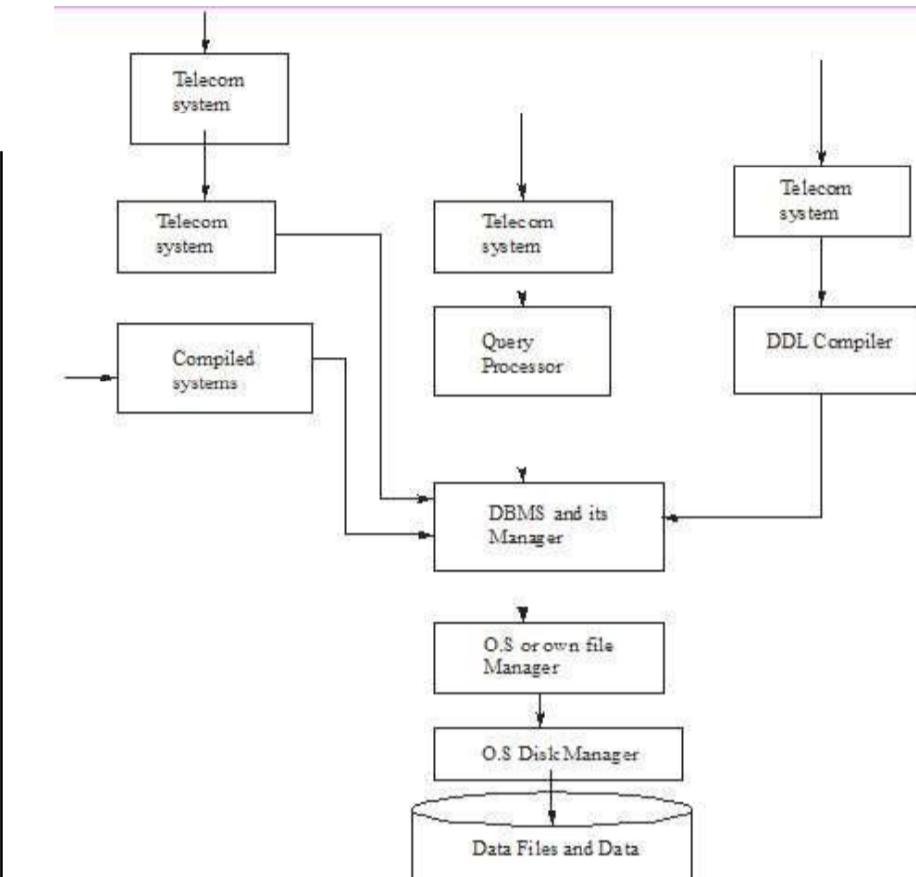


Fig. 2.12 Database System Structure

2.7.2 Applications of DBMS

- Computerized library system.
 - Automated teller machine.
 - Flight reservation system.
 - Computerized parts inventory systems.
 - Human resources.
 - Finance resources.

2.7.3 Disadvantages of DBMS

A database management system is vulnerable in the following areas:

- **Data Integrity:** A large number of users can access the database through the Internet. It becomes difficult to maintain the integrity of data with increase in the volume of users of database. Data integrity becomes vulnerable when multiple users try to update data at the same time.

- **Data Quality:** As data is accessible by remote users, it increases the chances of reduced data quality. The remote users can change, manipulate or damage the data. Adequate controls are needed to secure the data from manipulation.
- **Data Security:** In a centralized database, data is available to remote users. It increases the chances of data abuse. To reduce the chances of unauthorized users accessing important information, it becomes necessary to take administrative and technical measures.
- **Enterprise Vulnerability:** The centralization of all the enterprise information in the database makes the database an indispensable resource for the organization. The security of central database becomes a cumbersome task for the organization, as the survival of the organization may depend on the security of the database.

NOTES

2.7.4 Storage Manager

A storage manager is a software module responsible for the database storage, retrieval and updating of data.

Using the file system, which is typically supported by a conventional operating system, raw data is stored on the disk. The storage manager converts the different DML statements into low-level commands for the file system. It is also the duty of the storage manager to store, retrieve, and update data in the database.

- **Data Manager**

Data manager converts the user operations from user logical view to a physical file system. It is responsible for interfacing with the file system. In addition the task of enforcing constraints to maintain the consistency and integrity of the data, as well as its security, are also performed by data manager. Synchronizing the simultaneous operations performed by concurrent users is under the control of data manager. It is entrusted with backup and recovery operations.

- **File Manager**

It takes care of structures of files and managing file space. It is also responsible for locating the block containing the required record, requesting this block from disk manager and transmitting the required record to data manager.

- **Disk Manager**

Disk manager is the part of operating system of host computer and all physical input and output operations are performed by it. It transfers the block or page requested by the file manager, so that file manager need not be concerned with the physical characteristics of storage media.

- **Data Files**

Data files contain the disk portion of the database.

NOTES**• Data Dictionary**

Data dictionary contains a list of all files in the database, the number of records in each file and the name and type of each field. It basically contains metadata, i.e., data about data. The schema of a table is an example of metadata.

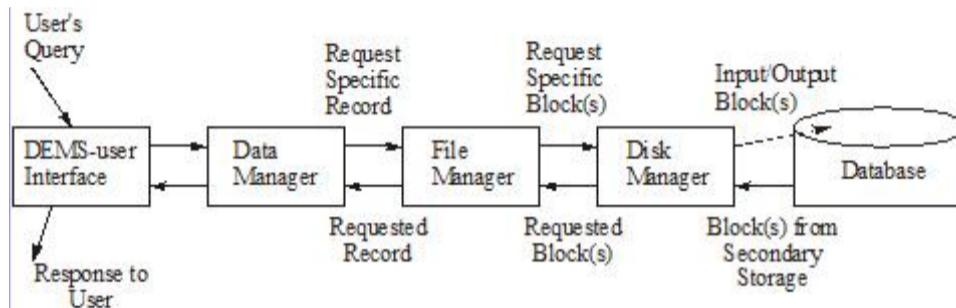


Fig. 2.13 Working of Database System Structure

2.7.5 Query Processor

It is used to interpret the online users query and convert it into an efficient series of operations in the form capable of being sent to the data manager for execution. It uses the data dictionary to find the structure of the relevant portion of database and use this information in modifying the query and prepare an optimal plan to access the database.

- **DML Compiler:** The DML statements are processed into low-level (machine language) instructions so that they can be implemented.
- **DDL Interpreter:** The DDL statements are processed into a table set containing meta data (data about data).
- **Embedded DML Pre-Compiler:** It processes DML statements into procedural calls embedded in an application programme.
- **Query Optimizer:** The instructions produced by the DML Compiler are executed.
- **Telecom System:** User of computer system communicate with it by sending and receiving messages over communication line. These messages are routed by tele communication systems.
- **DDL Compiler:** It takes the data definition statement that is in source form and converts it into object form, i.e., it converts data definition statements into a set of tables.

Check Your Progress

3. What is DDL?
4. What is DML?
5. What are the different categories of end-users?
6. What is Disk Manager?
7. What is Data Dictionary?

NOTES**2.8 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS**

1. The three types of data models are:
 - (a) Hierarchical data model
 - (b) Relational data model
 - (c) Network data model
2. The network data model can be defined as a database model used to represent objects and the relationships among these objects. In this model, a record can have any number of parent records and it can also have multiple child records.
3. DDL allows the creation and deletion of structures of database objects and provides facilities for defining and altering defined physical data structures.
4. DML is a language that allows users to access as well as manipulate data. Retrieving data from the database, inserting new data into the database and deleting or modifying the existing data are the activities that comprise data manipulation.
5. End-users can be classified into the following categories:
 - (a) Naive users
 - (b) Sophisticated users
 - (c) Casual users
6. Disk manager is the part of operating system of host computer and all physical input and output operations are performed by it. It transfers the block or page requested by the file manager, so that file manager need not be concerned with the physical characteristics of storage media.
7. Data dictionary contains a list of all files in the database, the number of records in each file and the name and type of each field. It basically contains metadata, i.e., data about data. The schema of a table is an example of metadata.

NOTES**2.9 SUMMARY**

- Data models can be defined as a collection of various concepts used to describe the structure of a database. Implementing a data model includes specifying data types, relationships among data types and constraints on the data.
- In the hierarchical model, also called hierarchical schema, data is organized in the form of a tree structure. The hierarchical model supports the concept of data independence.
- Data independence is the ability to change the representation of data at one level of a database system without the compulsion of changing the data representation at the next higher level.
- The main principle of the relational model is the information principle—all information is represented by data values in relations. The three components—structural, manipulative and integrity—make up the relational model.
- The network data model can be defined as a database model used to represent objects and the relationships among these objects. In this model, a record can have any number of parent records and it can also have multiple child records.
- DDL allows the creation and deletion of structures of database objects and provides facilities for defining and altering defined physical data structures.
- DML is a language that allows users to access as well as manipulate data. Retrieving data from the database, inserting new data into the database and deleting or modifying the existing data are the activities that comprise data manipulation.
- Data Administrator (DA) is a non-technical person who is responsible for managing the data resource of an organization and decides on policies.
- The Database Administrator (DBA) is the person (or a group of people) responsible for overall control of the database system.
- Naive users are users who interact with the system by invoking one of the permanent application programs that have been written previously by the application programmer.
- Sophisticated users interact with the system without writing a program. Instead they form their requests using database language especially DML.
- Casual users are those who access the database occasionally and have different needs each time; they use a sophisticated query language.
- A database system is portioned into modules that deal with each of the responsibilities of the overall system.

- A DBMS is responsible for supporting all the required operations on the database; it also manages the execution of transactions so that only the authorized and allowed actions are performed. The execution of transactions requires ACID properties (Atomicity, Consistency, Isolation and Durability).
- In database system the OS or Operating System provides the basic service and DBMS is built on that base.
- A large number of users can access the database through the Internet. It becomes difficult to maintain the integrity of data with increase in the volume of users of database. Data integrity becomes vulnerable when multiple users try to update data at the same time.
- A storage manager is a software module responsible for the database storage, retrieval and updating of data.

Models

NOTES

2.10 KEY WORDS

- **Data Administrator (DA):** A non-technical person who is responsible for managing the data resource of an organization and decides on policies.
- **Naive Users:** These are users who interact with the system by invoking one of the permanent application programs that have been written previously by the application programmer.
- **Casual Users:** These are the users who access the database occasionally and have different needs each time; they use a sophisticated query language.

2.11 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What are the advantages and disadvantages of hierarchical data model?
2. Discuss the disadvantages of relational model.
3. Differentiate between DDL and DML.
4. What are the different types of DBMS users?
5. What is Query Processor?

Long Answer Questions

1. Explain the various types of data models with the help of suitable diagrams.
2. Compare the hierarchical, relational and data model.
3. Explain the database system structure.
4. What is storage manager and query processor?

NOTES

2.12 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

UNIT 3 HISTORY OF DATABASE SYSTEMS

History of Database Systems

NOTES

Structure

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Database and Concept Design Using Entity–Relationship Model
 - 3.2.1 Entity
 - 3.2.2 Attribute
 - 3.2.3 Types of Attributes
 - 3.2.4 Relationship
 - 3.2.5 Classifying Relationships
 - 3.2.6 Unary Relationship
 - 3.2.7 Ternary Relationship
 - 3.2.8 Multiplicity
- 3.3 Entity–Relationship Diagram
 - 3.3.1 E–R Diagram Notation
 - 3.3.2 Attributes of a Relationship
 - 3.3.3 Representing Participation in E–R Diagram
- 3.4 Additional Features of E–R Model
 - 3.4.1 Attribute Inheritance
 - 3.4.2 Constraints on Specialization and Generalization
 - 3.4.3 Aggregation
- 3.5 Conceptual Design for Large Enterprises
- 3.6 Answers to check your progress Questions
- 3.7 Summary
- 3.8 Key Words
- 3.9 Self Assessment Questions and Exercises
- 3.10 Further Readings

3.0 INTRODUCTION

The method of converting a logical data model into an actual physical database is the creation of a database. Until creating the model of that implementation, technicians often jump to the physical implementation. It's unwise here. Before you can even start to build a physical database, a logical data model is required.

E–R model is used to define the conceptual schema or semantic data model of a relational database and its requirements, by using the diagrams is a data modelling method. E–R diagrams are graphical representation of entities and their relationships to each other to represent the organisation of data in the database. You will also learn about generalization, specialization and aggregation. Generalization refers to an abstraction in which a set of similar objects is regarded as a generic object. Specialization refers to an abstraction in which a relationship between objects is regarded as a higher level object. Aggregation is used to deal with a relationship set as an entity set for participation with other relationships.

Self-Instructional Material

45

NOTES

In this unit, you will study about the Entity-Relationship or E-R model, ER design entities, attributes and entity sets, relationship and relationship sets, additional features of ER model, concept design with the ER model, conceptual design for large enterprises.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand and draw the entity-relationship diagrams
 - Discuss entity and different types of attributes
 - Define and classify relationship between entities
 - Represent participation in E-R diagram
 - Discuss the additional features of E-R model
 - Understand the conceptual design for large enterprises
-

3.2 DATABASE AND CONCEPT DESIGN USING ENTITY-RELATIONSHIP MODEL

The Entity–Relationship (E–R) model facilitates database design by enabling the designer to express the logical properties of the database in an enterprise schema. Identification of real-world objects referred to as entities, forms the basis of this model. These entities are described by their attributes and are connected by relationships among them. The E–R model has its own set of symbols for drawing the E–R diagram which depicts the logical model of the database.

Regardless of the database system selected, the enterprise schema will be valid. It is capable of remaining constant even if the DBMS is modified. The E–R model enables you to express restrictions or constraints on the entities or relationships.

3.2.1 Entity

Any ‘thing’, that is, an object, a being or an event, that has an independent existence, in the real world, is called an entity. The entity may have a physical existence in the real world, such as in the case of a house, a bicycle or an employee.

It may also be an object with a conceptual existence, for example, a company, a job or a university course.

3.2.2 Attribute

An entity is described by a set of properties called attributes. An employee entity might have attributes, such as employee code, employee name, salary, date of join, and so on.

An entity set is a set of entities of the same type that share the same properties or attributes.

The individual entities that constitute a set are said to be the extension or instance of the entity set. As all entities in an entity set have the same attributes, entity sets also share the attributes of the contained entities. Therefore, attributes are descriptive properties possessed by each member of an entity set.

In other words, an entity is determined by its instantiations. A particular instance of an attribute is a **value**. For example, ‘Hari Nandan Tunga’ is one value of the attribute Name. The value of the attribute can be different for each entity in a set. A set of permitted values exists for each attribute. This is called domain or a value set of that attribute. The domain of the attribute employee_name, for instance, may be the set of all text strings of a particular length.

3.2.3 Types of Attributes

An attribute, as used in the E–R model, can be characterized by the following attribute types. Figure 3.1 shows the different types of attributes.

- **Simple vs Composite:** A simple attribute has an atomic value, i.e., an attribute, which cannot be decomposed into subparts. For example, employee code, salary of an employee. In contrast, composite attributes can be divided into subparts (that is, other attributes). For example, employee-name could be structured as a composite attribute consisting of first-name, middle-initial and last-name.

Composite attributes help us to group together related attributes making modeling more readable and flexible. The advantage of defining a composite attribute for an entity is that a user can refer to the entire composite attribute as well as only a component of the attribute according to the need.

- **Single-Valued vs Multivalued:** An attribute is said to be single-valued if it has a single value for any instance. There may be instances where an attribute has a set of values for a specific entity. This type of attribute is said to be multivalued. For example, an employee may have several degrees or he or she may have several different phones, each with its own number. It is to be noted that a single-valued attribute that changes value over time does not imply that it is multivalued. A single-valued attribute is not necessarily a simple attribute.

- **Stored vs Derived:** Although most of the attributes will be stored, there will be some attributes that can be derived or computed from other related attributes. A derived attribute need not be physically stored within the database; instead, it is derived by using an algorithm. For example, a person’s AGE attribute can be derived by subtracting the Date of Birth (DOB) from the current date. Similarly, the annual salary of an employee can be derived by multiplying the salary by 12.

NOTES

NOTES

- **Null Attributes:** When a certain entity does not possess a value for an attribute, a NULL value is used. This will mean ‘not applicable’ or that the ‘value is unknown’ or ‘non-existent’. A null value for ‘apt_number’ attribute, for example, could indicate that the address is not inclusive of an apartment number or that there is an apartment number which is not known to us, or that it is unknown whether an apartment number is included in the employee’s address or not.

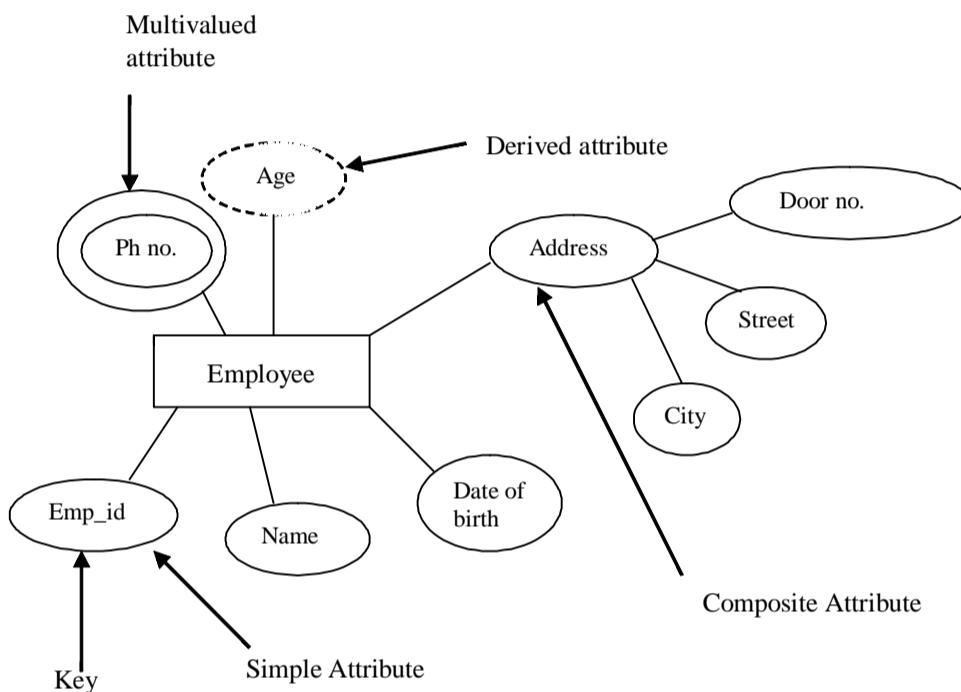


Fig. 3.1 E-R Diagram showing Different Types of Attributes

The attributes of an entity set may be divided into the following two groups:

- **Identifiers:** A set of one or more attributes that uniquely identifies an instance of an entity. These are also termed as *key* attributes; For example, employee code attribute of the EMPLOYEE entity set.
- **Descriptors:** These attributes provide a non-unique characteristic of an entity instance. These are said to be *non-key* attributes. For example, employee name, salary, date of join, and so on, attributes of the EMPLOYEE entity set.

3.2.4 Relationship

A relationship represents an association between two or more entities. For example, we can define a relationship that associates an employee, ‘Jayanta Dutta’ whose employee code is ‘E01’ with project ‘P01’. The relationship specifies that the employee E01, i.e., ‘Jayanta Dutta’ is assigned on project ‘P01’.

NOTES

A **relationship type or relationship set** is a set of ‘similar in kind’ relationships among one or more entities. A relationship set is a set of relationship of the same type.

3.2.5 Classifying Relationships

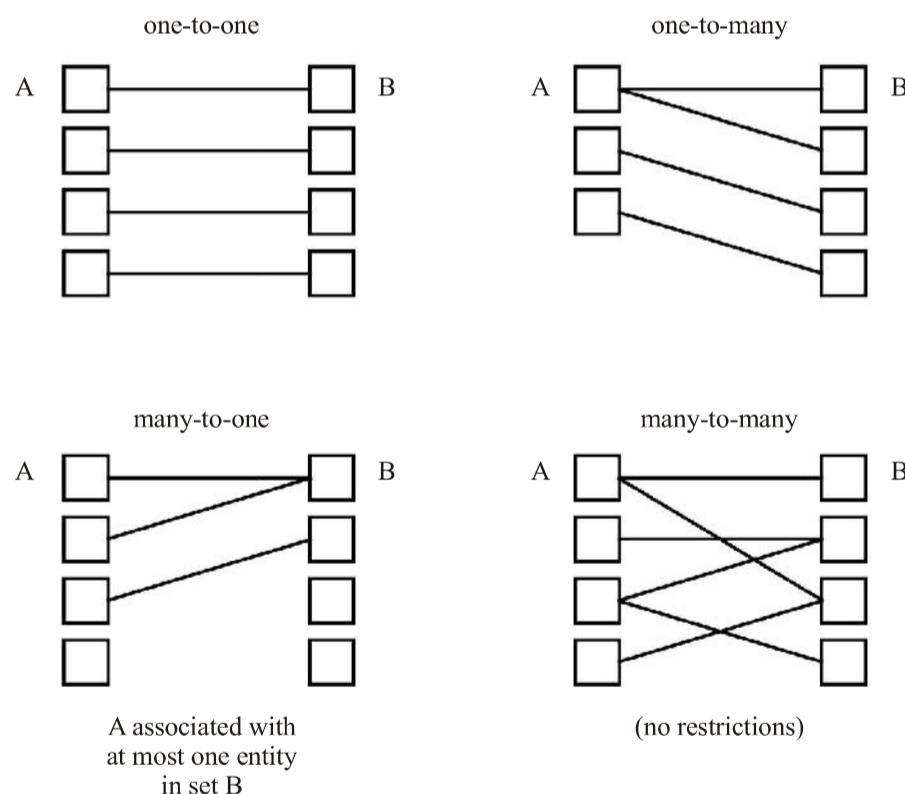
The degree of a relationship is the number of entities associated with the relationship. The n-ary relationship is the general form for degree n. Relationships is unary (recursive), binary and ternary where the degree is 1, 2 and 3, respectively.

The mapping of associated entity instances in a relationship is described by the **connectivity** of the relationship.

The actual number of related occurrences for each of the two entities is called the **cardinality** of the relationship.

The number of relationships wherein an entity can participate in a particular relationship type is specified by the **cardinality ratio**. In the most common case of binary relationships, the cardinality ratios are as follows:

- One-to-One (1:1)
- One-to-Many (1:M, 1:*)
- Many-to-One (M:1 or *:1)
- Many-to-Many (M:N)



NOTES

One-to-One: A one-to-one relationship type (1-1 or 1:1) means that there is at the most one entity from each type participating in the relationship.

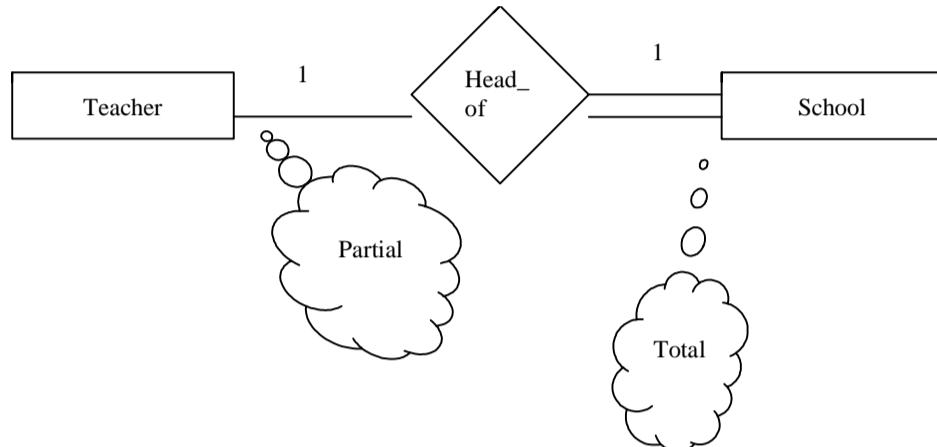


Fig. 3.2 E-R Diagram showing Non-Uniform Binary 1:1 Relationship

Figure 3.2 depicts non-uniform binary 1:1 relationship. As the relationship, Head_of, is connected to two entities, Teacher and School, so it is a binary relationship. Again, a school possesses only one single head master who belongs to the Teacher entity set. On the other hand, one person cannot act as a head master of more than one school. So, at the most, one entity from both entity sets can be associated with the relationship Head_of. Observe that every teacher cannot act as a head master of a school. Therefore, there are some teachers besides the head master. So, Teacher entity participates partially in the relationship but as every school possesses at least one head master and more than one teacher. Hence, the entity School participates totally.

Figure 3.3 is an example of uniform binary 1:1 relationship:

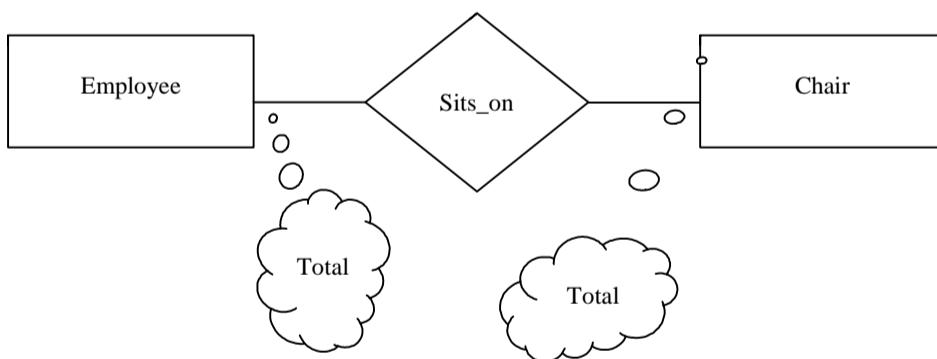


Fig. 3.3 E-R Diagram showing Uniform Binary 1:1 Relationship

In Figure 3.3, we see that Sits_on is a binary relationship as it involves two entity sets. As at a time only one employee can sit on a chair, so only one entity from this set can participate in the relationship Sits_on. Only one employee can use one chair. As only one entity of the Chair entity set is involved in the relationship, it is a

NOTES

binary 1:1 relationship. As each employee has a chair to sit and every chair is meant for one employee only, the participation here is total on both sides.

One-to-Many: A relationship wherein a single entity of a single entity type can be related to multiple entities of another type is called a one-to-many relationship type (1:N or 1:N). However, each entity of the other type is related to one entity of the first type, at the most (see Figure 3.4).

There exists a one-to-many relationship between the entity MANAGER and the entity EMPLOYEE because there are several employees working under a manager. The reverse of this relationship would be a many-to-one relationship because, that is, EMPLOYEE to MANAGER (see Figure 3.5). This is because, one manager has several employees working under him. If there is an entity set EMPLOYEE, for instance, there could be just a single instance of the entity set MANAGER to whom that employee directly reports (the assumption here is that no employee reports to more than one manager).



Fig. 3.4 E-R Diagram showing Binary 1:N Relationship

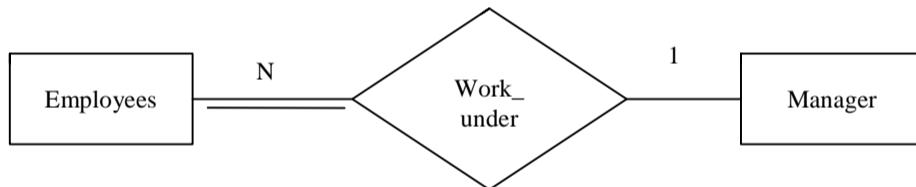


Fig. 3.5 E-R Diagram showing Binary N:1 Relationship

Many-to-Many: A many-to-many relationship type (M-N or M:N) is one in which a single entity of one entity type is related to N entities of another type and vice versa. Figure 3.6 shows the M-N relationship.

The relationship between the entity EMPLOYEE and the entity PROJECT can be understood as follows: Each employee could be involved in several projects; various employees could be working on a given project. This EMPLOYEE—PROJECT relationship is a many-to-one relationship.

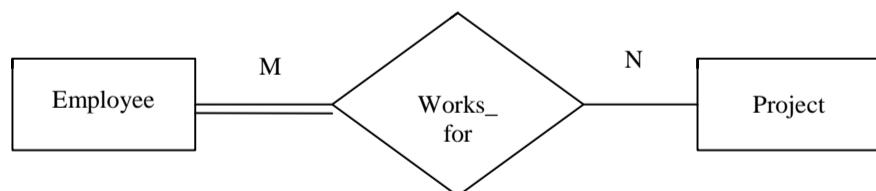


Fig. 3.6 E-R Diagram showing Binary M:N Relationship

NOTES

3.2.6 Unary Relationship

Up to now, we have worked with binary relationships only. Now, we shall observe unary relationship where only one entity participates in the relationship. Like binary relationship, unary relationship can also be subdivided into the following categories:

- Unary 1:1
- Unary 1:N
- Unary M:N

Unary 1:1

In Figure 3.7, an employee is married to another employee. Both employees here belong to the same Employee entity. So only one entity Employee is associated with the relationship set Married_to. If we assume that one employee can get married to only one employee, then it becomes a unary 1:1 relationship.

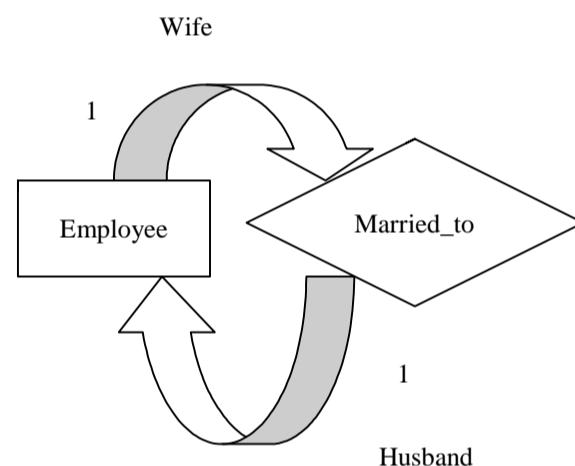


Fig. 3.7 E–R Diagram showing Unary 1:1 Relationship

Unary 1:N

Here, Employee is an entity set to which belongs an employee who is manager_of other employees belonging to the same Employee set. So here, the employee entity who is the manager is associated with more than one employee. Thus, it is an unary 1:N relationship (here we assume that an employee works under only one manager). Figures 3.8 depicts 1:N relationship and 3.9 depicts unary M:N relationship.

NOTES

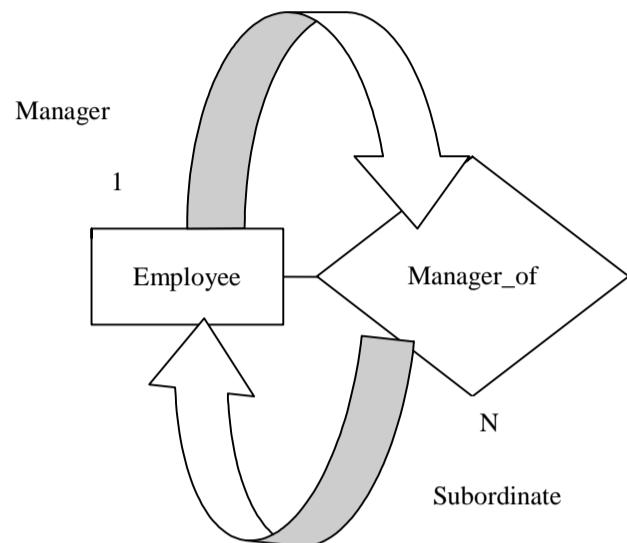


Fig. 3.8 E-R Diagram showing Unary 1:N Relationship

Unary M:N

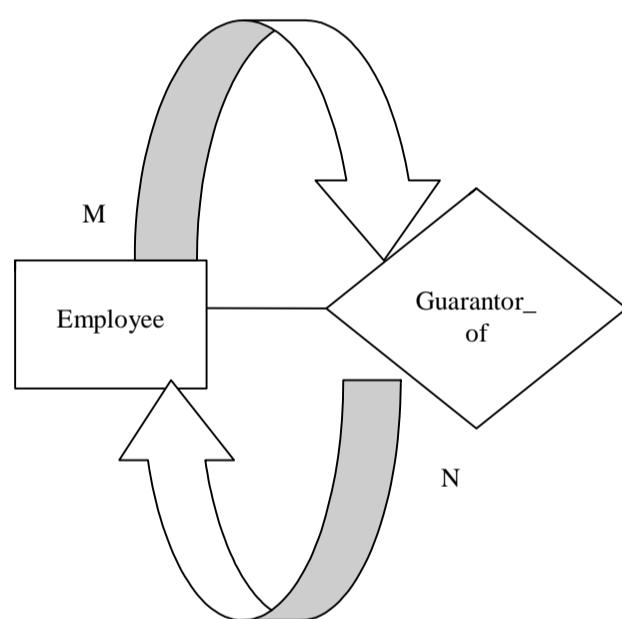


Fig. 3.9 E-R Diagram showing Unary M:N Relationship

3.2.7 Ternary Relationship

In ternary relationships, more than two entities are involved. In this example, Prescribes is a ternary relationship that associates three entity sets: Doctor, Medicine and Patient (see Figure 3.10).

NOTES

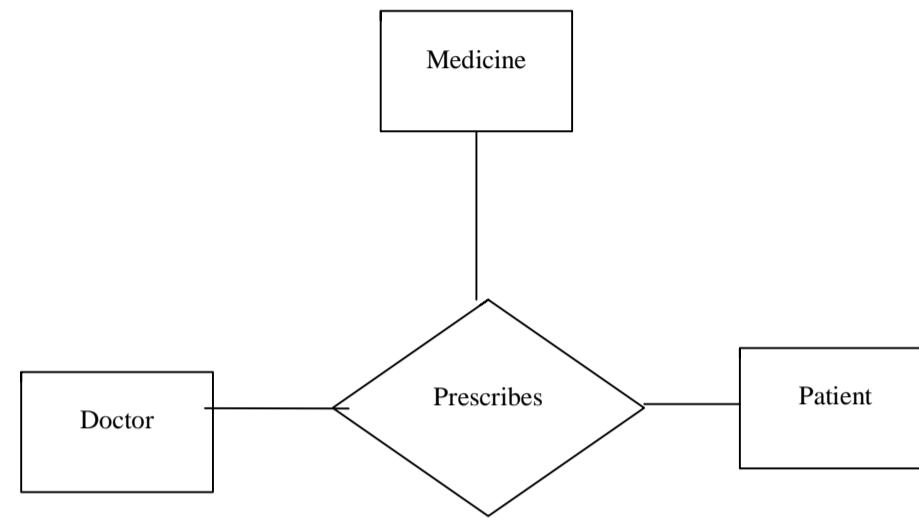


Fig. 3.10 E-R Diagram showing Ternary Relationship

3.2.8 Multiplicity

Multiplicity means the way entities are related. The numbers of possible occurrences of an entity type may associate to a single occurrence of an associated entity type through a particular relationship.

Determining the Multiplicity: Table 3.1 shows the ways to represent multiplicity constraints:

Table 3.1 Ways to Represent Multiplicity Constraints

	Description
(0, 1) OR 0..1	zero or one entity occurrence
(1, 1) OR 1..1	exactly one entity occurrence
(0, m) OR 0..*	zero or many entity occurrence
(1, m) OR 1..*	one or many entity occurrence

Multiplicity and Cardinality: Multiplicity is the number or range of possible occurrence of an entity type that may relate to single occurrence of any associated entity type via a particular relationship.

Cardinality describes the maximum number of possible relationship occurrence for an entity participating in a given relationship type.

3.3 ENTITY-RELATIONSHIP DIAGRAM

An E-R diagram can graphically represent the overall logical structure of a database. The relative simplicity and pictorial clarity of this diagramming technique may well account in large part of the widespread use of the E-R model.

NOTES

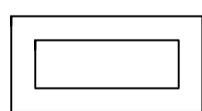
3.3.1 E-R Diagram Notation

The following should be kept in mind in case of E-R diagram notation:

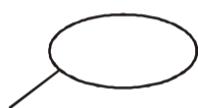
- In an E-R diagram, an **entity type** is represented with a rectangular box inscribed with the name of that entity type.



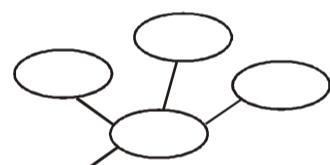
- In an E-R diagram, a weak entity is represented by a rectangle with double line. These are entities having no unique identifier of their own. It does not exist on its own. It always depends on a strong entity.



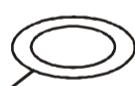
- In an E-R diagram, we will represent an **attribute** using an oval inscribed with the name of the attribute as follows:



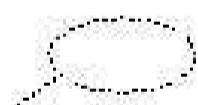
- A **hierarchy of ovals represents a composite attribute** where each oval represents an attribute value within the composite.



- A **multivalued attribute** is represented as an oval within an oval.



- Finally, a **derived attribute** is represented as an attribute with dashed or dotted lines.

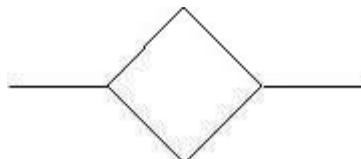


- In an E-R diagram, we depict a key attribute (or an attribute that is part of a key) by underlining the attribute name as follows:

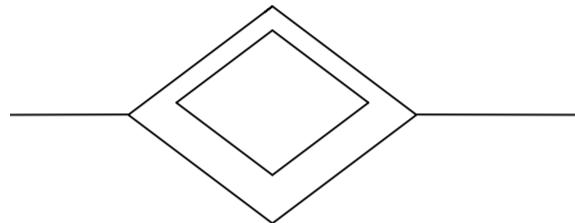


NOTES

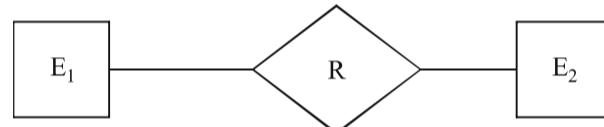
- In an E-R diagram, we depict a **relationship** type as a diagonal box. The cardinality ratio is also shown by adding 1, N or M to the lines connecting the relationship type to the entity type. Below is the outline of a one-to-many relationship type between two different entities.



- The relationship with which a weak entity is connected is known as weak relationship and is denoted by a double lined diagonal box.



- In an E-R diagram, a total participation is represented by double line between the entity that participates totally in the relationship and the relationship. Partial participation is represented by single line.



3.3.2 Attributes of a Relationship

Descriptive attributes may exist in a relationship. There may be an M:N relationship, for example, ‘supply’, between the entity VENDOR and the entity ITEM. The attribute ‘StartDate’ may be associated with the relationship to indicate the time when a particular employee began working on a particular project.

Direction: The direction of a relationship is indicative of the originating entity of a binary relationship. The source entity is the entity that a relationship originates from. The target entity is where the relationship comes to an end or terminates.

Connectivity determines the direction of a relationship. In a one-to-one relationship, the direction is from the source entity to a target entity. If neither of the entities are dependent on each other, the direction will be arbitrary. With one-to-many relationships, the entity that occurs once is the parent. The direction of many-to-many relationships is arbitrary.

The Relationship Set borrower may be many-to-many, one-to-many, many-to-one or one-to-one. To distinguish among these types, we draw either a directed line (\rightarrow) or an undirected line ($--$) between the Relationship Set and Entity Set in question.

NOTES**3.3.3 Representing Participation in E-R Diagram**

The *participation* constraint specifies whether the existence of an entity is dependent on the existence of another entity to which it must be related. There are two types of participation constraints:

- **Total:** It indicates that the existence of an entity depends on being related to another entity.
- **Partial:** It indicates that the existence of an entity is not dependent on being related to another entity.

In E-R diagrams:

- Partial participation is denoted by single lines.
- Total participation is denoted by double lines (or thick lines).

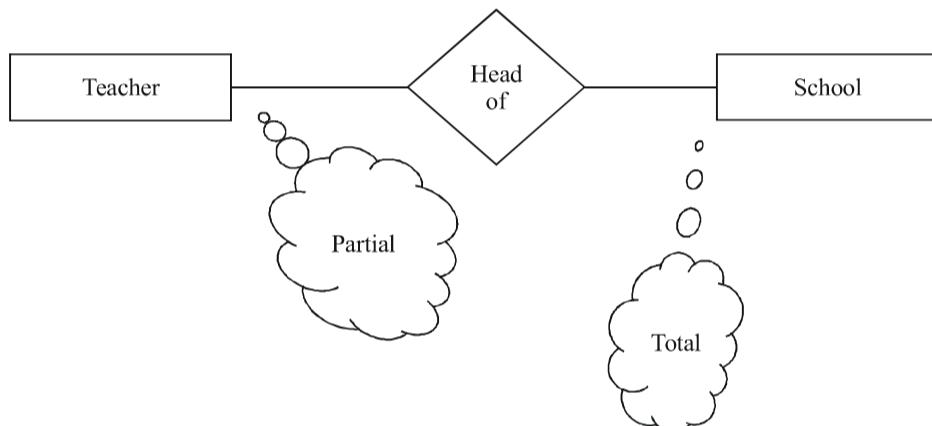


Fig. 3.11 E-R Diagram showing Partial and Total Participation

Figure 3.11 illustrates the concept of total and partial participation. Each school must have one head master. So, the entity School participates in the relationship Head of totally. On the contrary, from among many teachers in a school, only one is chosen to be the head master. So here, the entity Teacher participates partially in the relationship.

Cardinality Constraint: The lower and upper bounds on the number of relationships in which each entity is allowed to participate is specified.

Maximum cardinality: The maximum number of entities that can occur on a particular side of the relationship are indicated by the numbers inside the relationship.

Minimum Cardinality: This refers to the minimum number of entities that can exist on one side of the relationship.

The mapping cardinality of a relationship set depends on the real-world relationships it is modeling. Minimum cardinality and maximum cardinality pairs enclosed within parenthesis denote cardinality constraint.

NOTES

Figure 3.12 shows maximum and minimum cardinality.

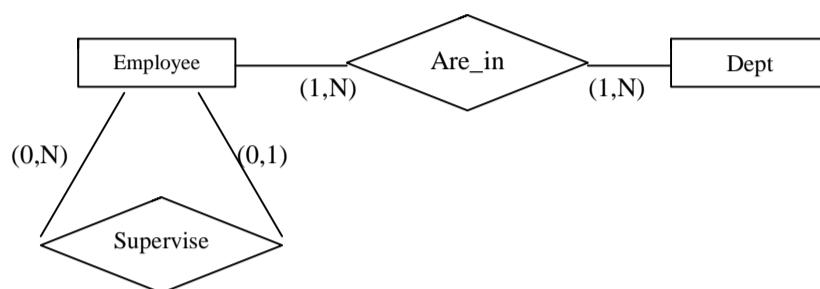


Fig. 3.12 E-R Diagram showing Maximum and Minimum Cardinality

Check Your Progress

1. Define entity.
2. What are attributes?
3. What is cardinality ratio?
4. What is multiplicity?

3.4 ADDITIONAL FEATURES OF E-R MODEL

Abstraction is the simplification mechanism used to concentrate on the properties that are of interest to the application. Vehicle is an abstraction that includes the types car, truck and bus.

There are two main abstraction mechanisms used to model information:

- Generalization
- Aggregation

Generalization is a powerful and widely used method for representing common characteristics among entities while suppressing or ignoring their differences. It is the relationship between an entity and one or more refined versions. The entity being refined is called **higher-level entity or super type or generic entity**, and each refined version is called **lower-level entity subtype**. For instance, a student is a generalization of graduate or undergraduate, full-time or part-time students.

Generalization hierarchies should be made use of when attributes are repeated for numerous entities, when numerous entities appear to be of the same type, when the model is evolving continually. They result in the stability and improvement of the model by allowing modifications to be made to only those entities that are pertinent to the change and contribute to the simplification of the model through reduction in the number of entities in the model.

NOTES

The abstracting process that introduces new characteristics to a current entity class for the creation of one or more new entity classes is known as **specialization**. This involves taking a higher-level entity/super type, using additional characteristics and generating lower-level entities or subtypes. The features of the higher-level entity or super type are inherited by the lower-level entities or subtypes. A characteristic property of generalization hierarchies is that every instance of a super type is an instance of the subtype. Figure 3.13 depicts generalization and specialization.

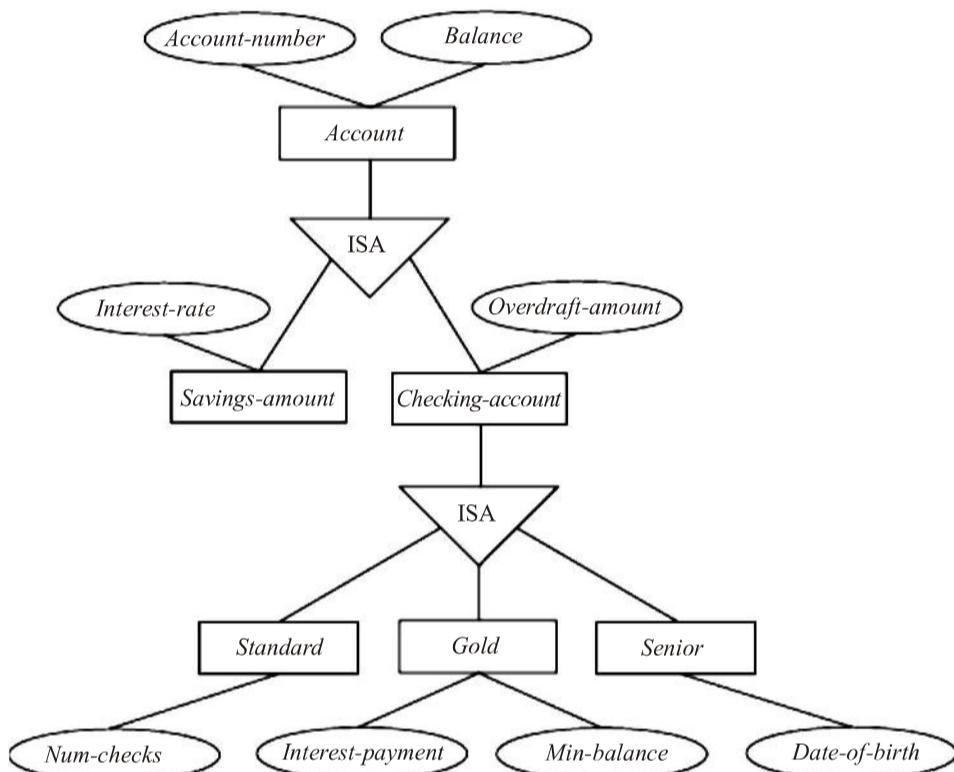


Fig. 3.13 E-R Diagram showing Generalization and Specialization

3.4.1 Attribute Inheritance

In specialization, the lower-level entity set inherits a set of attributes that are common in all lower-level entity sets. Consider an entity set X . Its subtype or subclass would be a set of entities comprising the following:

- All of the attributes found in entities of X
- Additional attributes to the entities of X
- Entities involved in all of the relationships involving X
- Entities that may be involved in additional relationships on its own

In other words, the subclass inherits the attributes and relationships of A . Generalization is an IS-A relationship. If all lower-level entity sets are involved in

NOTES

only one IS-A relationship, it is a hierarchy. If some or all lower-level entity sets are involved in more than one IS-A relationship, it is a lattice.

3.4.2 Constraints on Specialization and Generalization

We can define constraints on the super type relationships to restrict the participation of entities to the subtypes.

- **Inclusion Constraints**

- The disjoint constraint specifies that the subtypes of a super type are disjoint. This means that an entity can be a member of only one subtype.
- The non-disjoint (OVERLAPPING) constraints specify that the subtypes are overlapping and an entity may be a member of more than one subtype.

Based on inclusion constraint, a generalization hierarchy can be either overlapping or disjoint. In an overlapping hierarchy, an entity instance can be part of multiple subtypes. It is quite possible that an employee may be a manager as well as a secretary in a company. In a disjoint hierarchy, an entity instance can be in only one subtype. For example, the entity EMPLOYEE may have two subtypes: PERMANENT and CONTRACT-BASIS. An employee may be one type or the other but not both.

- **Completeness Constraints**

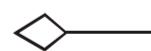
A total specialization constraint specifies that every entity in the super class must be a member of some of its subclasses. For example, a student must belong to one of the subclasses of Postgraduate and Undergraduate.

3.4.3 Aggregation

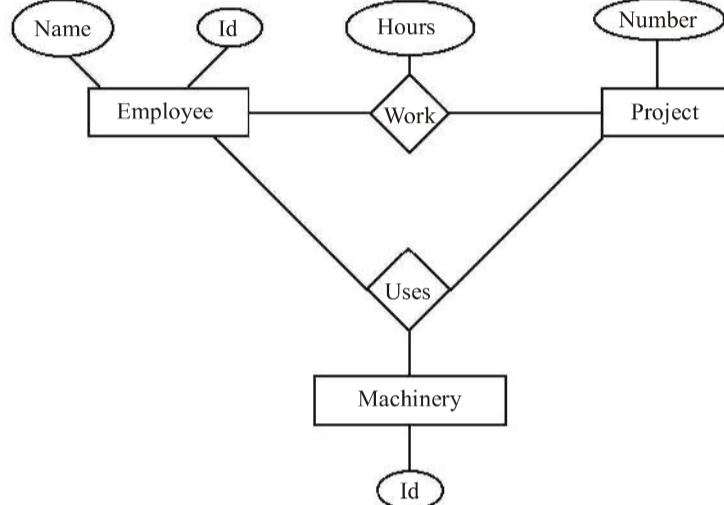
Aggregation is used to deal with a relationship set as an entity set for participation with other relationships. It is a special kind of association that specifies a whole or part relationship between the aggregate (whole) and a component part. It is different from an ordinary association as it has the following distinguishing characteristics:

- It is an asymmetric relationship.
- It is a transitive relationship.
- It implies stronger coupling.
- It propagates functions, such as copy, delete, and so on.

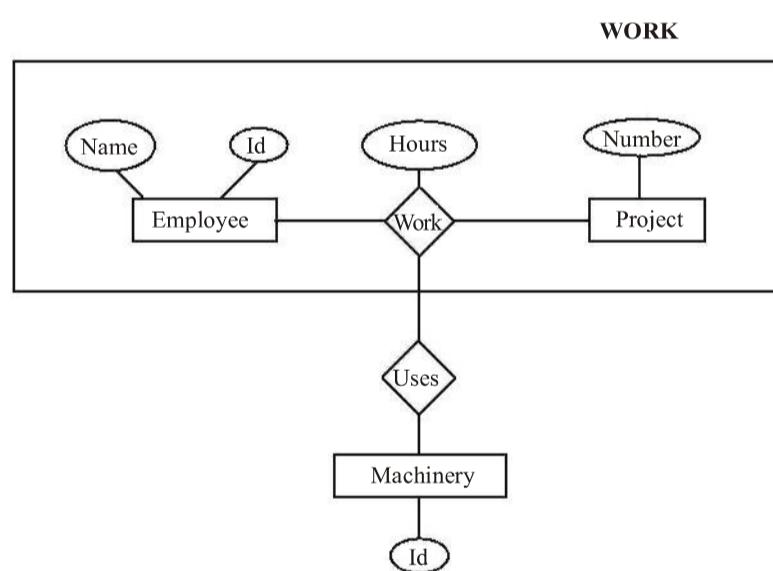
Because aggregation is not symmetric, it is visually distinguished as to which is the aggregate and which is a component part. As per the **Unified Modeling Language (UML)**, the association line is shown with a small diamond at the aggregate end. In an aggregation relationship, the part can be independent of the whole but the whole depends on the part. An aggregation relationship is defined with an unfilled diamond and a line as follows:



NOTES



It appears as if Work and Uses could be combined into one relationship but it would obscure the logical structure of the scheme. The best use of aggregation is in the form of an abstraction where relationships are considered as higher-level entities.



3.5 CONCEPTUAL DESIGN FOR LARGE ENTERPRISES

NOTES

The conceptual design require a data model for the information which is to be present in the system. The resultant ER diagram will represent real-world domain. This model is to be checked and validated. The system must be redundant. It also ensures that the system should meets the business requirements.

As we are aware about the ER model constructs which described various application concepts and relationships. To describe the small fragments of the application in terms of ER diagrams it is the main process of conceptual design. The design may require the lots of hard work of more than one designer and extent application code and data which is used by a number of user groups, this conceptual work for the large enterprise. The main aspect in the design process is methodology used which is used to structure the development of the overall design. It also make sure that the design is consistent takes into account all user requirements. For conceptual design, by using a high-level and semantic data model like ER diagrams which gives the benefit that the high-level design can be pictorially represented. It is also easily understood by the many people, and these must provide input to the design process. If there is any conflicting requirements these are some way resolved and the requirements of a variety of user groups are considered. At the end of the requirements analysis phase, a single set of global requirements are generated. To generate a single set of global requirements is a not easy task, but it allows the conceptual design phase to continue with the development of a logical schema. This procedure spans all the applications and data throughout the enterprise.

An approach is that, for the different user groups, to develop separate conceptual schema and these schemas then to be integrated. For this task, there is a requirement that there are correspondences between entities, relationships, and attributes. There are different types of conflicts like differences in measurement units, mismatches, naming conflicts and these must be resolved. In some situation, schema assimilation cannot be avoided. For example, when one organization merges with another organization, so the databases have to be integrated. So this integration of schema, it also raising the significance as users demand access and also maintain by different organizations.

Check Your Progress

5. What are two main abstraction mechanisms used to model information?
6. What is specialization in E-R model?

3.6 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Any ‘thing’, that is, an object, a being or an event, that has an independent existence, in the real world, is called an entity.
2. An entity is described by a set of properties called attributes. An employee entity might have attributes, such as employee code, employee name, salary, date of join, and so on.
3. The number of relationships wherein an entity can participate in a particular relationship type is specified by the cardinality ratio.
4. Multiplicity is the number or range of possible occurrence of an entity type that may relate to single occurrence of any associated entity type via a particular relationship.
5. There are two main abstraction mechanisms used to model information:
 - Generalization
 - Aggregation
6. The abstracting process that introduces new characteristics to a current entity class for the creation of one or more new entity classes is known as specialization.

NOTES

3.7 SUMMARY

- The Entity–Relationship (E–R) model facilitates database design by enabling the designer to express the logical properties of the database in an enterprise schema. Identification of real-world objects referred to as entities, forms the basis of this model.
- Any ‘thing’, that is, an object, a being or an event, that has an independent existence, in the real world, is called an entity
- An entity is described by a set of properties called attributes.
- A relationship represents an association between two or more entities.
- The degree of a relationship is the number of entities associated with the relationship.
- Multiplicity means the way entities are related. The numbers of possible occurrences of an entity type may associate to a single occurrence of an associated entity type through a particular relationship.
- An E–R diagram can graphically represent the overall logical structure of a database. The relative simplicity and pictorial clarity of this diagramming technique may well account in large part of the widespread use of the E–R model.

NOTES

- Abstraction is the simplification mechanism used to concentrate on the properties that are of interest to the application.
- Generalization is a powerful and widely used method for representing common characteristics among entities while suppressing or ignoring their differences.
- The abstracting process that introduces new characteristics to a current entity class for the creation of one or more new entity classes is known as specialization.
- Aggregation is used to deal with a relationship set as an entity set for participation with other relationships. It is a special kind of association that specifies a whole or part relationship between the aggregate (whole) and a component part.

3.8 KEY WORDS

- **Entity:** Any ‘thing’, that is, an object, a being or an event, that has an independent existence, in the real world.
- **Attributes:** These are the set of properties that describe an entity.

3.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Discuss the various types of attributes.
2. What are the two categories of the attributes of an entity set?
3. What is a relationship? Give examples of binary and ternary relationship.
4. What is a recursive relationship?
5. Distinguish between single-valued attribute and multivalued attribute.
6. Define the following terms: (a) Entity, (b) Composite attribute, (c) Multivalued attribute, (d) Derived attribute, (e) Weak entity and partial key, (f) Complex attribute

Long Answer Questions

1. Explain the concept of relationships in E-R model.
2. What is unary relationship? Explain its types.
3. Describe the various notations used in E-R diagram.

4. In the context of E-R relationship diagram, diagrammatically represent each of the following types of relationships: (a) One-to-One (b) One-to-Many (c) Many-to-Many
5. Write detailed notes on:
 - (a) Relationship
 - (b) Generalization and specialization in an E-R model
6. Draw an E-R diagram of an university/library/banking system.
7. Design and draw an E-R diagram for a database given the following: Airplanes have a model design (Phantom, Tomcat, Crusader, and so on), a unique side number, total flight hours, 14 pilot have certifications (pilot, carrier, check pilot, and so on), social security numbers, total hours and name. Pilots are scheduled to fly airplanes on specific dates, only the latest of which is recorded.

NOTES

3.10 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

BLOCK - II

RELATIONAL MODEL

UNIT 4 INTRODUCTION TO CONSTRAINTS AND VIEWS

Structure

- 4.0 Introduction
 - 4.1 Objectives
 - 4.2 Integrity Constraints and Enforcing Integrity Constraints Over Relations
 - 4.3 Querying Relational Data
 - 4.4 Logical Database Design
 - 4.5 Views
 - 4.6 Answers to Check Your Progress Questions
 - 4.7 Summary
 - 4.8 Key Words
 - 4.9 Self Assessment Questions and Exercises
 - 4.10 Further Readings
-

4.0 INTRODUCTION

The accuracy and consistency of the data is referred to as data integrity. It defines the relationally correct state for a database. Constraints are rules that govern the behaviour of the data at all times in a database. In order for a database to have integrity (to ensure that users perform only operations, which leave the database in a correct, consistent state), it needs to obey several rules called ‘Integrity Constraints’. You will also learn about the views in relations. A view is a basic schema object that is an access path to the data in one or more tables. It is called virtual table because they does not exist as independent entities in the database as do in real tables.

A view is a SQL query-defined virtual table. You query it in the same way as you query a table when you construct a view. When a user queries a view, the results of the query only contain data from the tables and fields defined in the view-defining query.

In this unit, you will study about the integrity constraints over relations, enforcing integrity constraints, querying relational data, logical database design, introduction to views, destroying/altering tables and views in RDBMS.

4.1 OBJECTIVES

After going through this unit, you will be able to:

- Define the integrity constraints
- Discuss the different types of integrity constraints
- Understand how constraints are applied on the relations
- Understand the concept of views

NOTES

4.2 INTEGRITY CONSTRAINTS AND ENFORCING INTEGRITY CONSTRAINTS OVER RELATIONS

Integrity means something like ‘be right’ and consistent. The data in database must be right and in good condition. Data integrity means accuracy, reliability and consistency of data which is stored in the database. Integrity constraints offers a means for ensuring that modifications made to the database by authorized users do not result in the loss of data consistency.

Need of Integrity Constraints

Many users enter data item in database which may results in inconsistencies. Therefore, data insertion, updating and other processes have to be performed in such a way that data integrity is not affected. For this integrity checks are made at data level by checking the value to certain specify rule.

Types of Data Integrity

There are different types of data integrity which are given below.

1. Domain Integrity Constraint

Domain constraint defines that the value taken by the attribute must be the atomic value from its domain. It defines the domain or set of values for an attribute. In domain integrity, you define data type, length or size, here null values are allowed, is the value unique or not for an attribute.

Consider the following employee table:

EMP_ID	NAME	AGE
E101	Amit	30
E102	Aman	32
E103	Akshay	29
E104	Rahul	Q

NOTES

Here, the domain constraint is that in EMP_ID E104, in AGE ‘Q’ is not allowed because in age attribute, only integer values can be taken.

2. Entity Integrity Constraint

Entity integrity constraint states that primary key can't be null. It ensures that a specific row in a table can be identified. There can be null values other than primary key fields. Null value is different from zero value or space. The two main properties are:

- The primary key is not null, no component of the primary key may be set to null.
- The primary key for a row is unique; it does not match the primary key of any other row in the table.

The first property ensures that the primary key has meaning, has a value; no component of the key is missing. The uniqueness property ensures that the primary key of each row uniquely identifies it; there are no duplicates. Any operation that creates a duplicate primary key or one containing nulls is rejected. The system enforces entity integrity by not allowing operations (INSERT, UPDATE) to produce an invalid primary key.

For example, consider there is a relation “EMP” Where “Emp_id” is a primary key. These must not contain any null value whereas other attributes may contain null value e.g “Age” in the following relation contains one null value.

EMP_ID	NAME	AGE
E101	Amit	30
E102	Aman	32
E103	Akshay	29
E104	Rahul	NULL

In the above table, in EMP_ID E104, in age attribute, NULL values are not allowed.

3. Referential Integrity Constraint

The referential integrity constraint is specified between two tables and it is used to maintain consistency among rows between two tables. The properties are:

- You can't delete a record from a primary table if matching record exist in a related table.
- You can't enter value in foreign key field of the related table that doesn't exist in the primary key of the primary table.
- You can't change a primary key value in the primary table if that record related record.
- You can enter a value in foreign key, specify that the record is unrelated.

NOTES

- You cannot update or delete a record of the referenced relation if the corresponding record exists in the referencing relation.
- You cannot insert a record into a referencing relation if the corresponding record does not exist in the referenced relation.

For example, consider there are two relations ‘EMP’ and ‘DEPT’ .Here, relation ‘EMP’ references the relation ‘Department’.

DEPT Table	
DEPT_NO	DEPT_NAME
D10	IT
D10	MGMT
D11	ADMISSION
D11	MARKETING

EMP Table		
EMP_ID	NAME	DEPT_NO
E101	Amit	D10
E102	Aman	D11
E103	Akshay	D11
E104	Rahul	D12

Here in the above table, the relation ‘EMP’ does not satisfy the referential integrity constraint because in relation ‘DEPT’, no value of primary key specifies department no. 12. So here, referential integrity constraint is violated.

4. Key Integrity Constraint

- In the relational table, a primary key contains a null and unique value in the relational table.
- In the entity set, there can be multiple keys and one key must be primary key.
- Key integrity constraint are the entity set which are used to detect an entity within its entity set (uniquely).

For example, given below is the employee table, which contains attributes like EMP_ID, NAME and DEPT_NO.

EMP Table		
EMP_ID	NAME	DEPT_NO
E101	Amit	D10
E102	Aman	D11
E103	Akshay	D11
E104	Rahul	D12
E101	Ananya	D13
E105	Priya	D14

NOTES

From the above table, it is clear that EMP_ID is a primary key. It means each entry in this domain should be unique. But it violates the rule.

5. Tuple Integrity Constraint

Tuple integrity constraint states that in a relational table, all the tuples must be unavoidably unique. For example, consider the below Employee table. Here, all the tuples are unique.

EMPTable

EMP_ID	NAME	DEPT_NO
E101	Amit	D10
E102	Aman	D11
E103	Akshay	D12
E104	Rahul	D15

Consider another table, STUDENT, it violates the rule because entry in row 2 and row 5 are not unique.

Student Table

STU_ID	STU_NAME	AGE
STU101	Diya	17
STU102	John	20
STU103	Smith	15
STU104	Andry	24
STU102	John	20
STU105	Pruny	14

4.3 QUERYING RELATIONAL DATA

The first step is to design the schema of the database and then create the schema using a data definition language. As we know that in a relational database the schema consists of the structure of the relations and the attributes of those relations. So for querying the relational database following are the steps.

- The structure and the attributes of the relations in relational database have to be set up inside our big disk.
- Once that's ready, the next step is to load up the database with the initial data.
- So the database to be initially loaded from data that come from an outside source. This data is just stored in files of some type or then that data could be loaded in database.
- Once the data is loaded, then there is bunch of tuples in the given relation.

- After that query and modify the data so that happens continuously over time as long as the database is in existence.

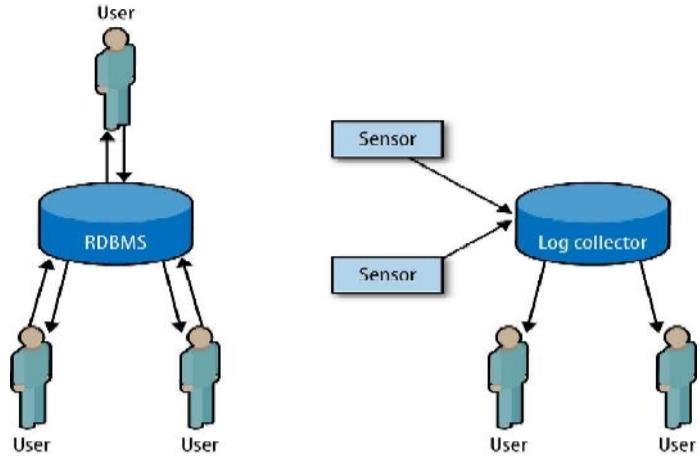


Fig. 4.1 Querying Relational Data

Relational databases support ad hoc queries and high-level languages. The user can pose queries that he didn't think of in advance. So it's not necessary to write long programs for specific queries. Rather the language can be used to pose a query, as you think about what you want to ask. As we know that the language supported by relational systems are high level, it means you can write in a fairly compact fashion rather complicated queries and you don't have to write the algorithms that get the data out of the database.

For example, consider some queries. Consider a database of students who are applying to colleges. The queries are:

- All students whose GPA > 3.7 can apply to Stanford and MIT only.
- All engineering departments in California with fewer than 500 applicants.
- College with the highest average accept rate over the last five years.

Now these might seem like a fairly complicated queries but all of these can be written in a few lines in say the SQL language or a simple expression in relational algebra. So we can say that, some queries are easier to pose than others, like above. Some queries are easier for the database system to execute efficiently than others. These two things are not necessarily correlated. There are some queries that are easy to pose but hard to execute efficiently and vice-versa.

NOTES

NOTES

The goal of logical database design is to create well-structured tables which properly reflect the company's business environment. The tables will be able to store data about the company's entities. The data should be stored in a non-redundant manner. To support all the relationships among the entities, the foreign keys will be placed in the tables so that will be supported.

After creating the logical database design, you can verify about the completion of design, its accuracy with users and management. You can check whether the design is complete it means, it consists of all of the data which must be followed and check for accuracy of data it means it redirects the correct table relationships and also imposes the business rules.

The logical database design is an iterative process. Creating a logical database design means we are doing information gathering. For logical database design there are some steps are given below:

- A logical database design describes the relationships between the tables.
- It also describes on the basis of your business requirements, define the tables which are based on the information.
- It uses normalization concept (will be discussed in unit 8) to normalize the tables that is the table should be free from redundancy and duplicity. It uses first NF, second NF, third NF to normalize the table. For further, sometimes fourth and fifth NF also used.
- Logical database design also Define the contents /columns of each table.

The role of logical database design is to define the column domain and primary key in the table. A domain can be defined as the set of valid principles for each column. For example: the domain for the employee number which contain all positive numbers related to that domain.

Relationship between Tables

In a relational database, there is a relationship between different tables and these share common columns/column. This column which is remaining in two or more tables, it allows the tables needs to be joined. Based on the rule of your business, while designing the database, the table relationship must be defined. The tables can also be associated by other non-key columns and this relationship is between foreign and primary key columns.

For example: consider the Student and Books_Order tables. In these two tables, both shows the relationship between them.

Student Table

STU_ID	STU_NAME	STU AGE
STU101	John	15
STU102	Smith	20
STU103	Danis	15

NOTES

Book_Order Table			
Order_No	STU_ID	Order_Date	Ship_Date
245007	STU101	March 2, 2019	March 4, 2019
245008	STU102	Feb 20, 2019	March 1, 2019
245009	STU103	March 2, 2019	March 4, 2019

In Student table, STU_ID acts as a primary key whereas in Book_Order table, STU_ID acts as a foreign key. So both the tables have a relationship in terms in primary and foreign key with the help of STU_ID.

If the EMP_ID is an index in both tables then we can do the following:

- Find all the orders related to the given student and query information for each order like order no, order date, shipping date etc.
- Find student information for each order using STU_ID like name and age.

4.5 VIEWS

A view or database view is a logical table or virtual table which is defined as a SQL SELECT query with the help of joins. A view also has rows and columns because these are formed using the actual tables in the database. We can create a view by selecting fields. MySQL, which allows to update the data in the given table through the database view.

Note: In order to understand this topic, it's better firstly to go through unit 7.

The following are the advantages of views.

- A database view helps limit data access to specific users. It is also used to expose only non-sensitive data to a specific group of users.
- A database view helps to hide the complexity of essential tables to the end-users and external applications.
- It enables backward compatibility after removing some tables and creating new tables and there is no need to the changes to affect other applications.
- As we know that security is a dynamic part of any relational database management system. So a database view provides extra security layer which provide further protection for a database management system.

Disadvantages of database view

There are various disadvantages of using database views:

- Whenever there is a change in the structure of the tables through which the view is associated, based on that there is a change in the view also.

NOTES

- If the view is created based on other views, querying data from a database view can be slow especially.

Creating Views

We can create view by using CREATE VIEW statement using a single table or multiple tables.

For example: consider the tables given as:

Student_Table

STU_ID	STU_NAME	STU_ADDRESS
101	John	Delhi
102	Smith	Agra
103	Diansh	Rajasthan
104	Ramjas	Haryana

Student_Marks_Table

STU_ID	STU_NAME	STU AGE	STU_MARKS
101	John	20	87
102	Smith	15	68
103	Diansh	17	57
104	Ramjas	18	59

Syntax:

```
CREATE VIEW view_name AS
    SELECT column1, column2.....
        FROM table_name
        WHERE condition;
```

Here view_name represent the name for the view, table_name represent the name of the table and condition represent the condition to select rows.

Creating View from a Single Table

For example, we will create a view named myinfo_view from student_table.

```
CREATE VIEW myinfo_view AS
    SELECT STU_NAME, STU_ADDRESS
        FROM Student_table
        WHERE S_ID < 104;
```

The output can be seen from the query:

```
SELECT * FROM Myinfo_view;
```

NOTES

OUTPUT:

STU_NAME	STU_ADDRESS
John	Delhi
Smith	Agra
Diansh	Rajasthan
Ramjas	Haryana

Now consider another example, in which we create a view named as STU_NAME from the table Student_table.

```
CREATE VIEW NAME AS
SELECT STU_ID, STU_NAME
FROM Student_table
ORDER BY STU_NAME;
```

The output can be seen using the query:

```
SELECT * FROM NAME;
```

OUTPUT:

STU_ID	STU_NAME
103	Diansh
101	John
104	Ramjas
102	Smith

Creating View from Multiple Tables

For example, consider we will create a view named as Marks_View from two tables i.e. Student_table and Student_Marks_table. To create a view from multiple tables, we can simply include multiple tables in the SELECT statement.

Syntax:

```
CREATE VIEW Marks_View AS
SELECT Student_table.STUNAME,
Student_table.STU_ADDRESS,
Student_Marks_table.STU_MARKS
FROM Student_table, Student_Marks_table
WHERE Student_table.STU_NAME =
Student_Marks_table.STU_NAME;
```

The output can be seen using the query:

```
SELECT * FROM Marks_View;
```

NOTES

OUTPUT:

STU_NAME	STU_ADDRESS	STU_MARKS
John	Delhi	87
Smith	Agra	68
Diansh	Rajasthan	57
Ramjas	Haryana	59

Deleting View

We have discussed about creating view by using single and multiple tables. But sometimes a situation arises when no more data is required. We can delete or drop a view using the DROP statement.

Syntax:

```
DROP VIEW view_name;
```

For example, if we want to delete the View Marks_View, then the query will be:

```
DROP VIEW Marks_View;
```

Updating View

To update the view, conditions given below must be satisfied. These conditions are the required one for updating the view.

- Nested queries or complex queries are not allowed while updating the view.
- The SELECT statement should not have the DISTINCT keyword.
- NOT NULL values are allowed here.
- To crate the view, we use SELECT statement. It should not include GROUP BY clause or ORDER BY clause.
- The updation in view is allowed only on a single table not on multiple tables.

We can use the CREATE OR REPLACE VIEW statement to add or remove fields from a view.

Syntax:

```
CREATE OR REPLACE VIEW view_name AS  
SELECT column1, coulmn2, ...  
FROM table_name  
WHERE condition;
```

Consider an example, if we want to update the view Marks_View and add the field STU_AGE to this view from Student_Marks_table. This can be done using the query given below:

```

CREATE OR REPLACE VIEW Marks_View AS
SELECT Student_table.STU_NAME, Student_table.STU_ADDRESS,
Student_Marks_table.STU_MARKS, Student_Marks_table.STU_AGE
FROM Student_table, Student_Marks_table
WHERE Student_table.STU_NAME= Student_Marks_
table.STU_NAME;

```

*Introduction to
Constraints and Views*

The output can be seen using the query:

```
SELECT * FROM Marks_View;
```

Output:

STU_NAME	STU_ADDRESS	STU_AGE	STU_MARKS
John	Delhi	20	87
Smith	Agra	15	68
Diansh	Rajasthan	17	57
Ramjas	Haryana	18	59

NOTES

Inserting a Row in a View

For inserting a row in a view, we can use the INSERT statement of SQL. The insertion of a row in a view can be done in the same way as we do in a table.

Syntax:

```

INSERT view_name(column1, column2 , column3,..)
VALUES(value1, value2, value3..);

```

Consider an example, we will insert a new row in the view Details_View.

```

INSERT INTO Details_View(STU_NAME, STU_ADDRESS)
VALUES("Anderson", "Delhi");

```

The output can be seen using the query:

```
SELECT * FROM Details_View;
```

Output:

STU_NAME	STU_ADDRESS
John	Delhi
Smith	Agra
Diansh	Rajasthan
Ramjas	Haryana
Anderson	Delhi

Deleting a Row from a View

For deleting a row in a view, we can use the DELETE statement of SQL. Deleting rows from a view is also as simple as deleting rows from a table.

NOTES

Syntax:

```
DELETE FROM view_name  
WHERE condition;
```

For example, we will delete the last row from the view Details_View which we have just added.

```
DELETE FROM Details_View  
WHERE STU_NAME="Anderson";
```

The output can be seen using the query:

```
SELECT * FROM DetailsView;
```

OUTPUT:

STU_NAME	STU_ADDRESS
John	Delhi
Smith	Agra
Diansh	Rajasthan
Ramjas	Haryana

Check Your Progress

1. What does integrity constraints offers?
2. Define view.

4.6 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Integrity constraints offers a means for ensuring that modifications made to the database by authorized users do not result in the loss of data consistency.
2. A view or database view is a logical table or virtual table which is defined as a SQL SELECT query with the help of joins. A view also has rows and columns because these are formed using the actual tables in the database.

4.7 SUMMARY

- The accuracy and consistency of the data is referred to as data integrity. It defines the relationally correct state for a database. Constraints are rules that govern the behaviour of the data at all times in a database.
- Domain constraint defines that the value taken by the attribute must be the atomic value from its domain.
- Entity integrity constraint states that primary key can't be null. It ensures that a specific row in a table can be identified.
- The referential integrity constraint is specified between two tables and it is used to maintain consistency among rows between two tables.

NOTES

- Tuple Integrity Constraint states that in a relational table, all the tuples must be unavoidably unique.
- Relational databases support ad hoc queries and high-level languages. The user can pose queries that he didn't think of in advance. So it's not necessary to write long programs for specific queries.
- Logical database design helps you communicate and define your business information requirements. While creating a logical database design, you should require to define each piece of information which is required to track.
- A view or database view is a logical table or virtual table which is defined as a SQL SELECT query with the help of joins. A view also has rows and columns because these are formed using the actual tables in the database.

4.8 KEY WORDS

- **Constraints:** These are rules that govern the behaviour of the data at all times in a database.
- **Referential Integrity Constraint:** A constraint that designates a column or combination of columns as FOREIGN KEY to establish a relationship between the foreign key and a specified primary or unique key called the referenced key.
- **View:** It is a basic schema object in an Oracle database, that is, an access path to the data in one or more tables.

4.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Discuss the need of integrity constraints.
2. Write the steps for querying a database.
3. Discuss the advantages and disadvantages of view.

Long Answer Questions

1. Explain the different types of integrity constraints.
2. What do you understand by logical database design? Explain.
3. How will you perform the following operations on view?
 - (i) Creating a view
 - (ii) Updating a view
 - (iii) Deleting a view

NOTES

4.10 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

UNIT 5 RELATIONAL ALGEBRA

Structure

- 5.0 Introduction
 - 5.1 Objectives
 - 5.2 Basic Operations of Relational Algebra
 - 5.3 Different Types of Join
 - 5.3.1 Natural Join
 - 5.3.2 θ -Join and Equijoin
 - 5.3.3 Semijoin
 - 5.3.4 Antijoin
 - 5.3.5 Outer Join
 - 5.4 Division-Examples of Algebra Overviews
 - 5.5 Answers to Check Your Progress Questions
 - 5.6 Summary
 - 5.7 Key Words
 - 5.8 Self Assessment Questions and Exercises
 - 5.9 Further Readings
-

NOTES

5.0 INTRODUCTION

Relational algebra is a language of procedural query, which takes relationship instances as input and yields relationship instances as output. To execute queries, it uses operators. An operator may either be binary or single. Relationships are recognised as their input and yield relationships as their output. Relational algebra is performed on a relationship recursively and intermediate outcomes are also called relationships.

Relational algebra is a formal language related to the relational model theory. In a simpler way, relational algebra is a high-level procedural language. A relational algebra operation deals with one or more relations to describe another relation without modifying the original relations.

Relational algebra is a language of procedural query, which takes relationship instances as input and yields relationship instances as output. To execute queries, it uses operators. An operator may either be binary or single. Relationships are recognised as their input and yield relationships as their output.

In this unit, you will study about the various operations in relational algebra, selection and projection, set operations, renaming, joins, different types of joins, division, and examples of algebra.

NOTES

5.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the basic concept of relational algebra
 - Perform the basic operations of relational algebra
 - Understand the different types of join
-

5.2 BASIC OPERATIONS OF RELATIONAL ALGEBRA

In pure mathematics, Relational Algebra has an algebraic structure, that has relevance to set theory and mathematical logic.

Relational Algebra received attention after relational model of database was published in 1970, by Codd, who proposed this algebra as a foundation for database query languages. Relational Algebra, is widely used in, and now, is a part of computer science. It is based on algebra of sets and is an extension of first-order logic. It is concerned with a set of relations, closed under operators, which operate on one or more relations, yielding another relation.

Relational Algebra has similar power of expression as relational calculus and first-order logic. To avoid a mismatch between relational calculus and algebra, Codd restricted the operands of this algebra to finite relations only and he put restricted support for NOT and OR operators. Similar restrictions are also found in other computer languages that use logic-based constructs. A new term was defined by Codd as '*Relational Completeness*' which refers to attributes of a language that is complete with respect to first-order predicate calculus, and also follows restrictions imposed by Codd.

Basic Operations

There are few operators in any algebraic systems which are basic or primitive, while others are defined in terms of these basic ones. Codd made a similar arbitrary choice for his algebra. There are six primitive operators of relational algebra, proposed by Codd. These are:

1. Selection (Unary)
2. Projection (Unary)
3. Cartesian Product (also called the cross product or cross join) (Binary)
4. Set Union (Binary)
5. Set Difference (Binary)
6. Rename (Unary)

NOTES

These six operators are fundamental and omission of any one loses the expressive power of relational expressions. Many other operators have been defined in terms of these six. Among the most important are set operations, set-intersection, division, assignment, and the natural join.

The operations of Relational Algebra are Selection, Projection, Set Operations, Rename, Joins and Division.

First four operations are being dealt with here and last two, joins and division are discussed under ‘additional operations’. Set operations contain union, intersection, set difference and product.

Notations Used

In relational algebra, symbols are used to denote operations. Such symbols, taken from symbols of Greek alphabet, are difficult to use in HyperText Markup Language or HTML. Actual name using capital letters are used for this. These symbols are represented in the Table 5.1.

Table 5.1 Symbols in Relational Algebra

Operation	My HTML	Symbol	Operation	My HTML	Symbol
Projection	PROJECT	π	Cartesian product	X	\times
Selection	SELECT	σ	Join	JOIN	\bowtie
Renaming	RENAME	ρ	Left outer join	LEFT OUTER JOIN	\bowtie_L
Union	UNION	\cup	Right outer join	RIGHT OUTER JOIN	\bowtie_R
Intersection	INTERSECTION	\cap	Full outer join	FULL OUTER JOIN	\bowtie_F
Assignment	\leftarrow	\leftarrow	Semijoin	SEMIJOIN	\bowtie_S

Relational algebra is a high level procedural language. It is used to instruct the DBMS how to build a new relation from one or more relations in the database. In order to implement a DBMS, there must be a set of rules which state how the database system will behave. Relational databases are more usually defined using Relational Algebra. In terms of DBMS, Relational Algebra can be:

- A description which denotes how a relational database operates.
- An interface which checks the data to be stored in the database itself.
- A mathematical representation which uses SQL operations.

Operators in Relational Algebra are not necessarily the same as SQL operators even if they have the same name. For example, the SELECT statement in SQL also exists in Relational Algebra but the uses of SELECT are not the same. The DBMS must take whatever SQL statements the user types in and translate them

NOTES

into Relational Algebra operations before applying them to the database. Table 5.2 summarizes the notations used in Relational Algebra with reference to SQL statement.

Table 5.2 Notations used in Relational Algebra and SQL Statements

Relational Algebra	SQL
Restrict: $\sigma_{\text{condition}}(R)$	<code>SELECT * FROM R WHERE condition</code>
Project: $\pi_{\text{list of attributes}}(R)$	<code>SELECT list of attributes FROM R</code>
Cartesian Product: $R \times S$	<code>SELECT * FROM R, S</code>
Join	
Theta/Equi Join: $R \bowtie \text{condition } S$	<code>SELECT * FROM R, S WHERE condition</code> (Oracle 8i) or <code>SELECT * FROM R INNER JOIN S ON condition</code> (Oracle 9i)
Natural Join: $R * S$	<code>SELECT list of all attributes without duplicates</code> <code>FROM R, S WHERE condition</code> (Oracle 8i) or <code>SELECT * FROM R NATURAL JOIN S</code> (Oracle 9i)
Outer Join	
Left Outer Join: $R \bowtie L condition S$	<code>SELECT * FROM R, S WHERE R.x = S.y (+)</code> (Oracle 8i) or <code>SELECT * FROM R LEFT OUTER JOIN S ON condition</code> (Oracle 9i)
Right Outer Join: $R \bowtie R condition S$	<code>SELECT * FROM R, S WHERE R.x (+) = S.y</code> (Oracle 8i) or <code>SELECT * FROM R RIGHT OUTER JOIN S ON condition</code> (Oracle 9i)
Full Outer Join: $R \bowtie F condition S$	<code>SELECT * FROM R, S WHERE R.x = S.y (+) UNION</code> <code>SELECT * FROM R, S WHERE R.x (+) = S.y</code> (Oracle 8i) or <code>SELECT * FROM R FULL OUTER JOIN S ON condition</code> (Oracle 9i)
Set Operations	
Union: $R \cup S$	<code>SELECT * FROM R UNION SELECT * FROM S</code>
Intersection: $R \cap S$	<code>SELECT * FROM R INTERSECT SELECT * FROM S</code>
Union: $R - S$	<code>SELECT * FROM R MINUS SELECT * FROM S</code>

Note: `SELECT` command is used in SQL for querying data.

Selection

A **generalized selection** is a unary operation. This is written as a propositional formula consisting of atoms in the normal selection and with logical operator conjunction, disjunction and negation. This selection selects all those tuples in R for which holds.

In relational algebra, a selection is written as $\sigma_a b(R)$ or $\sigma_a v(R)$ where:

- σ stands for selection
- a and b denote attribute names
- θ shows binary operation

- v denotes a value constant
- R denotes relation

Relational Algebra

The selection $\sigma_{a \theta b}(R)$ selects all tuples in R for which θ holds between attributes ‘ a ’ and ‘ b ’.

The selection $\sigma_{a \theta v}(R)$ selects all tuples in R for which θ holds between the attribute ‘ a ’ and the value v .

For example, consider the following tables in which the first table is about the relation named *Person*, the second table shows the result of $\sigma_{Age \geq 34}(Person)$ and the third table has the result for $\sigma_{Age = Weight}(Person)$.

Person			$\sigma_{Age > 34}(Person)$			$\sigma_{Age = Weight}(Person)$		
Name	Age	Weight	Name	Age	Weight	Name	Age	Weight
Rohan	34	70	Rohan	34	70	Sonia	50	54
Manju	25	55	Sonia	50	54			
Sohan	29	70	Patel	32	60			
Sonia	50	54						
Patel	32	60						

Semantics of the selection is shown mathematically as:

$$\sigma_{a \theta b}(R) = \{t : t \in R, t(a) \theta t(b)\}$$

$$\sigma_{a \theta v}(R) = \{t : t \in R, t(a) \theta v\}$$

Projection

A **projection** is mathematically written as $\pi_{a_1, \dots, a_n}(R)$, where a_1, \dots, a_n is a set of attribute names. It is a unary operation. The resultant set from such operation is a set obtained when all tuples in R contain the set $\{a_1, \dots, a_n\}$. All other attributes are discarded.

For example, if we use two attributes, Name and Age, written as (Name, Age), then projection of the relation $\{(Raman, 5), (Ratan, 8)\}$ attribute field list (Age), yields $\{5, 8\}$ and age are discarded. It only gives the value of the field age. If we project $(5, 8)$ on second component will give 8.

Projection in relational algebra is like a counterpart of existential quantification in predicate logic. Existentially quantified variables are attributes, *excluded* corresponding to existentially quantified variables in the predicate and their extensions are represented by its projection. Thus, projection is defined as excluded attributes. In Information Systems Base Language or ISBL notations for both have been provided. Other languages have followed ISBL.

NOTES

Relational Algebra

NOTES

Example of this concept exists in category of monoids. Projection of a string, with removal of one or more letters of alphabet in the string is a monoid and is a projection as well.

For example, there are two relations presented in the following two tables. One is the relation *Person* and its projection on the attributes *Age* and *Weight*.

Person			Age,Weight(Person)	
Name	Age	Weight	Age	Weight
Hari	30	70	30	70
Shila	26	62	26	62
Ganesh	27	75	27	75
Sonia	50	55	50	55
Patel	30	70		

If name is *N*, age is *A* and weight is *W*, then predicate is, ‘*N* is *A* years old and weighs *W*.’ Projection of this is the predicate, ‘There exists *N* such that *N* is *A* years old and weighs *W*.’

In this example, Hari and Patel are of the same age and they have same weight, but (Name, Age) combination appears only once in the result. This is so because it is a relation.

Mathematically, semantics of projection are expressed as follows:

$$\pi_{a_1 \dots a_n}(R) = \{t[t_{a_1 \dots a_n}]: t \in R\}$$

Where, $t[a_1, \dots, a_n]$ indicates the restriction of the tuple t to the set $\{a_1, \dots, a_n\}$ which is mathematically represented as;

$$t[a_1, \dots, a_n] = \{(a'_v) | (a'_v) \in t, a'_v \in a_1, \dots, a_n\}$$

Such a projection, $\pi_{a_1 \dots a_n}(R)$ is defined only if $\{a_1, \dots, a_n\} \subseteq \text{Header}(R)$. The header contains attributes and $\text{Header}(R)$ is a set of all attributes. Projection on NULL attribute is also possible. This yields a relation of degree zero.

Relational algebra is identical power of expression, like that of domain relational calculus or tuple relational calculus, but it has less expressive power than first-order predicate calculus without function symbols. Expression wise, Relational Algebra is a subset of first-order logic. These represent Horn clauses with no recursion and no negation.

Set Operators

Three basic operators have been taken from six basic operators of set theory, but with some differences. Some additional constraints are present in their form as adopted in relational algebra. For operations of union and difference, these two relations must have the same set of attributes which is a property known as union-

compatibility. The operation of difference, set intersection is used and this too should be union-compatible.

The Cartesian product is also adopted with a difference from that in set theory. Here tuples are taken to be ‘shallow’ for the purposes of the operation. Unlike set theory, the 2-tuple of Cartesian product in relational algebra has been ‘flattened’ into an $(n + m)$ tuple. Mathematically, $R \times S$ is expressed as follows:

$$R \times S = \{r \cup s \mid r \in R, s \in S\}$$

In relational algebra, the two relations involved in Cartesian product must have disjoint headers (unlike that of union and difference operations). Here R and S should be disjoint and hence, should not have any common attribute.

Rename

A rename operation is mathematically expressed as $\rho_{a/b}(R)$. This too, is a unary operation. Result of $\rho_{a/b}(R)$ is renaming of the ‘ b ’ field, in all tuples by an ‘ a ’ field. This operation is used to rename the attribute of a relation or even the relation, R , itself.

Relational Algebra and SQL Examples

Table 5.3 summarizes the selected fields (Ac_Number and Amount) of ‘Emp_eTrans’ Table which is used to display the result of SQL statements and their corresponding Relational Algebra statements.

Table 5.3 Ac_Number and Amount Fields in Emp_eTrans Table

Ac Number	Amount
1212	23000
3434	26000
5656	66000
7878	33000
5151	20000

The SQL statements and their corresponding Relational Algebra statements will be written in the Table 5.4 and the result is produced as data provided in ‘Emp_eTrans’ Table.

Table 5.4 SQL, Corresponding Relational Algebra Statements and Result

SQL	Result	Relational Algebra												
select Amount from Emp_eTrans	<table border="1"> <thead> <tr> <th>Amount</th> </tr> </thead> <tbody> <tr> <td>23000</td></tr> <tr> <td>26000</td></tr> <tr> <td>66000</td></tr> <tr> <td>33000</td></tr> <tr> <td>20000</td></tr> </tbody> </table>	Amount	23000	26000	66000	33000	20000	$\text{PROJECT}_{\text{Amount}}(\text{Emp_eTrans})$						
Amount														
23000														
26000														
66000														
33000														
20000														
select Ac_Number, Amount from Emp_eTrans	<table border="1"> <thead> <tr> <th>Ac Number</th> <th>Amount</th> </tr> </thead> <tbody> <tr> <td>1212</td> <td>23000</td></tr> <tr> <td>3434</td> <td>26000</td></tr> <tr> <td>5656</td> <td>66000</td></tr> <tr> <td>7878</td> <td>33000</td></tr> <tr> <td>5151</td> <td>20000</td></tr> </tbody> </table>	Ac Number	Amount	1212	23000	3434	26000	5656	66000	7878	33000	5151	20000	$\text{PROJECT}_{\text{Ac_Number, Amount}}(\text{Emp_eTrans})$
Ac Number	Amount													
1212	23000													
3434	26000													
5656	66000													
7878	33000													
5151	20000													

Note: There are no duplicate rows in the result.

NOTES

NOTES**Operations in Relational Algebra**

Operations in relational algebra include Write and Retrieval.

1. Write Operation Uses the Following Operators

- (a) **INSERT**: This operator provides a list of attribute values for a new tuple in a relation.
- (b) **DELETE**: This operator determines which tuple(s) to be removed from the relation.
- (c) **MODIFY**: This operator changes the values of one or more attributes in one or more tuples of a relation.

2. Retrieval Operation has Two Types of Operators

- (a) Mathematical set theory based relations

These include UNION, INTERSECTION, DIFFERENCE and CARTESIAN PRODUCT operators.

- (b) Special database operations

These include SELECT (not the same as SQL SELECT), PROJECT and JOIN operators.

5.3 DIFFERENT TYPES OF JOIN

Three basic operators have been taken from six basic operators of the set theory, but with some differences. Some additional constraints are present in their forms as adopted in Relational Algebra. For operations of union and difference, these two relations must have the same set of attributes which is a property known as union-compatibility. In the operation of difference, set intersection is used and this too should be union-compatible.

The Cartesian product is also adopted differently from the set theory. Here, tuples are taken to be ‘shallow’ for the purposes of the operation. Unlike the set theory, the 2-tuple of Cartesian product in Relational Algebra has been ‘flattened’ into an $n+m$ -tuple. Mathematically, $R \times S$ is expressed as follows:

$$R \times S = \{r \cup s \mid r \in R, s \in S\}$$

In Relational Algebra, the two relations involved in the Cartesian product must have disjoint headers (unlike that of union and difference operations). Here, R and S should be disjoint and, hence, should not have any common attribute.

Rename

A rename operation is mathematically expressed as $_{a/b}(R)$. This too, is a unary operation. The result of $_{a/b}(R)$ is renaming of the ‘b’ field, in all tuples by an ‘a’ field. This operation is used to rename the attribute of a relation or even the relation, R, itself.

5.3.1 Natural Join

Relational Algebra

It is a binary operator, written as $(R \bowtie S)$, where R and S are two relations. The result of this operation is the set of all tuple-combinations in R and S equal on their common attribute names. For example, natural join of two tables named *Employee* and *Dept* is $(\text{Employee} \bowtie \text{Dept})$ as shown in Table 5.5.

NOTES

Table 5.5 Two Natural Join

Employee			Dept		Employee \bowtie Dept			
Name	ID	Dept_N	Dept_N	Manager	Name	ID	Dept_N	Manager
Hari	3411	Finance	Finance	Ganesh	Hari	3411	Finance	Ganesh
Shalini	2242	Sales	Sales	Hemant	Shalini	2242	Sales	Hemant
Ganesh	3403	Finance	Production	Charu	Ganesh	3403	Finance	Ganesh
Hemant	2207	Sales			Hemant	2207	Sales	Hemant

The natural join is the relational counterpart of logical AND, and has great importance. The natural join permits combination of relations associated by a foreign key. For example, in the previous table, a foreign key probably holds from *Employee.Dept_N* to *Dept.Dept_N*, and then the natural join of *Employee* and *Dept* combines every employee with their respective department. Note that this works because the foreign key holds between attributes with the same name. If this is not the case such as in the foreign key from *Dept.Manager* to *Emp.ID*, then these columns have to be named before the natural join is taken up. Such a join is also known as an *equijoin*. Semantics of the natural join are given as follows:

$$R \bowtie S = \{t \cup s \mid t \in R \wedge s \in S \wedge p(t \cup s)\}$$

where p is a predicate having truth value T for a binary relation denoted as r if r is a functional binary relation. One constraint is put; and this requires that R and S should have at least one common attribute. In the absence of this constraint, the natural join becomes a Cartesian product.

5.3.2 θ -Join and Equijoin

Let there be two tables named *Car* and *Boat*. These tables list models of cars and boats against their respective prices. Suppose that a customer intends to buy a car and a boat without spending more money for the boat in comparison to the car. A θ -join using the relation ‘*CarPrice e BoatPrice*’ will give a table having all the possible options. Table 5.6 summarizes it.

Table 5.6 θ -Join and Equijoin**NOTES**

<i>Car</i>		<i>Boat</i>		<i>Car</i> \bowtie <i>Boat</i> <i>Car Price</i> \geq <i>Boat Price</i>			
CarModel	CarPrice	BoatModel	BoatPrice	CarModel	CarPrice	BoatModel	BoatPrice
CarA	20'000	Boat1	10'000	CarA	20'000	Boat1	10'000
CarB	30'000	Boat2	40'000	CarB	30'000	Boat1	10'000
CarC	50'000	Boat3	60'000	CarC	50'000	Boat1	10'000
				CarC	50'000	Boat2	40'000

If you want to combine tuples from two relations where the combination condition is not simply the equality of shared attributes, then it is convenient to have a more general form of join operator, which is the θ -join (or theta-join). The θ -join is a binary operator that is written as:

$R \bowtie S$ or $R \bowtie^{\theta} S$ where a and b are attribute names, \bowtie is a binary relation that belongs to the set $\{<, d', =, >, e''\}$, v is a value constant, and R and S are relations. This operation gives all combinations of tuples in R and S satisfying the relation θ . θ -join is defined for disjoint headers of S and R .

A fundamental operation is as follows:

$$R \bowtie_{\phi} S = \sigma_{\phi}(R \times S)$$

θ -join becomes equijoin if operator \bowtie becomes the equality operator ($=$). θ -join is not needed for a computer language that supports the natural join and rename operators, since this can be achieved by selection from the result of a natural join. This degenerates to a Cartesian product in the absence of shared attributes.

5.3.3 Semijoin

The semijoin is written as $R \bowtie^{\theta} S$ where R and S are relations. It is similar to the natural join and its result is only the set of all tuples in R for which there is a tuple in S , and it is equal on their common attribute names. For example, there are two tables named *Employee* and *Dept* and semijoin of these two follows in Table 5.7.

Table 5.7 Semijoin*Relational Algebra*

Employee			Dept		Employee Dept		
Name	EmpId	DeptName	DeptName	Manager	Name	EmpId	DeptName
Hari	3411	Finance	Sales	Hemant	Shalini	2242	Sales
Shalini	2242	Sales	Production	Charu	Hemant	2207	Production
Ganesh	3403	Finance					
Hemant	2207	Production					

NOTES

More formally, the semantics of the semijoin is defined as follows:

$$R \bowtie S = \{ t : t \in R, s \in S, \text{fun}(t \cup s) \}$$

where $\text{fun}()$ gives the definition of natural join.

The semijoin can follow from the natural join as given here:

$$R \bowtie S = \Pi_{a_1, \dots, a_n} (R \bowtie S), \text{ where, } a_1, \dots, a_n \text{ are the attribute names of } R.$$

Natural join can be stimulated from basic operators and hence this is also applicable for semijoin.

5.3.4 Antijoin

The antijoin, written as $R \triangleright S$ where R and S are relations. It has similarity to the natural join. The result of an antijoin contains only those tuples in relation R which are NOT a tuple in S that is equal on their common attribute names.

For example, in the Table 5.8 antijoin of two tables *Employee* and *Dept* are:

Table 5.8 Antijoin

Employee			Dept		Employee \triangleright Dept		
Name	EmpId	DeptName	DeptName	Manager	Name	EmpId	DeptName
Hari	3411	Finance	Sales	Hari	Hari	3411	Finance
Shalini	2242	Sales	Production	Charu	Ganesh	3403	Finance
Ganesh	3403	Finance					
Hemant	2207	Production					

NOTES

Formal definition of antijoin is:

$$R \triangleright S = \{ t : t \in R \wedge \neg \exists s \in S : \text{fun}(t \cup s) \}$$

or

$R \triangleright S = \{ t : t \in R, \text{ there is no tuple } s \text{ of } S \text{ that satisfies } \text{fun}(t \cup s) \}$ where $\text{fun}(\)$ is same as for natural join.

The antijoin is the complement of the semijoin, and is expressed as follows:

$$R \triangleright S = R - R \bowtie S$$

Thus, antijoin is the anti-semijoin, and the antijoin operator is expressed as semijoin symbol with a bar above it, in place of \triangleright .

5.3.5 Outer Join

OUTER JOIN is used to join two tables even if there is not a match. An OUTER JOIN can be used to return a list of all the customers and the orders even if no orders have been placed for some of the customers. A keyword, RIGHT or LEFT, is used to specify which side of the join returns all possible rows. LEFT is used because it makes sense to list the most important table first. Except for one example demonstrating the RIGHT OUTER JOIN, we will use left joins. The keywords INNER and OUTER are optional.

This example returns a list of all the customers and the SalesOrderID for the orders that have been placed, if any.

```
SELECT c.CustomerID, s.SalesOrderID
FROM Sales.Customer c
LEFT OUTER JOIN Sales.SalesOrderHeader s ON
c.CustomerID = s.CustomerID
```

It uses the LEFT keyword because the Sales.Customer table is located on the left side and we want all rows returned from that table even if there is no match in the Sales.SalesOrderHeader table. This is an important point. Notice also that the CustomerID column is the primary key of the Sales.Customer table and a foreign key in the Sales.SalesOrderHeader table. This means that there must be a valid customer for every order placed. Writing a query that returns all orders and the customers, if they match, does not make sense. The LEFT table should always be the primary key table when performing a LEFT OUTER JOIN.

If the location of the tables in the query are switched, the RIGHT keyword is used and the same results are returned:

```
SELECT c.CustomerID, s.SalesOrderID
FROM Sales.SalesOrderHeader s
RIGHT OUTER JOIN Sales.Customer c ON c.CustomerID =
s.CustomerID
```

LEFT OUTER JOIN

Relational Algebra

Returns all the rows from R_1 even if there are no matches in R_2 . If there are no matches in R_2 then the R_2 values will be shown as null.

```
SELECT * FROM R1 LEFT [OUTER] JOIN R2 ON R1.field = R2.field
```

NOTES

RIGHT OUTER JOIN

Returns all the rows from R_2 even if there are no matches in R_1 . If there are no matches in R_1 , then the R_1 values will be shown as null.

```
SELECT * FROM R1 RIGHT [OUTER] JOIN R2 ON R1.field = R2.field
```

FULL OUTER JOIN

Returns all the rows from both tables even if there are no matches in one of the tables. If there are no matches in one of the tables then its values will be shown as null.

```
SELECT * FROM R1 FULL [OUTER] JOIN R2 ON R1.field = R2.field
```

LEFT/RIGHT/FULL OUTER JOIN

An inner join excludes rows from either table that do not have a matching row in the other table. An outer join combines the unmatched row in one of the tables with an artificial row (all columns set to null) for the other table:

```
SELECT * FROM  
table-1 [ NATURAL ] { LEFT | RIGHT | FULL } OUTER JOIN  
table-2  
[ ON join_condition | USING ( join_column_list ) ]
```

Join Types

- **LEFT:** Only unmatched rows from the left side table (table-1) are retained.
- **RIGHT:** Only unmatched rows from the right side table (table-2) are retained.
- **FULL:** Unmatched rows from both tables (table-1 and table-2) are retained.

LEFT OUTER JOIN returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its ON condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the ON condition. This left-hand row is extended to the full width of the joined table by inserting NULLs for the right-hand columns.

RIGHT OUTER JOIN returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left). This is just a notational convenience, since you could convert it to a **LEFT OUTER JOIN** by switching the left and right inputs.

NOTES

FULL OUTER JOIN returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

LEFT OUTER JOIN example ‘project-percentage assignment’:

```
SELECT assign.project, projects.name, assign.percentage
  FROM projects LEFT OUTER JOIN assign
    ON projects.id = assign.project ;

  project |   name   | percentage
-----+-----+
      1 | compiler |      10
      1 | compiler |      60
      1 | compiler |      30
      2 | xpaint   |      50
      2 | xpaint   |      50
      3 | game     |      70
      3 | game     |      30
      | perl     |        |
(8 rows)
```

5.4 DIVISION- EXAMPLES OF ALGEBRA OVERVIEWS

There are different types of extended operators in Relational Algebra:

- **Join**
- **Intersection**
- **Divide**

The relations used to understand extended operators are EMP_SPORTS, WHOLE_SPORTS as shown in Table 5.9 and Table 5.10 respectively.

Table 5.9

STUDENT_SPORTS

ID	SPORTS
1	Badminton
2	Cricket
2	Badminton
4	Badminton

Table 5.10

WHOLE_SPORTS

SPORTS
Bdminton
Cricket

Division Operator (\div): The division operator $X \div Y$ can be applied if and only if:

- The attributes of Y is the proper subset of Attributes of X.
- The relation returned by division operator (All attributes of X – All Attributes of Y) will have attributes.
- The output by division operator will give those tuples from relation X which are associated to every Y's tuple.

For example consider the relation EMP_SPORTS and WHOLE_SPORTS given in Table 5.9 and Table 5.10 above.

To apply division operator as

EMP_SPORTS \div WHOLE_SPORTS

- Here the operation applied is valid as attributes in WHOLE_SPORTS is a proper subset of attributes in EMP_SPORTS.
- The resulting relation will have attributes {ID,SPORTS} - {SPORTS} = ID.
- The tuples in output relation consists of ID which are related with all Y's tuple {Badminton, Cricket}. ID 1 and 4 which are related to Badminton only. ID 2 is related to all tuples of Y. So the output will be:

ID
2

Check Your Progress

1. What is the use of symbols in relational algebra?
2. When θ -join becomes equijoin?
3. How is natural join also applicable for semi-join?
4. Why is OUTER JOIN used?

5.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. In relational algebra, symbols are used to denote operations.
2. A generalized selection is a unary operation. It is written as a propositional formula consisting of atoms in the normal selection and with logical operator conjunction, disjunction and negation.
3. θ -join becomes equijoin if operator θ becomes the equality operator (=).
4. Natural join can be simulated from basic operators and hence this is also applicable for semi-join.

NOTES

NOTES**5.6 SUMMARY**

- Relational Algebra has similar power of expression as relational calculus and first-order logic. To avoid a mismatch between relational calculus and algebra, Codd restricted the operands of this algebra to finite relations only and he put restricted support for NOT and OR operators. Similar restrictions are also found in other computer languages that use logic-based constructs.
- The operations of Relational Algebra are Selection, Projection, Set Operations, Rename, Joins and Division.
- Relational algebra is a high level procedural language. It is used to instruct the DBMS how to build a new relation from one or more relations in the database. In order to implement a DBMS, there must be a set of rules which state how the database system will behave.
- A generalized selection is a unary operation. This is written as a propositional formula consisting of atoms in the normal selection and with logical operator conjunction, disjunction and negation.
- A **projection** is mathematically written as $\pi_{a_1, \dots, a_n}(R)$, where a_1, \dots, a_n is a set of attribute names. It is a unary operation.
- A rename operation is mathematically expressed as $\rho_{a/b}(R)$. This too, is a unary operation.
- Natural join is a binary operator, written as $(R \ S)$, where R and S are two relations. The result of this operation is the set of all tuple-combinations in R and S equal on their common attribute names.
- The semijoin is written as where R and S are relations. It is similar to the natural join and its result is only the set of all tuples in R for which there is a tuple in S , and it is equal on their common attribute names.
- **OUTER JOIN** is used to join two tables even if there is not a match. An OUTER JOIN can be used to return a list of all the customers and the orders even if no orders have been placed for some of the customers.

5.7 KEY WORDS

- **Right Outer Join:** A join where records from the right table that have no matching key in the left table are included in the result set.
- **Full Outer Join:** A join where records from the first table are included that have no corresponding record in the other table and records from the other table are included that have no records in the first table.
- **MINUS:** As defined in relational algebra, it is the set difference of two relations.

5.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What is a relational algebra?
2. List the notations used in relational algebra.
3. What is natural join?
4. Name the various types of join.

Long Answer Questions

1. What are the basic operations of relational algebra?
2. How queries are built with the relational algebra? Explain with the help of a diagram.
3. Explain the different types of joins with the help of examples.
4. Explain the concept of q-join and equijoin with the help of examples.

NOTES

5.9 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications) P Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

NOTES

UNIT 6 RELATIONAL CALCULUS

Structure

- 6.0 Introduction
 - 6.1 Objectives
 - 6.2 Relational Calculus
 - 6.2.1 Tuple Relational Calculus
 - 6.2.2 Domain Relational Calculus
 - 6.3 Expressive Power of Algebra and Calculus
 - 6.4 Answers to Check Your Progress Questions
 - 6.5 Summary
 - 6.6 Key Words
 - 6.7 Self Assessment Questions and Exercises
 - 6.8 Further Readings
-

6.0 INTRODUCTION

Relational algebra and Relational calculus are the languages formally equivalent to one another. In a calculus expression, there is no sequence of operations specified in order to retrieve the query result. It specifies only what information the result should contain.

Relational algebra query specifies how data can be retrieved from a relation but the relational calculus specifies what is to be retrieved from a relation.

The Relational calculus consists of two equations, the relational calculus of the tuple and the relational calculus of the scope, which are part of the database relational model and provide a declarative way to define queries from the database.

In this unit, you will study about the relational calculus, tuples relational calculus, domain relational calculus, and expressive power of algebra and calculus.

6.1 OBJECTIVES

After going through this unit, you will be able to:

- Learn about the relational calculus
 - Discuss the types of relational calculus
 - Discuss the expressive power of algebra and calculus
-

6.2 RELATIONAL CALCULUS

Relational calculus is a non-procedural query language. It describes what to do but never explains how to do. Since, it's a non-procedural query language so here the user is concerned to obtain the end results with proper explanation.

The relational calculus is not the same as that of integral and differential calculus in mathematics, but it's a form of predicate calculus. It is expressed in two forms when implemented on the databases. There are two types of Relational calculus i.e.

1. Tuple relational calculus (proposed by Codd in 1972).
2. Domain relational calculus (proposed by Lacroix and Pirotte in 1977).

In predicate calculus or first-order logic, a predicate is defined as a truth-valued function with argument and for the arguments, when we replace with values then the function gives an expression which is known as proposition. The proposition has two values either true or false.

For example, consider the list all the ‘students’ who attend the ‘DBMS’ class, can be expressed as:

SELECT the tuples from SUBJECT relation with SUBJECT_NAME =‘DBMS’

6.2.1 Tuple Relational Calculus

We can write a query in Tuple Calculus as:

$\{t | P(t)\}$

The above expression produces set of all tuples t for which the Predicate $P(t)$ is true for t.

Here, t represents the resulting tuples, whereas $P(t)$ represents as Predicate. These are the conditions which are used to fetch t. you can use logical operators i.e. there are several conditions of $P(t)$ which can be combined with these operators.

A Tuple Relational Calculus can also uses quantifiers in the form of:

1. $\exists t \in r (Q(t))$
 2. $\forall t \in r (Q(t))$
- The first expression indicates that there exists (\exists), a tuple in t in relation r such that predicate $Q(t)$ is true.
 - The second expression indicates that $Q(t)$ is true for all (\forall) tuples in relation r .

For example, consider the tables given below:

Table 6.1 EMPLOYEE TABLE

EMP_NAME	EMP_STREET	EMP_CITY
Amit	D10	Delhi
Ashish	C5	Punjab
Priya	B8	Haryana

NOTES

NOTES

Table 6.2 ACCOUNT_BANK TABLE

AC_NO	AMT_BALANCE	BRANCH_NAME
AC11001	10000	XYZ
AC11002	40000	ABC
AC11003	8000	PQR

Table 6.3 BRANCH_DETAILS TABLE

BRANCH_NAME	BRANCH_CITY
XYZ	Punjab
ABC	Ludhiana
PQR	Haryana

Table 6.4 LOAN_DETAILS TABLE

LOAN_NO	BRANCH_NAME	AMOUNT
L20	PQR	10000
L24	ABC	5000
L36	XYZ	20000

Table 6.5 DEPOSITOR_DETAILS TABLE

EMP_NAME	AC_NO
Priya	AC11003
Ashish	AC11001
Amit	AC11002

Table 6.6 BORROWER_DETAILS TABLE

EMP_NAME	LOAN_NO
Priya	L24
Ashish	L36
Amit	L20

The following queries explain the concept of tuple relational calculus using the tables given above.

Query-1: Find the branch_name, loan_no, amount of loans of greater than or equal to 5000 amount.

$$\{t \mid t \in \text{loan_details} \wedge t[\text{amount}] \geq 5000\}$$

Output:

LOAN_NO	BRANCH_NAME	AMOUNT
L36	XYZ	20000

t[amount] is known as tuple variable.

Query-2: Find the loan_no for each loan of an amount greater or equal to 10000.

$$\{t \mid \exists s \in \text{loan_details}(t[\text{loan_no}] = s[\text{loan_no}] \wedge s[\text{amount}] \geq 10000)\}$$

Output:*Relational Calculus*

LOAN_NO
L20
L24
L36

NOTES

Query-3: Find the names of all employees who have an account in bank and have loan.

$$\{t \mid \exists s \in \text{borrower_details}(t[\text{emp_name}] = s[\text{emp_name}]) \\ \wedge \exists u \in \text{depositor_details}(t[\text{emp_name}] = u[\text{emp_name}])\}$$

Output:

EMP_NAME
Priya
Ashish
Amit

Query-4: Find the names of all employees having a loan at the “PQR” branch.

$$\{t \mid \exists s \in \text{borrower_details}(t[\text{emp_name}] = s[\text{emp_name}]) \\ \wedge \exists u \in \text{loan_details}(u[\text{emp_name}] = "PQR" \wedge u[\text{loan_no}] = s[\text{loan_no}])\}$$

Output:

EMP_NAME
Amit

6.2.2 Domain Relational Calculus

Another type of relational calculus which instead of using values from an individual tuple, uses values from an attribute domain. These attribute domain values in domain relational calculus are taken up by the domain variables. An expression in the domain relational calculus is of the form:

$$\{x_1, x_2, \dots, x_n \mid P(x_1, x_2, \dots, x_n)\}$$

Here, x_1, x_2, \dots, x_n represent domain variables. P represents a formula consists of atoms. This is the same case as in the tuple relational calculus.

In the domain relational calculus, an atom can be represented in the following forms:

If x is a domain variable, c is a constant in the domain of the attribute, \in is a comparison operator for which x is a domain variable can be in the form of $x \in c$.

If \in is a comparison operator ($<$, \leq , $=$, $/=$, $>$, \geq), x and y are domain variables and we want that attributes x and y have domains that can be related by \in can be represented as $x \in y$.

If and x_1, x_2, \dots, x_n are domain variables or domain constants and r is a relation on n attributes then this can be represented as $\langle x_1, x_2, \dots, x_n \rangle \in r$.

Rules to build up formulae from atoms:

NOTES

- An atom is a formula:
- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $P_1 \Rightarrow P_2$.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1) .
- If $P_1(x)$ is a formula in x , where x is a domain variable, then

We are using some queries for the examples that we considered earlier. Note the resemblance of these expressions and the matching tuple- relational-calculus expressions.

Query-1 Find the branch name, loan number, and amount for loans of over 10000.

$\{ \langle l, b, a \rangle | \langle l, b, a \rangle \in \text{loan_no} \wedge a > 10000 \}$

Output:

LOAN_NO	BRANCH_NAME	AMOUNT
L20	PQR	10000
L36	XYZ	20000

Query-2 Find all loan numbers for loans with an amount greater than 10000.

$\{ \langle l \rangle | \exists b, a (\langle l, b, a \rangle \in \text{loan_no} \wedge a > 10000) \}$

Output:

LOAN_NO
L20
L36

Even though the second query seems to be alike to the one which we wrote for the tuple relational calculus but it differs. In the tuple calculus, When we write $\exists b$ in the domain calculus, b refers not to a tuple, but rather to a domain value. We bind it immediately to a relation by writing $\exists s \in r$, when we write $\exists s$ for some tuple variables. So, here the domain of variable b is unrestricted until the subformula $\langle l, b, a \rangle \in \text{loan}$ constrains b to `branch_name` that appear in the loan relation.

Query-3 Find the names of all employees who have a loan from the PQR branch and find the loan amount.

$\{ \langle c, a \rangle | \exists l (\langle c, l \rangle \in \text{borrower_details} \wedge \exists b (\langle l, b, a \rangle \in \text{loan} \wedge b = "PQR")) \}$

Output:

EMP_NAME
Amit

Query-4 Find the names of all employees who have a loan, an account, or both at the ABC branch.

Relational Calculus

$$\begin{aligned} & \{ < c > | \exists l (< c, l > \in \text{borrower_details} \\ & \quad \wedge \exists b (< l, b, a > \in \text{loan} \wedge b = "ABC")) \} \\ & \quad \vee \exists a (< c, a > \in \text{depositor_details} \\ & \quad \wedge \exists b, n (< a, b, c > \in \text{account_bank} \wedge b = "ABC")) \} \end{aligned}$$

Output:

EMP_NAME
Priya

Queries-5 Find the names of all employees who have an account at all the branches located in PQR.

NOTES

$$\begin{aligned} & \{ < c > | \exists l (< c, n > \in \text{emp_name} \\ & \quad \vee x, y, z (< x, y, z > \in \text{branch_details} \wedge y = "PQR") \Rightarrow \\ & \quad \exists a, b (< a, x, b > \in \text{account_bank} \wedge < c, a > \in \text{depositor_details})) \} \end{aligned}$$

Output:

EMP_NAME	LOAN_NO
Priya	L24
Ashish	L36
Amit	L20

Here we can express the expression as “The set of all (*emp_name*) tuples *c* such that, for all (*branch-name*, *branch-city*, *assets*) tuples, *x*, *y*, *z*, if the branch city is PQR, then the following is true”:

- There exists a tuple in the relation depositor_detail with employee *c* and account_no *a*.
- There exists a tuple in the relation account_bank with account_no *a* and branch_name

Some important points that should be kept in mind while using tuple and domain relational calculus are:

- We can write expressions which create an infinite relation, in case of tuple relational calculus.
- In domain relational calculus, suppose an expression:
 $\{ < l, b, a > | \neg (< l, b, a > \in \text{loan}) \}$

Is not correct, because it allows some values in the result which are not in the domain of the expression.

- In case of domain relational calculus, we must be aware about the form of formulae within “there exists” and “for all” clauses.

NOTES

- In case of tuple relational calculus, to range over a specific relation, we regulated any existentially quantified variable.
- In case of domain calculus, we did not do so in the domain calculus. Here, we add rules to the definition of safety which deals with cases.

For example: consider an expression:

$$\{ < x_1, x_2, \dots, x_n > \mid P(x_1, x_2, \dots, x_n) \}$$

is correct, if it satisfies below mentioned some conditions:

- For every “for all” sub formula of the form $\forall x (P_1(x))$, the subformula is true if and only if $P_1(x)$ is true for all values x from $\text{dom}(P_1)$.
- All values that appear in tuples of the expression are values from $\text{dom}(P)$.
- For every “there exists” subformula of the form $\exists x (P_1(x))$, the subformula is true if and only if there is a value x in $\text{dom}(P_1)$ such that $P_1(x)$ is true.

6.3 EXPRESSIVE POWER OF ALGEBRA AND CALCULUS

We have offered two formal query languages for the relational model. But there are some questions arise:

- Can every query that can be expressed in relational algebra also be expressed in relational calculus?
- Are they equivalent in power?
- Can each query that is stated in relational calculus can also be expressed in relational algebra?

The answer of the above queries is yes. But here, before discussing the above points, we should discuss some important problems related with calculus.

Consider the query $\{ T \mid \neg(T \in \text{Customers}) \}$. Here, this query is syntactically correct. It states that, for all tuples T such that T is not in Customers . In the framework of infinite domains such as the set of all integers, the set of such S tuples is obviously infinite. But here, this example demonstrates an unsafe query. So, it is necessary to control relational calculus to prohibit unsafe queries.

Now consider an example related to calculus queries which are constrained to be safe:

Consider with one instance per relation that appears in the query Q , a set S of relation instances. Assume $\text{Dom}(Q, S)$ be the set of all constants and these constants appear in the formulation of the query Q or in these relation instances S or itself. So here, only finite instances S allowed that is $\text{Dom}(Q, S)$ is finite.

At a minimum we want to ensure that for any given S , for a calculus formula Q to be considered safe, there is a set of answers for Q holds only values that are

NOTES

in $\text{Dom}(Q, s)$. But this type of constraint is clearly compulsory but it is not enough. We wish to compute the set of answers by only examining tuples that contain constants in $\text{Dom}(Q, I)$, not only do we want the set of answers to be collection of constants in $\text{Dom}(Q, S)$. This step points to an indirect point related with the use of quantifiers \forall and \exists . Suppose given a TRC formula of the form $\forall R(p(R))$, we want to find any values for variable R which make this formula false by checking only tuples that contain constants in $\text{Dom}(Q, S)$. Similarly, there is a given a TRC formula of the form $\exists R(p(R))$. Here, we want to find all values for variable R which make this formula true by checking only tuples that comprise constants in $\text{Dom}(Q, S)$.

So the benign TRC formula Q to be a formula like:

- If a tuple r (assigned to variable R) makes the formula true, for each sub expression of the form $\exists R(p(R))$ in Q , then r comprises only constants in $\text{Dom}(Q, S)$.
- Consider for the given S , the set of answers for Q comprises only values that are in $\text{Dom}(Q, S)$.
- If a tuple r (assigned to variable R) contains a constant that is not in $\text{Dom}(Q, S)$, for each sub expression of the form $\forall R(p(R))$ in Q , then r must make the formula true.

Here the point must be noted down that this description is not constructive, it means, it does not express about how to check if a query is safe.

By this definition, the query $Q = \{S \mid \neg(S \in \text{Customers})\}$ is insecure. $\text{Dom}(Q, S)$ is the set of all values which appear in (an instance I of) Customers .

So here, the easy-to-read power of relational algebra is frequently used as a metric in terms of how is powerful a relational database query language. Returning back to the question in the form of expressiveness, we can demonstrate that each query which can be stated using a safe relational calculus query. It can also be conveyed as a relational algebra query. If any query language which can precise all the queries that can be direct in relational algebra. So, it is said to be relationally complete. Therefore, we can say that, a practical query language is expected to be relationally complete. So we can conclude that commercial query languages classically support features which allow us to explain some queries and these can't be expressed in relational algebra.

Check Your Progress

1. What is relational calculus?
2. What are the two types of relational calculus?

NOTES

6.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Relational calculus is a non-procedural query language. It describes what to do but never explains how to do.
2. There are two types of relational calculus i.e.
 - (i) Tuple relational calculus
 - (ii) Domain relational calculus

6.5 SUMMARY

- Relational algebra query specifies how data can be retrieved from a relation but the relational calculus specifies what is to be retrieved from a relation.
- The relational calculus is not the same as that of integral and differential calculus in mathematics, but it's a form of predicate calculus.
- A predicate is defined as a truth-valued function with argument and for the arguments, when we replace with values then the function gives an expression which is known as proposition.

6.6 KEY WORDS

- **Relational Calculus:** It is a query language used by databases to retrieve the required data.
- **Domain Relational Calculus:** It is a type of Relational calculus that uses domain variables. A domain variable takes a value from the attributes domain rather than from an entire tuple.

6.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What do you understand by relational calculus?
2. Write the form of tuple and domain calculus.

Long Answer Questions

1. Explain the tuple and domain relational calculus with the help of an example.
2. Describe the expressive power of algebra and calculus.

6.8 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd).
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

NOTES

BLOCK - III
SQL QUERY

**UNIT 7 FORM OF BASIC SQL
QUERY**

Structure

- 7.0 Introduction
- 7.1 Objectives
- 7.2 Relational Database Query Language
 - 7.2.1 Forms of SQL
 - 7.2.2 Types of SQL Commands
 - 7.2.3 SQL Statements
 - 7.2.4 Data Definition Language (DDL)
 - 7.2.5 Data Manipulation Language (DML)
 - 7.2.6 Join in SQL
 - 7.2.7 Nested Query/Subquery in SQL
 - 7.2.8 Impact on SQL Constructs
- 7.3 Integrity Constraints in SQL Triggers and Active Databases
- 7.4 Introduction to Schema Refinement
- 7.5 Answers to Check Your Progress Questions
- 7.6 Summary
- 7.7 Key Words
- 7.8 Self Assessment Questions and Exercises
- 7.9 Further Readings

7.0 INTRODUCTION

SQL is a programming domain-specific language intended for the management of data maintained in a relational database management system or for the processing of streams in a relational data stream management system.

SQL is a language that facilitates interaction with relational database management systems. The prototype for SQL was originally developed by IBM based on E.F. Codd's paper 'A Relational Model of Data for Large Shared Data Banks'. In 1979, ORACLE, an SQL product, was released. Today, SQL has become a very important relational database management system.

In this unit, you will study about the basic form of Structured Query Language (SQL) query, examples of basic SQL queries, introduction to nested queries, correlated nested queries set, comparison operators, aggregative operators, NULL values, comparison using NULL values, logical connectivity's, AND, OR and NOT,

impact on SQL constructs, outer joins, disallowing NULL values, complex integrity constraints in SQL triggers and active database and schema refinement.

7.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand what Structured Query Language is and its advantages
- Describe SQL in ORACLE, forms and types of SQL commands
- Explain DDL and DML commands
- Explain SELECT, INSERT, DELETE and UPDATE commands
- Understand what is join and subquery
- Explain the integrity constraints in SQL triggers and active databases
- Discuss the schema refinement

NOTES

7.2 RELATIONAL DATABASE QUERY LANGUAGE

RDBMS is the basis for SQL. SQL is the language used to interact with relational database management system.

7.2.1 Forms of SQL

There are two forms of SQL:

- Interactive
- Embedded

Interactive SQL operates on a database to produce output for user demand. In **embedded SQL**, SQL commands can be put inside a program written in some other language (called host language) like C, C++, etc. Data is passed to a program environment through SQL. The combined source code is accepted by a special SQL precompiler and, along with other tools, it is converted into an executable program.

7.2.2 Types of SQL Commands

SQL commands are of varied types to suit different purposes.

The primary types are as follows:

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Query Language (DQL)
- Data Control Language (DCL)
- Transactional control language (TCL)

NOTES

- **Data Definition Language (DDL)**

Data Definition Language (DDL) is a part of SQL that allows a database user to create and restructure database objects, such as the creation or deletion of a table. Some of the most fundamental DDL commands include the following:

CREATE
ALTER
DROP

- **Data Manipulation Language (DML)**

Data Manipulation Language or DML is that part of SQL which is used to manipulate data within objects of a relational database. There are three basic DML commands:

INSERT
UPDATE
DELETE

- **Data Query Language (DQL)**

Though comprising only one command, Data Query Language (DQL) is the most concentrated focus of SQL for modern relational database users. The base command is as follows:

SELECT

This command, accompanied by many options and clauses, is used to compose queries against a relational database. Queries, from simple to complex, from vague to specific, can be easily created. A *query* is an inquiry to the database for information. A query is usually issued to the database through an application interface or via a command line prompt.

It is to be noted that, according to some authors, the SELECT command may be treated as DML as a query is nothing but a part of DML that is used for retrieval of data from a database.

- **Data Control Language (DCL)**

Data control commands in SQL allow us to control access to data within the database. These DCL commands are normally used to create objects related to user access and also control the distribution of privileges among users. Some data control commands are as follows:

GRANT
REVOKE

- **Transactional Control Languages (TCL)**

In addition to the previously introduced categories of commands, the following commands allow the user to manage database transactions:

COMMIT
ROLLBACK
SAVEPOINT
SET TRANSACTION

*Form of Basic SQL
Query*

NOTES

7.2.3 SQL Statements

Directions to Write SQL Statements

- SQL statements are insensitive to case.
- They can be of one or more lines.
- SQL keywords cannot be split across lines or abbreviated.
- They are generally entered in uppercase; all other words, such as column names and table are entered in lowercase. However, this is not a rule.
- SQL keywords are typically aligned in the first column.
- SQL statements are terminated with a semi-colon.

Some more information about SQL and ORACLE SQL in particular are as follows:

- A NAME for a view, table, synonym, index, column, or user variable MUST
- Start with a letter.
 - Consist of only the characters A–Z, 0–9, _, \$, and #.
 - Not be the same as an ORACLE reserved word.
 - Be 1 to 30 characters long (Database names need not exceed 8 characters).
 - Not consist of a quotation mark.

Notations used in the Syntax of SQL Commands

In this unit, lowercase is used to identify names or conditions entered by the user, and uppercase is used to identify SQL keywords.

- User-given data or object name or expression or condition is enclosed within a “<“ and ’>”.
- Square brackets (“[]”) identify optional items. Do not include the brackets when we enter a query.
- A vertical bar (“|”) indicates a choice and underlining indicates a default.
- Ellipses (“...”) are used to specify items that may repeat.

7.2.4 Data Definition Language (DDL)

DDL is the subset of SQL commands used to modify create or remove Oracle database objects, including tables. These commands may also be used to update information in the Data Dictionary accordingly.

NOTES

SQL has three main commands for data definition:

- CREATE for creation of database objects.
- ALTER for changing database objects.
- DROP for deleting database objects.

As creation, alteration and deletion of databases are database administration tasks, you will learn only tables in this section. Before discussing the DDL commands, you will learn about data types supported by Oracle.

Different Data Types in Oracle

There are several data types supported by Oracle, which are as follows:

Built-In Data type	Description
CHAR(size)	Fixed-length character data of length 'size' bytes. Default and minimum size is 1 byte and the maximum size is 2000 bytes..
VARCHAR2(size)	Variable-length character string that has the maximum length 'size' bytes. The maximum size is 4000 and minimum is 1. Here, 'size' for VARCHAR2 must be specified. Note: Since the VARCHAR2 data type is the successor of VARCHAR, it is recommended to use VARCHAR2 as a variable-sized array of characters.
NUMBER(p, s)	Specifies integer as well as real numbers, having precision p and scale s. The precision p has a range from 1 to 38. The scale s has a range from - 84 to 127. The default value for p in a number data type is 38. The default scale s is 0 for no decimal places. When s is positive, the number of decimal places in scale is increased and when it is negative, the actual data is rounded off to the specified number of places at the left of the decimal point. Example: 7456123.89 → Number (7, - 2) → 7456100 A NUMBER data type with no parameters is set to its maximum size.
LONG	This denotes character data of variable length up to 2 GB, or 2^{31} - 1 bytes. LONG columns in SQL statements can be referenced in the following places: <ul style="list-style-type: none"> • SELECT clause • SET clauses in UPDATE statements • VALUES clauses in INSERT statements There are some restrictions to the use of LONG values: <ul style="list-style-type: none"> • A table should not have more than one LONG column. • LONG columns cannot be in integrity constraints (exception: NULL and NOT NULL constraints). • LONG columns cannot be indexed. • A procedure or stored function should not accept a LONG argument. Also, LONG columns cannot be in certain parts of SQL statements, such as: <ul style="list-style-type: none"> • WHERE, GROUP BY, ORDER BY, or CONNECT BY clauses or with the DISTINCT operator in SELECT statements. • SQL functions such as SUBSTR or INSTR. • expressions or conditions. • select lists of queries containing GROUP BY clauses. • select lists of subqueries or queries combined by set operators. • select lists of CREATE TABLE AS SELECT statements.
DATE	Valid date in ORACLE ranges from January 1, 4712 BC to December 31, 9999 AD. The default input date format is DD-MON-YY HH:MM:SS where DD is the Day of the month, MON is a three-letter abbreviation for the Month, YY is the two-digit representation of the Year. HH, MM and SS are two-digit representations of Hour, Minute and Seconds.
RAW(size)	Raw represents binary data of length 'size' bytes. The size for a RAW value must be specified and the maximum size is 2000 bytes.
LONG RAW	Long raw represents binary data of variable length up to 2 gigabytes.
ROWID	Hexadecimal string represents the unique address of a row in a table. This data type is mainly used for the values returned by the ROWID pseudo column. Pseudo column is like a column in a table, but is not actually stored in it. On pseudo column one can only select; delete, insert and update are not allowed.
CLOB	A character large object contains 1-byte characters. Both fixed-width and variable-width character sets are supported and both use the CHAR database character set. The maximum size is 4 GB.
BLOB	This represents a binary large object having a maximum size of 4 GB.
BFILE	This contains a locator to a large binary file that is stored outside the database. This enables byte stream I/O access to external LOBs, which reside on the database server. The maximum size is 4 GB. The size is also limited by the operating system.

Instead of using the ANSI standard data types, Oracle-defined data types can be used. The following table gives the Oracle data type alternative for the ANSI standard data types:

ANSI Standard	Oracle Data Type
CHARACTER and CHAR	CHAR
CHARACTER VARYING and CHAR VARYING	VARCHAR2
NUMERIC, DECIMAL, DEC, INTEGER, INT and SMALLINT	NUMBER
FLOAT, REAL and DOUBLE PRECISION	FLOAT

NOTES

CREATE TABLE Command

A table denotes the basic structure that holds user data. The CREATE TABLE command creates a table, defines its columns, integrity constraints and storage allocation.

The CREATE TABLE system privilege must be possessed to create a table in the user's own schema. Likewise, to make a table in other user's schema, the user must have the CREATE ANY TABLE system privilege. Also, the owner of the schema to contain the table should have either space quota on the tablespace to contain the table or the UNLIMITED TABLESPACE system privilege.

- To make a new table, the following syntax is used:

```
CREATE TABLE <table name>
  (<column name> <data type> [(<size>) ] <column
   constraint>,
  .....
  <Table constraints>);
```

where:

- o <table name> is the name of the table; it must be an alphanumeric identifier.
- o <column name> is the name of an attribute; it must be an alphanumeric identifier.
- o <data type> is the type of the table.

The syntax is made simple and for simplicity it does not have the optional clauses, such as PCTFREE, PCTINCREASE, STORAGE, etc., which gives instruction to Oracle how to allocate space dynamically to the table data.

Example:

```
CREATE TABLE emp
  (ecode VARCHAR2(5),
  ename VARCHAR2(30),
```

NOTES

```
dno VARCHAR2 (10),  
salary NUMBER(7,2));
```

- **Note:** Each table must have at least one column, and column names within one table must be unique.

A column can be given a default value through the *DEFAULT option*. It assigns a value to be given to the column, if a later `INSERT` statement omits a value for the column.

```
CREATE TABLE emp  
(ecode VARCHAR2(5),  
ename VARCHAR2(30),  
dno VARCHAR2(10),  
salary NUMBER(7,2),  
da NUMBER(7,2) DEFAULT 0);
```

- The default may be literals, an expression having the same data type as the column, but not the name of another column. Functions, such as `SYSDATE` and `USER` are valid.

```
CREATE TABLE emp  
(ecode VARCHAR2(5),  
ename VARCHAR2(30),  
dt_jn DATE DEFAULT SYSDATE,  
dno VARCHAR2(10),  
salary NUMBER(7,2));
```

A `CONSTRAINT clause` is included in a `CREATE TABLE` statement for defining an *integrity constraint*.

Integrity constraints in Oracle are of the following types:

- **NOT NULL:** Specifies a column that cannot contain null values (no value is assigned)
- **UNIQUE:** Indicates the values of a column (or columns) that must be unique.
- **PRIMARY KEY:** A set of one or more columns, which acts as the table's primary key.
- **FOREIGN KEY:** A set of one or more columns, which designates the foreign key in a referential integrity constraint.
- **CHECK:** Specifies a condition that each row of the table must satisfy.

ORACLE allows integrity constraints to be defined for tables and columns for the following purposes:

- Enforcing rules at table level whenever a row is inserted, updated or deleted from that table, where the constraint must be satisfied for the operation to succeed

- Preventing the deletion of a table if there are dependencies from other tables
- There are two basic types of constraints—column constraints and table constraints.

Column constraints apply only to individual columns and are provided with the column definition.

Table constraints apply to groups of one or more columns and are defined separately from the definitions of the columns in the table.

Column constraints are of the following form:

```
[CONSTRAINT <constraint_name>]
[NOT] NULL |
CHECK (expression) |
UNIQUE |
PRIMARY KEY |
REFERENCES <table-name> [<column-name>] |
[ON DELETE CASCADE]
```

Table constraints are of the following form:

```
[CONSTRAINT <constraint-name>]
UNIQUE (<column-name>, ..., <column-name>) |
PRIMARY KEY(<column-name>, ..., <column-name>) |
FOREIGN KEY (<column-name>, ..., <column-name>)
REFERENCES table-name [<column-name>, ...,
<column-name>]
[ON DELETE CASCADE]
```

[CONSTRAINT <constraint-name>] clause of the CREATE TABLE command is optional. CONSTRAINT denotes the integrity constraint by the name that follows (Pk_snum, Ck_city) it. The definition of the integrity constraint in the data dictionary is stored in this name by ORACLE. Constraints were named with the following prefixes:

- Primary key constraints: pk_
- Foreign key constraints: fk_
- Check constraints: ck_

Naming constraints in this fashion is simply a convenience. Any name may be given to a constraint.

If a name for the constraint is not specified, then Oracle generates a name. Most Oracle's generated constraint names are of the form SYS_C#####; e.g., SYS_C000145.

NOTES

NOTES

NOT NULL Constraints

The requirement of the NOT NULL constraint is that the column contains a value when it is initially inserted into the table or whenever the column is updated.

Specifying PRIMARY KEY Constraint

The most widely used means of enforcing integrity are Column constraints. Of these, the most significant one is the PRIMARY KEY. It ensures that each row in the table is unique. When a column is affirmed as the PRIMARY KEY, an index on this column is automatically created and assigned a unique name by Oracle. The additional constraints UNIQUE and NOT NULL are implied by the PRIMARY KEY constraint.

The following example defines a primary key using column constraint:

```
CREATE TABLE emp
  (ecode    VARCHAR2(5) PRIMARY KEY,
   ename   VARCHAR2(30) NOT NULL,
   dno     VARCHAR2(10),
   salary   NUMBER(7, 2));
```

Generally, composite primary key is defined using table constraint. The following example defines a composite primary key using table constraint:

```
CREATE TABLE order
  (itemcode   VARCHAR2(10),
   vendorcode  VARCHAR2(10),
   qty        VARCHAR2(10),
   PRIMARY KEY (itemcode, vendorcode));
```

We can give a name to the constraint using the CONSTRAINT clause. The above CREATE TABLE commands can be written as follows:

```
CREATE TABLE emp
  (ecode    VARCHAR2(5) CONSTRAINT pk_emp PRIMARY KEY,
   ename   VARCHAR2(30) NOT NULL,
   dno     VARCHAR2(10),
   salary   NUMBER(7, 2));
CREATE TABLE order
  (itemcode   VARCHAR2(10),
   vendorcode  VARCHAR2(10),
   qty        VARCHAR2(10),
   CONSTRAINT pk_order PRIMARY KEY (itemcode,
                                     vendorcode));
```

Specifying UNIQUE Constraint

The *UNIQUE constraint* recognizes a column or grouping of columns as a unique key. The unique key, which is made up of a single column, contains NULLS. In

case of a composite unique key, any row that contains NULLS in all key columns automatically satisfies the constraint.

Form of Basic SQL Query

Column constraint syntax:

```
[CONSTRAINT <constraint_name>] UNIQUE
```

Table constraint syntax:

```
[CONSTRAINT <constraint_name>] UNIQUE  
(<column_name> [, <column_name>] ....)
```

To create a table, Sales, to keep track of the total amount of sales per day per salesperson, ensuring that each day has no more than one row for a given salesperson, enter:

```
CREATE TABLE Sales  
    (snum NUMBER (5) NOT NULL,  
     odate DATE NOT NULL,  
     totamt NUMBER (7, 2),  
     CONSTRAINT Unq_snum_odate UNIQUE (snum, odate));
```

UNIQUE is a constraint that enforces separate column values on a table column and contains a NULL record also. Similar to UNIQUE is the PRIMARY KEY constraint; however, it cannot have a Null value for the column. There has to be only one primary key in a table. Moreover, a PRIMARY KEY is the parent column for a Parent–Child relationship (Foreign Key).

NOTES

Specifying CHECK Constraint

A condition that each row in the table must evaluate for either true or unknown (due to a NULL) is defined by the *CHECK constraint*. This condition can only refer to columns in the table to which it belongs. The condition cannot include the following:

- Subqueries
- Functions like SYSDATE, USER
- Pseudocolumns

Syntax: [CONSTRAINT <constraint_name>] CHECK (<condition>)

CHECK is useful as a table constraint, when more than one field of a row is involved in a condition. To ensure that city within Kolkata, Chennai, Mumbai or Delhi is permitted for employees, while creating emp table, enter:

```
CREATE TABLE emp  
    (ecodeVARCHAR2 (5) PRIMARY KEY,  
     ename VARCHAR2 (30) CONSTRAINT Nn_enm NOT NULL,  
     city  VARCHAR2 (10) CONSTRAINT Check_city  
           CHECK (city IN 'KOLKATA', 'CHENNAI', 'MUMBAI', 'DELHI')),  
     salaryNUMBER (7, 2)
```

```
CONSTRAINT Chk_sal CHECK(sal BETWEEN 4000 AND 50000)
);
```

Another Example:

NOTES

```
CREATE TABLE prod_tbl
(prod_no NUMBER(3) PRIMARY KEY,
description VARCHAR2(50),
cost NUMBER(8,2) CHECK (cost > .50));
```

Specifying REFERENTIAL INTEGRITY Constraint

To establish a relationship between the foreign key and a specified primary or unique key called the referenced key, a *referential integrity constraint* is assigned a column or combination of columns as *FOREIGN KEY*. In this relationship, the table containing the foreign key is called the **child table** and the table containing the referenced key is called the **parent table**. The referenced unique or primary key constraint on the parent table should already be defined before defining a referential integrity constraint in the child table.

The REFERENCES clause refers to any unique or primary key in the foreign table (foreign key).

Column constraint syntax:

```
[CONSTRAINT <constraint_name>]      REFERENCES <table_name>
[(<column>)]
[ON DELETE CASCADE]
Table constraint syntax:
[CONSTRAINT <constraint_name>]      FOREIGN KEY (<column [,,
column] ...>)
REFERENCES <table_name> [(<column [,, column] .....>)]
[ON DELETE CASCADE]
```

Through ON DELETE CASCADE the referenced key values in the parent table is deleted, which have dependent rows in the child table. In this case, ORACLE automatically deletes dependent rows from the child table to maintain referential integrity. If this option is omitted, ORACLE does not allow deletion of referenced key values in the parent table, which have dependent rows in the child table.

There is a DEPT table, which is created using the following CREATE TABLE command:

```
CREATE TABLE dept
(deptno VARCHAR2(3) PRIMARY KEY,
dname VARCHAR2(20) NOT NULL);
```

To create the EMP table, with a referential integrity constraint defined as a table constraint, enter:

```
CREATE TABLE emp
```

```

(ecode VARCHAR2 (5) PRIMARY KEY,
ename  VARCHAR2 (30) NOT NULL,
city   VARCHAR2 (10),
dno VARCHAR2 (3),
salary NUMBER (7, 2),
FOREIGN KEY (dno) REFERENCES dept (deptno)
);

```

*Form of Basic SQL
Query*

NOTES

Here, the ON DELETE CASCADE option is omitted by the foreign key definitions. So, ORACLE does not allow the deletion of a department if any employee works in that department.

The following statement creates the Student table, using the column constraint syntax to define a foreign key on the cnum column that references the primary key of the Course table (ccode) with the constraint name fk_student:

```

CREATE TABLE student
(roll  NUMBER (5)      PRIMARY KEY,
sname  VARCHAR2 (10)    NOT NULL,
section  VARCHAR2 (1),
cnum  NUMBER (5) CONSTRAINT fk_student REFERENCES
course(ccode );

```

ALTER TABLE Command

The definition of a table is altered by this command in one of the following ways:

- By adding a column.
- By adding an integrity constraint.
- By redefining a column (data type, size, default value).
- By modifying storage characteristics or other parameters.
- By enabling, disabling or dropping an integrity constraint.
- By deleting or renaming a column.

On the successful execution of the ALTER TABLE command, the system will give a message 'Table altered'.

Adding Column(s) to a Table

The ADD clause of the ALTER TABLE command defines additional columns and integrity constraints after the creation of a table.

To add a column to an existing table, the ALTER TABLE syntax is as follows:

```

ALTER TABLE <table_name> ADD (<column_name>
column_definition);
column_definition includes constraint specification also.

```

NOTES

For Example:

```
ALTER TABLE emp ADD (address VARCHAR2 (50));
```

The above DDL statement will add a column called *address* to the *emp* table.

To add multiple columns to an existing table, the ALTER TABLE syntax is as follows:

```
ALTER TABLE <table_name>
ADD (<column_name> column-definition,.....);
```

For Example:

```
ALTER TABLE emp
ADD (address VARCHAR2 (50), tel_no NUMBER(15));
```

This adds two columns (address and tel_no) to the emp table.

Methods for Adding a Column to a Table

- A column can be added at any time if NOT NULL is not specified. A column defined as NOT NULL cannot be added. If we try to add it, the column will not have anything in it. Every row in the table will have a new empty column defined as NOT NULL. So Oracle will generate an error message.
- A NOT NULL column may be added in the following three steps:
 - o Add a column without NOT NULL specified.
 - o Fill every row in that column with data.
 - o Modify the column to be NOT NULL.

Modifying Columns in a Table

The MODIFY clause can be used to change some of the parts of a column definition as given here:

- Data type
- Size
- Default value
- NOT NULL column constraint

The MODIFY clause specifies the modified part of the definition and the column name but not the entire definition.

To modify a column in an existing table, the ALTER TABLE syntax is as follows:

```
ALTER TABLE table_name MODIFY column_name column_definition;
```

For Example:

```
ALTER TABLE emp MODIFY ename VARCHAR2 (35) NOT NULL;
```

This will modify the column called *ename* to be a data type of varchar2 (35) and force the column to not allow null values.

To modify multiple columns in an existing table, the ALTER TABLE syntax is as follows:

```
ALTER TABLE table_name
MODIFY (<column_name> column_definition,
.....);
```

For example:

```
ALTER TABLE emp
MODIFY (ename VARCHAR2(35) NOT NULL, Cityvarchar2(50));
```

This will modify both the *ename* and *city* columns.

Rules for modifying a column

- A column of character type can be increased any time.
- The number of digits in a NUMBER column can be increased any time.
- The number of decimal places in a NUMBER column can be increased or decreased any time.

A column's data type may be changed or its size decreased, only if the column contains NULL in all rows. However, the size of a character column or the precision of a numeric column can always increase.

Any alteration to a column's default value only affects rows that are subsequently inserted into the table and not the values previously inserted.

The only integrity constraint that can be added to an existing column using the MODIFY clause with the column constraint syntax is a NOT NULL constraint.

The following ALTER TABLE command add NOT NULL constraint against dname column:

```
ALTER TABLE dept MODIFY (dname VARCHAR2(14)
CONSTRAINT dept_dname_nn NOT NULL);
```

However, the other integrity constraints like PRIMARY KEY, UNIQUE, CHECK, REFERENTIAL and INTEGRITY is defined on the existing columns using the ADD clause with the table constraint syntax.

A NOT NULL constraint, if it contains no NULLS, is only defined on an existing column.

Dropping Columns in a Table

With the introduction of Oracle 8i, it is possible to drop a column from a table. Prior to this edition, the only way to do this is to drop the entire table and rebuild it. Using Oracle 8i, a column can be deleted in the following two ways:

- Logical Delete: marking a column as unused.
- Physical Delete: delete it completely.

NOTES

NOTES

Logical Delete

The reason for logical delete is that the procedure of physically removing a column from large tables is very resource and time consuming. The syntax for logical delete of column(s) is as follows:

```
ALTER TABLE <table_name> SET UNUSED (<column_name>);  
ALTER TABLE <table_name> SET UNUSED (<column_name1>, <column_name2>, .....);
```

The columns will no longer be visible to the user, once this is done.

```
An example include ALTER TABLE emp SET UNUSED (salary);
```

The columns can be deleted physically later by the following statement:

```
ALTER TABLE <table_name> DROP UNUSED COLUMNS;  
e.g. ALTER TABLE emp DROP UNUSED COLUMNS;
```

The DBA_UNUSED_COL_TABS view can be used to view the number of unused columns per table.

Physical Delete

To physically drop a column from an existing table, the ALTER TABLE syntax is as follows:

```
ALTER TABLE <table_name>  
DROP COLUMN <column_name> [CASCADE CONSTRAINTS];
```

Suppose the dropped columns are part of unique constraint or the primary keys, then it is necessary to use the optional CASCADE CONSTRAINTS clause as a part of the ALTER TABLE command.

For example, the following command will drop the column called *city* from the table called *emp*:

```
ALTER TABLE emp DROP COLUMN city;
```

Multiple columns can be dropped in a single command. The syntax is as follows:

```
ALTER TABLE table_name DROP (column_name1, column_name2, .....);
```

For example:

```
ALTER TABLE emp DROP (city, comm);
```

Notice that when dropping multiple columns, the column keyword of the ALTER command should not be used as it causes a syntax error.

```
ALTER TABLE prod_on_ord  
DROP COLUMN order_no CASCADE CONSTRAINTS;
```

Notes:

- A column, which is a primary key, cannot be dropped.
- All columns of a table cannot be dropped.

Rename column(s) in a table (only available in Oracle 9i Release 2 onwards)

Starting in Oracle 9i Release 2, we can now rename a column.

To rename a column in an existing table, the ALTER TABLE syntax is as follows:

```
ALTER TABLE table_name
    RENAME COLUMN <old_column_name> TO <new_column_name>;
```

For Example:

```
ALTER TABLE emp RENAME COLUMN ename TO empname;
```

The column is thus renamed and called *ename* to *empname*.

Adding Constraints to a Table

The ADD clause is used to add constraint to a table.

```
ALTER TABLE emp ADD ( CONSTRAINT emp_pk PRIMARY KEY (ecode) );
```

Renaming Integrity Constraints

Apart from allowing to rename tables and columns Oracle9i Release 2 permits the renaming of constraints on tables.

The syntax for renaming constraint is as follows:

```
ALTER TABLE <table_name>
    RENAME CONSTRAINT <old_constraint_name> TO
        <new_constraint_name>;
```

In the following example, we rename the primary key constraint of the EMP table:

```
SELECT constraint_name
    FROM user_constraints
    WHERE table_name = 'EMP'
        AND constraint_type = 'P' ;
```

Output:

```
CONSTRAINT_NAME
PK_emp
```

Now we enter the following ALTER TABLE command:

```
ALTER TABLE test RENAME CONSTRAINT PK_emp TO emp_PK;
```

Output:

Table altered.

```
SELECT constraint_name
    FROM user_constraints
    WHERE table_name = 'EMP'
        AND constraint_type = 'P' ;
```

Output:

```
CONSTRAINT_NAME
Emp_PK
```

NOTES

NOTES

Dropping Integrity Constraints

The DROP clause is used to remove an integrity constraint from a table.

The syntax for the DROP clause is as follows:

```
DROP { PRIMARY KEY | UNIQUE (<column_name>[,  
column_name].....) }  
| CONSTRAINT<constraint_name> } [CASCADE]
```

For example, to drop the check_city constraint, the SQL statement is as follows:

```
ALTER TABLE emp DROP CONSTRAINT Check_city;
```

Dropping the primary key constraint of dept table

```
ALTER TABLE dept DROP PRIMARY KEY CASCADE;
```

The CASCADE option of the DROP clause causes any dependent constraints also to be dropped.

To Enable and Disable Existing Integrity Constraints

The ENABLE/DISABLE clause of the ALTER TABLE command allows constraints to be enabled or disabled without dropping or recreating them.

The syntax for the ENABLE/DISABLE clause is as follows:

```
[ENABLE | DISABLE } {UNIQUE (<column_name>[, column_name] .....) }  
| PRIMARY KEY  
| CONSTRAINT<constraint_name> } [CASCADE]
```

For example:

```
ALTER TABLE emp DISABLE PRIMARY KEY CASCADE;
```

ORACLE enforces a constraint, which is enabled by enforcing it to all data in the table. An enabled constraint has to be satisfied by all table data.

ORACLE ‘does not enforce it, if an integrity constraint is disabled. However, along with enabled integrity constraints it appears in the data dictionary.

The alter table command is used to **disable a constraint**. Use the alter table command again to **enable a disabled constraint**. The following example disables and then re-enables the salary check condition:

```
ALTER TABLE emp DISABLE CONSTRAINT CHK_SALARY;  
ALTER TABLE emp ENABLE CONSTRAINT CHK_SALARY;
```

DROP TABLE Command

This command removes a table including all its data from the database.

Syntax

```
DROP TABLE <table_name> [CASCADE CONSTRAINTS ]
```

CASCADE CONSTRAINTS All referential integrity constraints are dropped that refer to unique and primary keys in the dropped table.

The following statement drops the emp table:

```
DROP TABLE emp;
```

When a table is dropped, the following operations are performed automatically:

- All rows from the table are deleted.
- All the table's indexes regardless of who created or owns them are dropped.
- All data blocks allocated to the table and its indices to the tablespaces containing the table and indices are returned.
- Suppose the table is a base table for views or if it is referenced in stored procedures, functions or packages, these objects are invalidated but not dropped.

NOTES

Renaming a Table

Renaming a table can be done as:

```
RENAME <oldtablename> TO <newtablename>;
```

For example:

```
RENAME emp to empl;
```

Obtaining Table Information

To learn the structure of a table, use the DESC command.

For example, the current table columns and type definitions are displayed by the following command:

```
DESC emp;
```

Output:

```
Name Null? Type
E CODE NOT NULL VARCHAR2 (5)
ENAME VARCHAR2 (30)
SALARY NUMBER (7, 2)
DNO VARCHAR2 (5)
DESG VARCHAR2 (30)
DT_JN DATE
```

To obtain all tables owned by a user, use the following command:

```
SELECT * FROM TAB;
Or SELECT * FROM cat;
```

Output:

```
TNAME TABTYPE CLUSTERID
ASSIGNS TABLE
DEPT TABLE
EMP TABLE
```

NOTES

```
EMP_VIEWVIEW
PROJECT    TABLE
SAL_VIEWVIEW
```

6 rows selected.

7.2.5 Data Manipulation Language (DML)

DML is used to modify, add, query or delete data.

Inserting Rows – INSERT Command

The INSERT command is employed to add rows to a table. It is to be noted that to insert rows into a table, the table has to be in the user's own schema or the user must have the INSERT privilege on the table. Users can also insert rows into any table through the INSERT ANY TABLE system privilege.

The two general formats for the INSERT command are as follows:

- `INSERT INTO <table_name> VALUES (<value>,);`
- `INSERT INTO <table_name> (<column_name>,)
VALUES (<value>,);`

In the first format, the columns that will be given values for are not listed. The values must exactly match default and the type order of all the columns in the table. This can be found out with the DESC command. If the column list is omitted, the values clause has to specify values for all columns in the table. Omitted columns will be set to default values, which will be either NULL or an explicitly defined default. If a constraint prevents a NULL from being accepted in a given column, that column must be provided with a value.

For example, to insert a row into the emp table, enter:

```
INSERT INTO emp
VALUES ('E01', 'KOUSHIK GHOSH', 5000, 'D01', 'PROGRAMMER',
'10-MAR-93');
```

Oracle will notify the user with the following message for successful insertion:

1 row inserted

It is to be noted that CHARACTER and DATE values should be within in single quotes. If we wish to omit a value (and it is valid to do so, i.e., NULL-able column), we can use the keyword NULL, as in the following example:

```
INSERT INTO emp
VALUES ('E01', 'KOUSHIK GHOSH', NULL, 'D01', 'PROGRAMMER',
NULL);
```

In order to insert the same row omitting the designation, salary and dt_jn column the columns must be specified:

```
INSERT INTO emp (e_code, ename, dno)
VALUES ('E01', 'KOUSHIK GHOSH', 'D01');
```

NOTES

Here, this `INSERT` statement has only the `ecode`, `ename` and `dno` columns. The order in which the values appear corresponds to the statement's column list, not the table as listed by the `DESCRIBE` command.

To interactively prompt users to enter values for the fields at SQL*Plus terminal, use the following syntax:

```
INSERT INTO employee VALUES ('&ecode', '&ename', &salary,  
'&dno', '&desg', '&dt_jn');
```

The SQL*Plus terminal then will prompt users to enter values for the three variables `ecode`, `ename` and `salary`.

It is possible to insert multiple rows into a table by using the `INSERT INTO SELECT` statement. The syntax for the `INSERT INTO SELECT` statement is as follows:

```
INSERT INTO TABLE SELECT statement;
```

To copy all rows from the `emp` table with the `desg = 'PROGRAMMER'` into the `PROGRAMMER` table, enter:

```
INSERT INTO programmer  
SELECT * FROM emp WHERE desg = 'PROGRAMMER' ;
```

The chosen list of the subquery must have equal columns as the column list of the `INSERT` statement.

Possible Caveats with `INSERT`

When using `INSERT` the following integrity errors can occur:

- Attempting to insert `NULL`, values into a column with a `NOT NULL` constraint.
- Trying to insert non-allowed values into a column with a `CHECK` constraint.
- Attempting to insert a record with a foreign key value that has no corresponding primary key record in another table.

Updating Rows—`UPDATE` Command

The update statement facilitates the change of a column value in a row. The `WHERE` clause can contain any condition allowed in a `SELECT` statement including subqueries.

The syntax for the `UPDATE` command is as follows:

```
UPDATE <table_name>  
SET <column_name1>=<value1>, ... , <column_nameK>=<valueK>  
WHERE <condition>;
```

The `SET` clause determines which values are updated and what net values are stored in them. The `SET` clause accepts any number of column assignments, separated by commas. The `WHERE` clause is optional and can be used to select which row or rows are to be updated. The `WHERE` clause is followed by a logical

NOTES

expression and every record for which this expression is true is updated.
For example, to increase the salaries of all employees by 5 per cent, enter:

```
UPDATE emp SET salary = salary * 1.05;
```

The system will notify the user with the following message:

n rows updated:

where n is the number of rows affected.

To perform the same change on all employees of department 'D02', enter:

```
UPDATE emp SET salary = salary * 1.5 WHERE dno = 'D02' ;
```

To update the record of employee 'E01', enter:

```
UPDATE emp  
SET ename = 'JAYANTA GANGULY', city = 'KOLKATA',  
salary = salary * 2  
WHERE ecode = 'E04' ;
```

The above command updates ename, city and column.

Possible Caveats with UPDATE

When using UPDATE, the following integrity errors can occur:

- Attempting to update to NULL values a column with a NOT NULL constraint.
- Trying to update to non-allowed values a column with a CHECK constraint.
- Attempting to update a record to a foreign key value that has no corresponding primary key record in another table.
- A flawed or vacuous WHERE clause causing update of all records.

Deleting Records—DELETE Command

The DELETE command allows the removal of one or more rows from a table. The table has to be in user's own schema or the user must have the DELETE privilege on the table to delete rows from a table. The DELETE ANY TABLE system privilege gives freedom to a user to delete rows from any table.

The syntax for the DELETE command is as follows:

```
DELETE FROM tablename [WHERE condition]
```

Like the UPDATE command, omitting the WHERE clause means the command will be performed on all records. To remove all the rows of emp, enter:

```
DELETE FROM emp ;
```

The system will notify the user with the following message:

n rows deleted

where n is the number of rows deleted.

The following statement deletes from the emp table all employees located in Chennai :

```
DELETE FROM emp WHERE city = 'Chennai' ;
```

A condition can reference a table and contain a subquery. To the predicate of a DELETE command subqueries are also used. Use of subquery will be covered in the section on subquery.

Possible Caveats with **DELETE**

When using **DELETE**, the following integrity errors can occur:

- Attempting to delete a primary key record with foreign key records of another table. This only works if `ON DELETE CASCADE` constraint has been set.
- A flawed or vacuous WHERE clause causing the deletions of all records.

TRUNCATE Command

This command also allows the removal of all rows from a table, but it flushes a table more efficiently since no rollback information is retained.

For example, to delete all data and index (if any) rows of the emp table and return the freed space to the tablespace containing emp, enter:

```
TRUNCATE TABLE emp;
```

Syntax

TRUNCATE TABLE	<code><table_name> [{DROP REUSE } STORAGE]</code>
DROP STORAGE	It performs the deallocation of space from the deleted rows from the table.
REUSE STORAGE	It keeps the space from the deleted rows allocated to the table. This space can be used later only by new data in the table resulting from <code>INSERTS</code> or <code>UPDATES</code> .

Possible Caveats with **TRUNCATE**

When using **TRUNCATE**, the following integrity errors can occur:

- An implicit `COMMIT` will occur. Previous DML statements executed during the same session will be committed.

Difference between **DELETE** and **TRUNCATE** command

At the basic plane, **delete** scans the table and omits any rows that is similar to the given criteria in the (optional) where clause. It creates rollback information so that if needed the the deletions could be reverted. And from the indexes, the index entries for the deleted rows are changed. The `COMMIT` data control command must be given to permanently delete them.

Extents are not deallocated while deleting rows from a table,; the result is that the extents in the table remain even after the deletion. At the same time, the high water mark is not moved down, thus it also remains at its place as before the deletion.

NOTES

NOTES

At the same time, **Truncate**, just moves the high-watermark on the table to the beginning. This is done very quickly Truncate and it need not be committed. Once one table is truncated, there is no going back. Thus, indexes are also truncated as there is no facility to specify which row to ‘delete’. Also in the case with the where clause on the delete command.

All the extents of a table are deallocated, when it is truncated, leaving only the extents identified when the table was created in the first place. Thus, if the table was in the first place made with MINEXTENTS 3, 3 extents will remain when the table is truncated.

The extents are **not** deallocated if the REUSE STORAGE clause is specified. It saves time in the recursive SQL department if one intends to reload the table with data from an export, and can lessen the time it takes to do the import as there is no need to vigorously allocate any new extents.

Querying Data—SELECT Command

The most important statement in SQL is the SELECT statement. The following table used for subsequent discussion:

EMP		DEPT	
ecode	VARCHAR2(5)	dno	VARCHAR2(5)
ename	VARCHAR2(30)	dname	VARCHAR2(30)
salary	NUMBER(7,2)	city	VARCHAR2(25)
dno	VARCHAR2(5)		
dt_jn	DATE		
desg	VARCHAR2(30)		

Instance of EMP Table:

E CODE	E NAME	SALARY	D NO	D E S G	D T _ J N
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93
E02	JAYANTA DUTTA	3500	D01	PROGRAMMER	15-JAN-94
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E07	RANI BOSE	3000	D01	PROJECT ASSISTANT	17-JUN-97
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

Instance of DEPT table:

DNO	DNAME	CITY
D01	PROJECT	KOLKATA
D02	RESEARCH	CHENNAI
D03	PERSONNEL	KOLKATA

NOTES

The general syntax of this command is as follows:

```
SELECT [ALL | DISTINCT] <column_name1> [,<column_name2>]
FROM table_name1 [,table_name2]
[WHERE <condition>] [ AND|OR <condition>.....]
[GROUP BY <column-list>]
[HAVING <condition>]
[ORDER BY <column-list> [ASC | DESC] ];
```

where

SELECT	SELECT clause lists the columns to retrieve data from.
FROM	FROM clause lists the tables the columns are located in.
WHERE	WHERE clause specifies criteria return values must match.
GROUP BY	GROUP BY clause specifies groups for summary results.
HAVING	HAVING clause specifies filter conditions for summary results.
ORDER BY	ORDER BY clause specifies sort order.

The sections between the brackets [] are optional.

The following is the basic format of the SELECT statement:

```
SELECT <select_list>
FROM <table_name>
[WHERE <condition>]
```

The SELECT statement returns one or more rows, called the *result set*. If a WHERE clause is not specified, it returns one row for every row in the table. The select_list specifies the columns that are returned in the result set. The simplest form of the column list is an asterisk, which represents all the columns and this statement, therefore, returns all the columns and rows of the table.

Query: Display all rows of emp table

```
SELECT * FROM emp;
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93
E02	JAYANTA DUTTA	3500	D01	PROGRAMMER	15-JAN-94
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E07	RANI BOSE	3000	D01	PROJECT ASSISTANT	17-JUN-97
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

9 rows selected.

Query: List all employees' code, name and salary in the EMP table

```
SELECT ecode, ename, salary FROM emp;
```

Output:

NOTES

ECODE	ENAME	SALARY
E01	KOUSHIK GHOSH	5000
E02	JAYANTA DUTTA	3500
E03	HARI NANDAN TUNGA	4000
E04	JAYANTAGANGULY	6000
E05	RAJIB HALDER	4000
E06	JISHNU BANERJEE	6500
E07	RANI BOSE	3000
E08	GOUTAM DEY	5000
E09	PINAKI BOSE	5500

9 rows selected.

Arithmetic Expression in SELECT

It is possible to include arithmetic expressions involving column, literals with +, “* and / operators. Expressions on NUMBER and DATE data types can be used by means of the arithmetic operators.

Query: Display the employee name, salary and annual salary

```
SELECT ename, salary, salary * 12 FROM emp;
```

Output:

ENAME	SALARY	SALARY*12
KOUSHIK GHOSH	5000	60000
JAYANTA DUTTA	3500	42000
HARI NANDAN TUNGA	4000	48000
JAYANTAGANGULY	6000	72000
RAJIB HALDER	4000	48000
JISHNU BANERJEE	6500	78000
RANI BOSE	3000	36000
GOUTAM DEY	5000	60000
PINAKI BOSE	5500	66000

9 rows selected.

Operator Precedence

- Precedence over addition and subtraction is taken by multiplication and division.
- The same priority operators are evaluated from left to right.
- Parentheses are used to clarify statements and to force prioritized evaluation.

NOTES

Column Aliases in SELECT

To provide an alternate name to a column or expression, aliases are used that appears as column heading in the result set. For the above query, the following command can be used:

```
SELECT ename "Employee Name", salary "Salary", salary*12  
      "Annual salary" FROM emp;
```

Output:

Employee Name	Salary	Annual salary
KOUSHIK GHOSH	5000	60000
JAYANTA DUTTA	3500	42000
HARI NANDAN TUNGA	4000	48000
JAYANTAGANGULY	6000	72000
RAJIB HALDER	4000	48000
JISHNU BANERJEE	6500	78000
RANI BOSE	3000	36000
GOUTAM DEY	5000	60000
PINAKI BOSE	5500	66000

9 rows selected.

Column alias can also be given using 'AS' followed by alias name after column name. Only restriction is that alias should not contain spaces.

```
SELECT ename AS EmployeeName, salary AS Salary, salary*12 AS  
      Annual_salary FROM emp;
```

Output:

EmployeeName	Salary	Annual_salary
KOUSHIK GHOSH	5000	60000
JAYANTA DUTTA	3500	42000
HARI NANDAN TUNGA	4000	48000
JAYANTAGANGULY	6000	72000
RAJIB HALDER	4000	48000
JISHNU BANERJEE	6500	78000
RANI BOSE	3000	36000
GOUTAM DEY	5000	60000
PINAKI BOSE	5500	66000

9 rows selected.

|| Operator is used to concatenate character expressions. e.g.,

```
SELECT 'Designation of ' || ename || ' is ' || desg from emp;
```

NOTES

Output:

Designation of KOUSHIK GHOSH is SYSTEM ANALYST

Designation of JAYANTA DUTTA is PROGRAMMER

Designation of HARI NANDAN TUNGA is PROGRAMMER

Designation of JAYANTA GANGULY is ACCOUNTANT

Designation of RAJIB HALDER is CLERK

Designation of JISHNU BANERJEE is SYSTEM MANAGER

Designation of RANI BOSE is PROJECT ASSISTANT

Designation of GOUTAM DEY is PROGRAMMER

Designation of PINAKI BOSE is PROGRAMMER

9 rows selected.

Removing Duplicates—DISTINCT

Strictly speaking, it is incorrect to refer to SQL tables as relations as SQL queries might result in duplicate tuples.. SQL gives a mechanism to do away with duplicates. It is done by specifying the keyword **DISTINCT** after **SELECT**.

The default display of queries is all rows, including duplicate rows.

```
SELECT dno FROM emp;
```

Output:

DNO

D01

D01

D02

D03

D03

D02

D01

D01

D02

9 rows selected.

The above command will display all the dept numbers including duplicate dept numbers.

Instead of the above command if we use:

```
SELECT DISTINCT dno FROM emp;
```

Output:

DNO

D01

D02

D03

Notice that output contains non-duplicated department numbers. It is to be noted that we can use only one DISTINCT in the select column list.

Filtering Rows by Conditional Selection—WHERE Clause

The WHERE clause specifies the search condition and join criteria on the data that are selected. If a row fulfils the search conditions, it is returned as part of the result set.

The predicate employed in the WHERE clause is a simple comparison that uses the following:

- Relational Operators
- BETWEEN....AND....
- IN
- IS NULL
- IS NOT NULL
- LIKE

WHERE Clause and Relational Operator

These may be as follows:

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to

The relational operator condition is fulfilled when the expression on either side of the relational operator satisfies the relation set up by the operator. There are two other differences between the SQL operators and operators used in C/C++. First, the equality comparison and assignment operators, both represented by a single equal sign (=), are the same in SQL. Ambiguity is resolved by context. Second, the standard SQL inequality operator is represented by angle brackets (<>), though Oracle also supports the C/C++-style (!=).

- Comparisons with numeric column

Query: Display all employees getting salary over Rs 5000

```
SELECT * FROM emp WHERE salary > 5000;
```

NOTES

NOTES

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

- Comparisons with character and date column

String comparisons are performed using the numeric value of the characters, which are determined by the database character set. The character set is generally compatible with ASCII. The decimal values of the numbers and letters, for example, are found in the following table:

Decimal Values of ASCII Numbers and Letters

Character range	Decimal value range
0–9	48–57
A–Z	65–90
a–z	97–122

The most useful simple comparisons for strings can be done with the equality and inequality operators “=” and “<>”. They are used to select or omit specific rows. It is to be noted that

- Date values and character strings are placed within single quotation marks.
- Date values are format-sensitive and character values are case-sensitive .
- Default date format is ‘DD-MON-YY’.

Query: List the details of the employee whose name is ‘JAYANTA DUTTA’

```
SELECT * FROM emp WHERE ename = 'JAYANTA DUTTA'
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E02	JAYANTA DUTTA	3500	D01	PROGRAMMER	15-JAN-94

Query: Find the code, name and salary of employees of the department ‘D01’

```
SELECT ecode, ename, salary FROM emp WHERE dno='D01' ;
```

Output:

ECODE	ENAME	SALARY
E01	KOUSHIK GHOSH	5000
E02	JAYANTA DUTTA	3500
E07	RANIBOSE	3000
E08	GOUTAM DEY	5000

Query: List the information of the employees who are not working in the department 'D01'

```
SELECT * FROM emp WHERE dno<>'D01';
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

Note: The comparison operators, when used with strings, must match case correctly. There are SQL functions that are employed to convert strings in the database to every uppercase or lowercase to find a case-insensitive match. The functions are as follows:

- LOWER (CHAR) returns CHAR, with each letter in lowercase.
- UPPER (CHAR) returns CHAR, with each letter in uppercase.

Query: Find the code, name and salary of employees of the department 'D01'

```
SELECT ecode, ename, salary FROM emp WHERE UPPER(dno)='D01';
```

The equivalent SQL statement is as follows:

```
SELECT ecode, ename, salary  
FROM emp WHERE dno='D01' OR dno='d01';
```

Date and time comparisons are similar to comparisons with numbers because underlying every date and time is a number. In Oracle, there is a single data type, DATE, that represents both date and time with a single number. If we want to compare a date column with another date, we can use a string literal in the default date format and Oracle performs the conversion for:

Query: List the employees who joined after 20th September, 1996

```
SELECT * FROM emp WHERE dt_jn>'20-SEP-96';
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E07	RANI BOSE	3000	D01	PROJECT ASSISTANT	17-JUN-97
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97

The current date and time can be obtained by using the pseudo column, SYSDATE. Oracle automatically stores information, such as row numbers and row descriptions and Oracle is directly accessible, i.e., not through tables. This information is contained within pseudo columns. These pseudo columns can be retrieved in queries. These pseudo columns can be included in queries, which select data from tables.

NOTES

NOTES

Available **pseudo columns** in oracle include the following:

- **ROWNUM:** Row number. Order number in which a row value is retrieved.
- **ROWID:** Physical row (memory or disk address) location, i.e. unique row identification.
- **SYSDATE:** System or today's date.
- **UID:** User identification number indicating the current user.
- **USER:** Name of the currently logged in user.
- **Oracle Table 'DUAL'**

Oracle automatically creates DUAL table along with the data dictionary. DUAL is in the schema of the user SYS, however, it is accessible by the name DUAL to all users. It has one column, DUMMY, defined to be VARCHAR2(1), and contains one row with a value 'X'. Selecting from the DUAL table is useful for computing a constant expression with the SELECT statement. Since, DUAL has only one row, the constant is returned only once. Each Oracle account has access to a table called **dual**. We can query against this table to get the current account, system date/time, and execute mathematical functions.

The following example shows how to obtain the username used when the user logged into Oracle:

```
SELECT USER FROM DUAL;
```

Output:

USER

MANAS

```
SELECT SYSDATE FROM DUAL;
```

Output:

SYSDATE

15-APR-05

```
SELECT POWER(4, 3) FROM DUAL;
```

Output:

POWER(4,3)

64

Query: List all of the names of employees in the **EMP** table, the date they joined and the current system date

```
SELECT ename, dt_jn, SYSDATE FROM emp;
```

Output:

ENAME	DT_JN	SYSDATE
KOUSHIK GHOSH	10-MAR-93	15-APR-05
JAYANTA DUTTA	15-JAN-94	15-APR-05
HARI NANDAN TUNGA	01-JUL-95	15-APR-05
JAYANTA GANGULY	12-SEP-96	15-APR-05
RAJIB HALDER	07-OCT-95	15-APR-05
JISHNU BANERJEE	19-SEP-96	15-APR-05
RANI BOSE	17-JUN-97	15-APR-05
GOUTAM DEY	23-OCT-97	15-APR-05
PINAKI BOSE	26-AUG-94	15-APR-05

9 rows selected.

Suppose there is one of the employees, ASOK BASU, who has joined today. Then SYSDATE and dt_jn are the same. But if we query for the employees joined today using SYSDATE, we will find no results. When the dt_jn were entered using the default date format that doesn't have a time part. The time is defaulted to 12:00:00 a.m. SYSDATE, which is the current date and time, does have a time part even though only the date part is displayed by default.

```
SELECT * FROM emp WHERE dt_jn=SYSDATE;
```

The above command will display the message 'no rows selected'.

The TRUNC function is used to remove the time part of an Oracle DATE. This provides us a way to compare the dates in the table with today's date, disregarding any hours, minutes or seconds.

```
SELECT * FROM emp WHERE dt_jn=TRUNC (SYSDATE) ;
```

Similarly, suppose the date of join column is filled with SYSDATE as follows:

```
INSERT INTO emp
VALUES ('E11', 'KUNTAL GHOSH', 5000, 'D03', 'JRASSISTANT',
SYSDATE);
```

Say SYSDATE is 20-JUL-05.

Sometimes, to recover data based on something like:

```
SELECT * FROM emp WHERE dt_jn = '20-JUL-05' ;
```

It has been found that the output shows now rows selected. But there is a record for that day. What happened is that the records are not set to midnight (which is the default value if time of day is not specified).

One of the solutions is:

```
SELECT * FROM emp
WHERE dt_jn >= '20-JUL-05' AND dt_jn < '19-JUL-05' ;
```

NOTES

NOTES

WHERE Clause with Logical Operator

AND, OR, NOT are known as logical operators.

Operator	Meaning
AND	Returns TRUE if both component conditions are TRUE
OR	Returns TRUE if either component condition is TRUE
NOT	Returns TRUE if the following condition is FALSE

SQL has three logical values, TRUE, FALSE and NULL. Every condition, simple or compound, evaluates to one of these three values. In a WHERE clause, if this condition evaluates to TRUE, the row is returned if it is part of a SELECT statement. If it is FALSE or NULL, it is not.

NOT

FALSE and NULL are not the same. When FALSE is negated, TRUE is obtained. But when NULL is negated, we still get NULL. The following table is the truth table for NOT:

Truth Table for NOT

	NOT
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Query: List the employees who does not work in the department ‘D01’

```
SELECT * FROM emp WHERE dno <> 'D01' ;
```

The equivalent command using logical NOT is as follows:

```
SELECT * FROM emp WHERE NOT dno = 'D01' ;
```

Output:

EPCODE	ENAME	SALARY	DNO	DESG	DT_JN
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

AND

AND is used to combine two conditions. The following is the truth table for AND:

Truth Table for AND

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	NULL
NULL	NULL	NULL	NULL

Query: List the employees who does not work in the department ‘D01’

```
SELECT * FROM emp WHERE dno <> 'D01';
The equivalent command using logical NOT is as follows:
```

```
SELECT * FROM emp WHERE NOT dno = 'D01';
```

Output:

E CODE	E NAME	SALARY	D NO	D ESG	D T _ J N
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

The following is a query combining conditions with AND:

Query: List the employees of the department ‘D01’ who are getting salary over Rs 4000

```
SELECT * FROM EMP WHERE dno = 'D01' AND salary > 4000;
```

Output:

E CODE	E NAME	SALARY	D NO	D ESG	D T _ J N
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97

Query: List the employees who have joined in the year 1996

```
SELECT * FROM EMP
WHERE DT _ JN >= '01-JAN-96' AND DT _ JN <= '31-DEC-96';
```

Output:

E CODE	E NAME	SALARY	D NO	D ESG	D T _ J N
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96

OR

OR is also used to combine two conditions according to the following truth table:

The following is a sample query with OR:

Truth Table for OR

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Query: List all those who are getting salary less than Rs 5000 or work as Programmer, listed together.

```
SELECT ecode, ename, dno FROM emp WHERE salary < 5000 OR
desg='Programmer';
```

NOTES

NOTES

Output:

ECODE	ENAME	DNO
E02	JAYANTA DUTTA	D01
E03	HARI NANDAN TUNGA	D02
E05	RAJIB HALDER	D03
E07	RANI BOSE	D01

Query: List code and designation of the employees whose code is either 'E01' or 'E07'

```
SELECT ecode, desg FROM emp WHERE ecode='E01' OR ecode='E07';
```

Output:

ECODE	DESG
E07	PROJECT ASSISTANT
E01	SYSTEM ANALYST

Logical operator precedence

Logical operators have order of precedence. First parenthesized expressions are evaluated, then the NOT operator, followed by the AND operator and finally the OR operator.

Order Evaluated	Operator
1	All comparison operators
2	NOT
3	AND
4	OR

The rules of precedence can be overridden by using parentheses

Illustration 7.1:

Query: Display employee name, designation and salary whose designation is either Programmer or System Manager, and salary is greater than Rs 5000.

```
SELECT ename, desg, salary FROM emp WHERE desg='PROGRAMMER'  
OR desg='SYSTEMMANAGER' AND salary>5000;
```

Output:

ENAME	DESG	SALARY
JAYANTA DUTTA	PROGRAMMER	3500
HARI NANDAN TUNGA	PROGRAMMER	4000
GOUTAM DEY	PROGRAMMER	5000
PINAKI BOSE	PROGRAMMER	5500

The query gives wrong output. The output is not the one what we want. While employing the combinations of AND and OR, it is vital to use parentheses to make

sure the proper selection of rows returned to the results table. The parentheses give for exact criteria in the selection procedure.

If we write the query as follows:

```
SELECT ename, desg, salary FROM emp  
WHERE (desg='PROGRAMMER' OR desg='SYSTEM MANAGER') AND  
salary>5000;
```

Then output will be:

ENAME	DESG	SALARY
JISHNU BANERJEE	SYSTEM MANAGER	6500
PINAKI BOSE	PROGRAMMER	5500

This is the right output. First, SQL gets the rows where the designation is either PROGRAMMER or SYSTEM MANAGER; then considering this new list of rows, SQL sees if any of these rows satisfies the condition that the salary column is greater than Rs 5000.

WHERE Clause with IN Operator

The IN operator defines a set in which a given value may or may not be included. The IN operator checks for a value to match any value in a list of values. The format of IN comparisons is as follows:

```
Column_name [NOT] IN (value-1 [, value-2] ...)
```

Query: List all system managers and programmers

```
SELECT ecode, ename, dno, desg  
FROM emp  
WHERE desg IN ('SYSTEM MANAGER', 'PROGRAMMER');
```

is equivalent to

```
SELECT ecode, ename, dno, desg  
FROM emp  
WHERE desg='SYSTEM MANAGER' OR desg='PROGRAMMER';
```

Output:

ECODE	ENAME	DNO	DESG
E02	JAYANTA DUTTA	D01	PROGRAMMER
E03	HARI NANDAN TUNGA	D02	PROGRAMMER
E06	JISHNU BANERJEE	D02	SYSTEM MANAGER
E08	GOUTAM DEY	D01	PROGRAMMER
E09	PINAKI BOSE	D02	PROGRAMMER

NOTES

WHERE Clause with BETWEEN Operator

BETWEEN defines a range that value must fall in to make predicate true.

NOTES

The BETWEEN operator tests whether a value is *between* two other values. BETWEEN comparisons have the following format:

<column_name> [NOT] BETWEEN <value-1> AND <value-2>

This comparison tests if *column_name* is greater than or equal to *value-1* and less than or equal to *value-2*. It is equivalent to the following predicate:

column_name >= value-1 AND column_name <= value-2

Or, if NOT is included:

NOT (column_name >= value-1 AND column_name <= value-2)

For example,

Query: List those employees who are getting salary greater than or equal to Rs 4000, but less than or equal to Rs 6000

```
SELECT * FROM emp WHERE salary BETWEEN 4000 AND 6000;
```

It is equivalent to:

```
SELECT * FROM emp  
WHERE salary >= 4000 AND salary <= 6000;
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E01	KOUSHIK GHOSH	5000	D01	SYSTEMANALYST	10-MAR-93
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95
E04	JAYANTAGANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

Query: List all those who are not in this range

```
SELECT * FROM emp WHERE salary NOT BETWEEN 15000 AND 20000;
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E02	JAYANTA DUTTA	3500	D01	PROGRAMMER	15-JAN-94
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E07	RANIBOSE 3000	D01		PROJECT ASSISTANT	17-JUN-97

The Mystery of Oracle NULL

A column is said to be null, or to contain a null if a column in a row has no value. Nulls appears in columns of any data type that are not restricted by NOT NULL or are not PRIMARY KEY or a part of it. Null is used when the actual value is unknown or when a value is not meaningful. Null must not be used to represent a zero value, because they are not equivalent. Though **Oracle currently treats a character value with a length of zero as null**, this may not continue in the

future. It is suggested not to treat empty strings as the same as NULLs. An arithmetic expression containing a null always evaluates to null. As for example, null added to 15 is null. All operators (except concatenation) return null when given a null operand.

IS NULL/IS NOT NULL

Oracle implements and checks null values with the help of IS NULL and IS NOT NULL.

Query: List the employees whose designation is NULL

```
SELECT * FROM emp WHERE desg IS NULL;
```

String Comparisons with Wildcards

Character strings can be compared for equality, using the comparison operators if we want to compare entire strings for exact matches. SQL provides a special comparison operator, LIKE, which allows us to match portions of strings by using wildcard characters as placeholders in pattern strings to perform the comparison. There are two wildcard characters in SQL, underscore and percent, as shown in the following table:

Wildcard Characters in SQL

_	Matches any one single character
%	Matches any number of occurrences (including zero) of any character

These are similar to the wildcards (?) and (*) used in the DOS and UNIX environments, respectively.

To match a pattern string, one character—any character—must appear wherever the (_) appears. Any literals in the rest of the string (anything except the percent symbol, in other words) must appear exactly as it appears in the pattern strings. The following table shows some examples:

Examples of the Underscore Wildcard

Pattern	Matches
'_'	Any single letter string
'__'	Any string two letters long
'_BC'	Any string three letters long, ending in BC
'A_C'	Any string three letters long, beginning with A and ending with C
'_A_'	Any string three letters long, with middle letter A

The percent sign (%) can represent any number of any character or characters, including none. However, if there are any literals in the string or (_) in the string, these must be matched as well. The following table shows some examples:

NOTES

NOTES

Examples of the Percent Sign Wildcard

Pattern	Matches
'%'	Any string, but not NULL
'%A'	Any string that ends with A
'A%'	Any string that begins with A
'%A%'	Any string that contains the letter A
'_%A%'	Any string that contains the letter A, except as the first letter

The following are a few sample queries:

Query: List the employees whose name begin with 'J'

```
SELECT * FROM emp WHERE ename LIKE 'J%';
```

Output:

E CODE	E NAME	SALARY	D NO	D E S G	D T _ J N
E02	JAYANTA DUTTA	3500	D01	PROGRAMMER	15-JAN-94
E04	JAYANTAGANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96

Query: List name and salary of the employees whose title is BOSE

```
SELECT ename, salary FROM emp WHERE ename LIKE '%BOSE' ;
```

Output:

ENAME	SALARY
RANIBOSE	3000
PINAKI BOSE	5500

The optional ESCAPE subclause specifies an escape character for the pattern, allowing the pattern to use '%' and '_' (and the escape character) for matching. The ESCAPE value has to be a single character string. In the pattern, the ESCAPE character precedes any character to be escaped.

For example, to match a string ending with '%', use:

```
x LIKE '%/%' ESCAPE '/'
```

The following example escapes the escape character:

```
y LIKE '/%//%' ESCAPE '/'
```

The optional NOT reverses the result so that:

```
z NOT LIKE 'abc%'
```

is equivalent to:

```
NOT z LIKE 'abc%'
```

Rearranging Rows—ORDER BY Clause

The ORDER BY clause may be used to sort the retrieved rows. The general format of the ORDER BY clause is:

```
ORDER BY <column_name> [ASC | DESC], .....
```

Column_name,..... are column names either specified or implied in the select list. A new name is used in the ORDER BY list if a select column is renamed. ASC and DESC request ascending or descending sort for a column, respectively. The default is ASC.

For Example:

```
SELECT ecode, ename, salary, desg FROM emp ORDER BY ename DESC;
```

Output:

ECODE	ENAME	SALARY	DESG
E07	RANI BOSE	3000	PROJECT ASSISTANT
E05	RAJIB HALDER	4000	CLERK
E09	PINAKI BOSE	5500	PROGRAMMER
E01	KOUSHIK GHOSH	5000	SYSTEM ANALYST
E06	JISHNU BANERJEE	6500	SYSTEM MANAGER
E04	JAYANTA GANGULY	6000	ACCOUNTANT
E02	JAYANTA DUTTA	3500	PROGRAMMER
E03	HARI NANDAN TUNGA	4000	PROGRAMMER
E08	GOUTAM DEY	5000	PROGRAMMER

9 rows selected.

Multiple column name can be specified in the ORDER BY clause. ORDER BY sorts rows scanning the ordering columns in major-to-minor order and the left-to-right. On the first column name in the list the rows are sorted first. If there are any duplicate values for the first column, the duplicates are sorted on the second column (within the first column sort) in the ORDER BY list, and so on. For rows that have duplicate values for all Order By columns, there is no defined inner ordering.

For Example: To select the employees from emp table order first by ascending department number name and then by descending salary, issue the following statement:

```
SELECT ename, dno, salary FROM emp ORDER BY dno , salary DESC;
```

Output:

ENAME	DNO	SALARY
KOUSHIK GHOSH	D01	5000
GOUTAM DEY	D01	5000
JAYANTA DUTTA	D01	3500
RANI BOSE	D01	3000

NOTES

NOTES

JISHNU BANERJEE	D02	6500
PINAKI BOSE	D02	5500
HARI NANDAN TUNGA	D02	4000
JAYANTA GANGULY	D03	6000
RAJIB HALDER	D03	4000

9 rows selected.

While employing expressions in the select list, it can be more convenient to specify the select items by number (starting with 1). Both names and numbers can be intermixed.

```
SELECT ename, dno, salary FROM emp ORDER BY 2 ASC, 3 DESC;
```

The positional ORDER BY notation is used above, according to positions in the SELECT list.

Special processing in ORDER BY is needed by Database *nulls*. A *null* column sorts higher than all regular values; this is reversed for DESC. *Nulls* are considered duplicates of each other for ORDER BY in sorting.

Selecting Rows using ROWNUM Pseudo Column

The pseudocolumn ROWNUM is available since Oracle versions 7. For each row returned by a query, the ROWNUM pseudo column returns a number indicating the order in which Oracle selects the row from a table or a set of joined rows. The first row selected has a ROWNUM of 1, the second has 2, and so on.

ROWNUM can be used to limit the number of rows returned by a query as in the following example:

```
SELECT * FROM emp WHERE ROWNUM < 6;
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93
E02	JAYANTA DUTTA	3500	D01	PROGRAMMER	15-JAN-94
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95

If an ORDER BY clause follows ROWNUM in the same query, then the rows will be reordered by the ORDER BY clause.

For Example

```
SELECT * FROM emp WHERE ROWNUM < 6 ORDER BY salary;
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E02	JAYANTA DUTTA	3500	D01	PROGRAMMER	15-JAN-94
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95

E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96

NOTES

If the ORDER BY clause is embedded in a subquery and the ROWNUM condition is placed in the top-level query, then the ROWNUM condition can be forced to be applied after the ordering of the rows. This is occasionally called ‘top-N query’:

It in general refers to getting the top-n rows from a result set. For example, find the top 3 employees ranked by salary:

```
SELECT *
  FROM ( SELECT * FROM emp ORDER BY salary DESC )
 WHERE ROWNUM <= 3;
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

That is a top-n type query. The ROWNUM values are those of the top-level SELECT statement, and they are produced after the rows have already been ordered by salary in the subquery. This is also known as inline view. The later is a construct in Oracle SQL where we can place a query in the SQL FROM clause, just as if the query was a table name. A general use for inline views in Oracle SQL is to make simple complex queries by removing join operations and condensing several separate queries into a single query.

The above query can also be written as follows:

```
SELECT ename, salary
  FROM (SELECT ename, salary
        FROM emp
       ORDER BY salary DESC)
 WHERE ROWNUM < 4;
```

Output:

ENAME	SALARY
JISHNU BANERJEE	6500
JAYANTAGANGULY	6000
PINAKIBOSE	5500

Conditions that test for ROWNUM values larger than a positive integer are always false. As for example, the following query returns no rows:

```
SELECT * FROM employees WHERE ROWNUM > 1;
no rows selected
```

The first row that is fetched is assigned a ROWNUM of 1 and this makes the condition false. The second row to be fetched is now the first row and is also

NOTES

assigned a ROWNUM of 1 and this also makes the condition false. All rows subsequently fail to satisfy the condition, so no rows are returned.

Changing Date Formats using the ALTER SESSION Statement

In the earlier examples of SQL statements, the default format of data of type DATE was in the format: DD-MON-YY

The TO_CHAR and TO_DATE functions are used to convert dates to other formats; however, this is not convenient, especially when a large number of rows have to be inserted.

The ALTER SESSION statement alters various characteristics of the current SQL*Plus session including the default date format. This statement is often used to format dates to conform to regional customs. The following is the syntax for ALTER SESSION for use with changing the default date format:

ALTER SESSION

SET NLS_DATE_FORMAT = <date_format>;

The date_format can include the following codes:

YY	A 2-digit year such as 98
YYYY	A 4-digit year such as 1998
NM	A month number
MONTH	The full name of the month
MON	The abbreviated month (Jan, Feb, Mar)
DDD	The day of the year; for use in Julian dates
DD	The day of the month
D	The day of the week
DAY	The name of the day
HH	The hour of the day (12-hour clock)
HH24	The hour of the day (24-hour clock)
MI	The minutes
SS	The seconds

To change the default date to include a full four-digit year, give the following ALTER SESSION statement:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY'
```

From this point, all INSERT, UPDATE and DELETE statements must format the date accordingly. Also, any SELECT statements will return the date formatted accordingly.

Note that this change only remains in effect for the current session. Logging out of SQL*Plus and logging back in (or reconnecting to the Oracle database using the connect command) will reset the date format back to its default.

NOTES

Dealing with Centuries

The Oracle RDBMS stores dates using a four-digit year (the ‘OraDate’ format). To facilitate year 2000 compliance for applications that use the two-digit year format, Oracle provides a special year format mask ‘RR’. According to Oracle, the date format should be NLS_DATE_FORMAT = ‘YY-MON-RR’ in order to overwrite the default of ‘YY-MON-YY’. Using the ‘RR’ format, any two-digit year entered will be converted thus:

Current Year: Last Two Digits	Two-digit Year Specified	Year ‘RR’ Format Returns
0–49	0–49	Current Century
50–99	0–49	Current century + 1
0–49	50–99	Current century ? 1
50–99	50–99	Current Century

The ‘RR’ date format is available for inserting and updating DATE data in the database but it is not needed for retrieval/query of data already stored in the database as Oracle always stores the YEAR component of a date in its four-digit form. Therefore, irrespective of the current century, the ‘RR’ format will ensure that the year stored in the database is as follows:

Current Year	Specified Date	RR Format	YY Format
2005	20-AUG-14	2014	2014
2005	20-AUG-95	1995	2095
2060	20-AUG-14	2114	2014
2060	20-AUG-95	2095	2095

Aggregate Functions or Group Functions

To compute a single summary value from a column of individual values aggregate functions or group functions are used. Aggregate functions supported by oracle are described as follows:

AGGREGATE FUNCTIONS	DESCRIPTION
AVG([DISTINCT ALL]column_name)	Average value for a group of rows.
COUNT([DISTINCT ALL]column_name)	Counts the number of rows where the expression is not null. If we use * in the select clause, then all rows are counted whether they are null or not.
MIN([DISTINCT ALL]column_name)	Minimum of all values for a group of rows.
MAX([DISTINCT ALL]column_name)	Maximum of all values for a group of rows.
SUM([DISTINCT ALL]column_name)	Sum of all values for a group of rows.

Apart from the above functions, there are two more: STDDEV and VARIANCE. The basic group functions are AVG, COUNT, MIN, MAX and SUM. Their use is uncomplicated. These functions take two options: DISTINCT and ALL. ALL is default.

DISTINCT	This causes a group function to consider only distinct values in the argument expression.
ALL	This causes a group function to consider all values including all duplicates.

NOTES

All group functions except COUNT(*) ignore nulls. NVL is used in the argument to a group function to substitute a value for a null. If a query with a group function returns no rows or only rows with nulls for the argument to the group function, the group function returns a null.

Illustration 7.2:

Query: Compute the average salary of all employees listed in the EMP table

```
SELECT AVG(salary) FROM emp;
```

Output:

AVG(SALARY)
4722.22222

Query: Find the minimum and maximum salaries listed in the emp table

```
SELECT MIN(salary), MAX(salary) FROM emp;
```

Output:

MIN(SALARY)	MAX(SALARY)
3000	6500

Query: Find the total salary amount paid to employees per month in Department 'D04'

```
SELECT SUM(salary) FROM emp WHERE dno = 'D01';
```

Output:

SUM(SALARY)
16500

Query: Find out how many employees are given in the EMP table

```
SELECT COUNT(*) FROM emp;
```

Output:

COUNT(*)
9

DISTINCT is used with the COUNT function to count only distinct rows.

Query: Find how many different designations are listed in the EMP table

```
SELECT COUNT(DISTINCT desg) FROM emp;
```

Output:

COUNT(DISTINCTDESG)

6

Notes:

- Aggregate functions except **COUNT(*)** ignore null values in their input collection. Consequently, the collection of values is empty. The **COUNT** of an empty set is defined to be 0, and all other aggregate operations return a value of null when applied on an empty collection.
- Remember that an aggregate function cannot be used directly with the **WHERE** clause.

`SELECT * FROM emp WHERE salary=MAX(salary);`

Output:

`SELECT * FROM emp WHERE salary=MAX(salary)
*`

ERROR at line 1:

ORA-00934: group function is not allowed here

`SELECT * FROM emp WHERE AVG(salary)>3000;`

Output:

`SELECT * FROM emp WHERE AVG(salary)>3000
*`

ERROR at line 1:

ORA-00934: group function is not allowed here

Group functions might not be listed with column names in the **SELECT** clause (unless the **GROUP BY** clause is used, which is discussed below). For example, to get the name of the employee who is paid the lowest salary, the query,

`SELECT ename, MIN(salary) FROM emp;`

will generate an error.

`SELECT ename, MIN(salary) FROM emp
*`

ERROR at line 1:

ORA-00937: not a single-group group function

A subquery will return the desired information, which is discussed later in this unit. Interestingly, in the **ORDER BY** clause, group function may be specified.

For example:

```
SELECT dno, COUNT (*)
FROM emp
GROUP BY dno
ORDER BY COUNT (*) DESC;
```

NOTES

NOTES

Output:

DNO	COUNT(*)
D01	4
D02	3
D03	2

It should be noted that if a column in the ORDER BY clause is specified that is not a part of group function, it must be in the GROUP BY clause.

DISTINCT IN GROUP FUNCTION

All group value functions have a DISTINCT versus ALL options. COUNT provides good example of how this works.

```
SELECT COUNT (dno) FROM emp;
```

Output:

COUNT(DNO)
9

```
SELECT COUNT (DISTINCT dno) FROM emp;
```

Output:

COUNT(DISTINCTDNO)
3

It is clearly seen that DISTINCT forces COUNT to count only the number of unique department member.

GROUP QUERY—GROUP BY Clause

The GROUP BY clause basically allows us to partition the results of a query into groups with similar characteristics based upon predicate satisfaction. The columns named in the GROUP BY clause are called the *grouping columns*. The ISO standard requires the SELECT clause and the GROUP BY clause to be closely integrated. When GROUP BY is employed, each item in the SELECT list has to be single-valued per group. Further, the SELECT clause may contain only column names, aggregate functions, constants and an expression involving combinations of these.

The GROUP BY clause has the following general format:

```
GROUP BY <column_name1> [, <column_name2>] .....
```

column_name1 and column_name2 are the grouping columns. They have to be names of columns from tables in the FROM clause, not expressions.

GROUP BY operates on the rows from the FROM clause as filtered by the WHERE clause. GROUP BY assumes a *null* as distinct from every other *null* and so each row, which has a *null* in one of its grouping columns, forms a separate group.

Query: List the total amount paid in salary to each department*Form of Basic SQL
Query*

SELECT SUM(salary) FROM emp GROUP BY dno;

Output:

SUM(SALARY)
16500
16000
10000

NOTES

The GROUP BY clause causes a select statement to produce one summary row for all selected rows that have the identical value as specified in the GROUP BY clause. The column referred to in the GROUP BY clause does not have to be in the select clause. However, it is not known with which group the returned data is associated, if the column referred to in the GROUP BY clause is not listed in the SELECT clause.. Without the DNO column in the SELECT clause, we have data, but it is not known which department is associated with each row of data. To solve this problem, write the query as follows:

SELECT dno, SUM(salary) FROM emp GROUP BY dno;

DNO	SUM(SALARY)
D01	16500
D02	16000
D03	10000

This results table furnishes information that is helpful.

Query: Find the number of employees in each department from the EMPLOYEE table

SELECT dno, COUNT(*) FROM emp GROUP BY dno;

Output:

DNO	COUNT(*)
D01	4
D02	3
D03	2

Multiple aggregate functions may be used in the SELECT clause in association with the GROUP BY clause.

Query: Find the minimum maximum salary, salary and the average salary for each department

```
SELECT dno, MIN(salary),  
MAX(salary), AVG(salary) ← group functions in select clause  
FROM emp  
GROUP BY dno; ← GROUP BY clause
```

NOTES

Output:

DNO	MIN(SALARY)	MAX(SALARY)	AVG(SALARY)
D01	3000	5000	4125
D02	4000	6500	5333.33333
D03	4000	6000	5000

Multiple columns may also be used in the GROUP BY clause.

Query: Find the number of designation in each department

```
SELECT dno, desg, COUNT(*) FROM emp GROUP BY dno, desg;
```

Output:

DNO	DESG	COUNT(*)
D01	PROGRAMMER	2
D01	PROJECT ASSISTANT	1
D01	SYSTEM ANALYST	1
D02	PROGRAMMER	2
D02	SYSTEM MANAGER	1
D03	ACCOUNTANT	1
D03	CLERK	1

7 rows selected.

- All the column names in the SELECT list must show in the GROUP BY clause unless the name is used only in an aggregate function; otherwise, SQL generates the following error. The contrary is not true, however, as there may be column names in the GROUP BY clause, which may not appear in the SELECT list. Syntactically it is:

```
SELECT dno, desg, COUNT(*) FROM emp GROUP BY dno;
```

ORA-00979: not a group by expression

- If anyone wants to use aggregate expression in a SELECT list without the GROUP BY clause, then the SELECT list must consist only of aggregate function.

When the WHERE clause is applied in a grouped query, the WHERE clause is applied first, then the groups are formed from the residual rows that satisfy the search condition.

Query: Count the designation wise number of employees of the department 'D01'

```
SELECT desg, COUNT(*) FROM emp WHERE dno='D01' GROUP BY desg;
```

Output:

DESG	COUNT(*)
PROGRAMMER	2
PROJECTASSISTANT	1
SYSTEMANALYST	1

NOTES

It should be noted that the WHERE clause must come into view before the GROUP BY clause.

The ISO standard stipulates two nulls to be equal for purposes of the GROUP BY clause. Thus, if two rows have nulls in the same grouping columns and identical values in all the non-null grouping columns, they are combined into the same group.

Combining Group Functions and Arithmetic Functions

Suppose, we want to find out how much total salary is paid out per month to all employees listed in the emp table, and also want to find out how much total monthly salary would be paid if every employee is given a 3 per cent raise in salary. A single SQL query would give this information.

```
SELECT SUM(salary), SUM(salary * 1.03) FROM emp;
```

Output:

SUM(SALARY)	SUM(SALARY*1.03)
42500	43775

We may choose names for columns in the results by including the names in the SELECT clause as illustrated below:

```
SELECT SUM(salary) NOW, SUM(salary * 1.03) WITHRAISE  
FROM emp;
```

This query returns the following results table:

NOW	WITHRAISE
42500	43775

Be cautious when naming columns in results tables. Reserved words must be avoided, and must not include symbols other than the underscore. Otherwise, Oracle will generates an error.

Query: Get the number of SYSTEM ANALYST, PROGRAMMER, PROJECT ASSISTANT, SYSTEM MANAGER in each department

```
SELECT  
dno  
, SUM(decode(desg, 'SYSTEMANALYST', 1, 0)) "SYSTEMANALYST"  
, SUM(decode(desg, 'PROGRAMMER', 1, 0)) "PROGRAMMER"  
, SUM(decode(desg, 'PROJECT ASSISTANT', 1, 0)) "PROJECT  
ASSISTANT"  
, SUM(decode(desg, 'SYSTEMMANAGER', 1, 0)) "SYSTEMMANAGER"
```

Form of Basic SQL Query

```
FROM emp  
GROUP BY dno;
```

Output:

NOTES

DNO	SYSTEM ANALYST	PROGRAM MER PROJECT	ASSISTANT MANAGER	SYSTEM
D01	1	2	1	0
D02	0	2	0	1
D03	0	0	0	0

HAVING Clause with GROUP BY

The HAVING clause is intended for use with the GROUP BY clause to confine the groups that come out in the final result. Although HAVING and WHERE are similar in syntax, they serve different purposes. The WHERE clause filters individual rows going into the result table, whereas the HAVING clause filters groups going into the final result. The WHERE clause eliminates rows earlier to summarization and the HAVING clause filters groups going into the final result. The ISO standard stipulates that column names employed in the HAVING clause must also appear in the GROUP BY clause or be restricted within an aggregate function. In practice, the search condition in the HAVING clause always includes at least one aggregate function; otherwise, the search condition could be moved to the WHERE clause and applied to individual rows. Therefore, the Aggregate Functions used in the HAVING clause. However, they cannot be applied in the WHERE clause.

The general format of the HAVING Clause is as follows:

```
HAVING <search_condition>
```

The search condition applies to each group. It can be formed using predicates like between, in, like, null, comparison, etc. Combined with Boolean (AND, OR, NOT) operators. Since, the search condition is grouped table. The predicates should be

- On a column by which grouping is done
- A group function (Aggregate function) on other columns

If the Having predicate evaluates to true for a grouped or aggregate row, that row is included in the query result; otherwise, the row is skipped.

As for example, if we want to find which departments have at least 3 employees. Performing the query above with the HAVING clause we need to return only the rows with more than two employees per department. That query would be as follows:

```
SELECT dno, COUNT(*) FROM emp GROUP BY dno HAVING COUNT(*) > 2;
```

Output:

DNO	COUNT(*)
D01	4
D02	3

NOTES

Note:

- If an expression is used in the HAVING clause that is not in the SELECT list or that is not an aggregate function, then oracle will generate an error.
- Oracle does not care whether the HAVING clause before GROUP BY clause or after. GROUP BY clause can be specified before HAVING clause and vice versa.

Identifying Duplicate Rows using GROUP BY and HAVING

It is feasible to establish duplicate rows using the select with a count of all the rows that have the same values in the ‘unique’ fields as given here.

```
select a,b,count(*)  
from test  
group by a,b  
having count(*) > 1;
```

Order of Execution of WHERE/GROUP BY/HAVING/ORDER BY

- WHERE clause.
- GROUP BY clause.
- Group functions for each group.
- Elimination of groups based on the HAVING clause.
- the ORDER BY clause. The ORDER BY clause has to use either a group function or a column given in the GROUP BY clause.

The order of execution is essential, because it has a direct bearing on the task of the queries. In general, the more the records that can be eliminated via the WHERE clause, the faster the execution of the query due to the decline in the number of rows that should be processed at the time of the GROUP BY operation.

7.2.6 Join in SQL

In this section, in addition to EMP and DEPT tables, two more tables are introduced.

PROJECT

PID	VARCHAR2(5)
PNAME	VARCHAR2(30)
LOCATION	VARCHAR2(30)

Instance of the PROJECT table:

PID	PNAME	LOCATION
P01	HOSPITAL MANAGEMENT SYSTEM	KOLKATA
P02	ACCOUNTING SYSTEM	KOLKATA
P03	BANKING INFORMATION SYSTEM	CHENNAI

ASSIGN

ECODE	VARCHAR2(5)
PID	VARCHAR2(5)

NOTES

Instance of the ASSIGN table:

ECODE	PID
E01	P01
E01	P02
E01	P03
E02	P01
E08	P01
E02	P02
E08	P02
E07	P02
E03	P03
E09	P03
E06	P03

Join is discussed in quite details in Relational algebra. A **join** is a query that retrieves rows from more than one table or view. More than one table can be specified in the **FROM** clause. Sometimes more than one table may have the same column name. When we use these tables for join, this causes a problem with Oracle; it does not identify the table to which the column pertains. When this occurs, Oracle will terminate the query with the following error message:

‘Column ambiguously defined’

To overcome this problem, there are two methods of qualifying a column:

- The column name should be preceded with the table name to qualify all references to these columns throughout the query with table names to avoid vagueness. Two names should be separated by a period; for example emp.dno.
- The alternate way is to define the temporary labels in the **FROM** clause and use them somewhere else in the query. These type of temporary labels are sometimes referred to as table aliases. There are the following two ways to specify a column alias:
 - o Naming the alias after the column specification separated by a space.

Example:

```
SELECT ecode id, ename name FROM employee
```

- o Use of ‘AS’ word to specify the alias more clearly

Example:

```
SELECT ecode AS id, ename AS name FROM employee
```

Like before, alias name should be used with a dot before column name; e.g.

```
table_alias_name.column_name.
```

The advantage of second method is that it allows the developer to use a shorter custom name for the table. The table alias and the second method is must for self-join where a table is joined with itself. In that case, both table and column names are same.

Join Conditions

Most of the join queries contain the WHERE clause conditions that compare two columns, each from a different table. Such a condition is referred to as **join condition**. For executing a join, Oracle combines pairs of rows, each containing one row from each table, for which the join condition evaluates to TRUE. The columns in the join conditions need not appear in the select list.

In pre-Oracle 9i, join condition is specified using the WHERE clause. In Oracle 9i it is supported by the ANSI SQL-99 syntax. The following join types are available:

- *Cross join*
- *Natural join*
- *Join with USING clause*
- *Join with ON clause*
- *Outer join (right, left and full)*

The order of tables in the FROM clause does not matter. Also, the order of comparisons does not matter.

Cartesian Product

If two tables in a join query have no join condition, their **Cartesian product is returned**. It is also known as cross-join. Each row of one table is combined with each row of the other. Consider the following query illustrating a Cartesian product:

```
SELECT ecode, ename, salary, dname  
FROM emp, dept
```

Output:

ECODE	ENAME	SALARY	DNAME
E01	KOUSHIK GHOSH	5000	PROJECT
E02	JAYANTA DUTTA	3500	PROJECT
E03	HARI NANDAN TUNGA	4000	PROJECT
E04	JAYANTA GANGULY	6000	PROJECT
E05	RAJIB HALDER	4000	PROJECT

NOTES

NOTES

E06	JISHNU BANERJEE	6500	PROJECT
E07	RANI BOSE	3000	PROJECT
E08	GOUTAM DEY	5000	PROJECT
E09	PINAKI BOSE	5500	PROJECT
E01	KOUSHIK GHOSH	5000	RESEARCH
E02	JAYANTA DUTTA	3500	RESEARCH
E03	HARI NANDAN TUNGA	4000	RESEARCH
E04	JAYANTA GANGULY	6000	RESEARCH
E05	RAJIB HALDER	4000	RESEARCH
E06	JISHNU BANERJEE	6500	RESEARCH
E07	RANI BOSE	3000	RESEARCH
E08	GOUTAM DEY	5000	RESEARCH
E09	PINAKI BOSE	5500	RESEARCH
E01	KOUSHIK GHOSH	5000	PERSONNEL
E02	JAYANTA DUTTA	3500	PERSONNEL
E03	HARI NANDAN TUNGA	4000	PERSONNEL
E04	JAYANTA GANGULY	6000	PERSONNEL
E05	RAJIB HALDER	4000	PERSONNEL
E06	JISHNU BANERJEE	6500	PERSONNEL
E07	RANI BOSE	3000	PERSONNEL
E08	GOUTAM DEY	5000	PERSONNEL
E09	PINAKI BOSE	5500	PERSONNEL

27 rows selected.

A Cartesian product contains many rows of no practical interest. There are 9 rows in EMP and 3 rows in DEPT; this will result in 9 times 3 (i.e. 27) rows.

In Oracle 9i, the **cross-join** represents the Cartesian product of two or more tables selected without join conditions. The same output as above will be given.

```
SELECT ecode, ename, salary, dname  
FROM emp CROSS JOIN dept;
```

It is important to note here that a Cartesian product of two tables is in practice required only very rarely.

Equijoins

An **equijoin** is a join in which the join condition contains an equality operator. In pre-Oracle 9i join condition is specified in the WHERE clause. In Oracle 9i, join predicates can also be defined with ON.

NOTES

Query: List the employee code, name and department name of each employee

Using the WHERE clause	Using the ON clause
<pre>SELECT ecode, ename, dname FROM emp, dept WHERE emp.dno=dept.dno;</pre>	<pre>SELECT ecode, ename, dname FROM emp JOIN dept ON emp.dno=dept.dno;</pre>

Output:

ECODE	ENAME	DNAME
E01	KOUSHIK GHOSH	PROJECT
E02	JAYANTA DUTTA	PROJECT
E03	HARI NANDAN TUNGA	RESEARCH
E04	JAYANTAGANGULY	PERSONNEL
E05	RAJIB HALDER	PERSONNEL
E06	JISHNU BANERJEE	RESEARCH
E07	RANI BOSE	PROJECT
E08	GOUTAM DEY	PROJECT
E09	PINAKI BOSE	RESEARCH

9 rows selected.

When we join the table **dept** to the table **emp**, the join condition contains the equality operator. Such joins are known as **equijoins**.

The above query may be written as follows:

```
SELECT ecode, ename, dname
FROM emp e, dept d
WHERE e.dno = d.dno;
```

OR

```
SELECT ecode, ename, dname
FROM emp e JOIN dept d
ON e.dno=d.dno
```

The table **emp** is given an alias e, and **dept** an alias d. Generally, alias or table name may be omitted if the column names are unique.

All other conditions could also be defined in the **ON** clause. For example,

```
SELECT ename, d.dno, d.dname
FROM emp e INNER JOIN dept d
ON (e.dno = d.dno AND desg = 'PROGRAMMER');
```

NOTES

Output:

ENAME	DNO	DNAME
JAYANTA DUTTA	D01	PROJECT
HARI NANDAN TUNGA	D02	RESEARCH
GOUTAM DEY	D01	PROJECT
PINAKI BOSE	D02	RESEARCH

However, it is better to separate the join condition from the restricting conditions (**WHERE**). The following statement would, therefore, be better:

```
SELECT e.ename, d.dno, d.dname
FROM emp e INNER JOIN dept d ON (e.dno = d.dno)
WHERE desg = 'PROGRAMMER';
```

USING Clause

In the above example, the join column was specified in the **ON** clause. Specifying the join condition can be simplified by the **USING** clause if the following conditions are satisfied:

- The join depends on an equality condition between two columns or between sets of two columns to relate the rows from the two tables.
- The columns must have the same name and data type in both tables.

The following query

```
SELECT ecode, ename, dname
FROM emp JOIN dept
ON emp.dno=dept.dno;
```

This can be rewritten as:

```
SELECT ecode, ename, dname
FROM emp JOIN dept
USING (dno);
```

The **USING** clause, however, does subtly affect the semantics of the query. It is to be remembered that if a join column is included in the **SELECT** list, Oracle does not allow qualifying that column with a table name or tabling alias resulting in an error message.

```
SELECT ecode, ename, dept.dno, dname
FROM emp JOIN dept
USING (dno);
```

Output:

```
SELECT ecode, ename, dept.dno, dname
*
ERROR at line 1:
ORA-25154: column part of USING clause cannot have qualifier
```

NOTES

Any join that does not use the equality operator(=) is known as a **non-equijoin**.
For example,

Using the WHERE clause	Using the ON clause
<pre>SELECT ecode, ename, salary, dname FROM emp,dept WHERE salary > 5000;</pre>	<pre>SELECT ecode, ename, salary, dname FROM emp JOIN dept ON salary > 5000;</pre>

Output:

ECODE	ENAME	SALARY	DNAME
E04	JAYANTA GANGULY	6000	PROJECT
E06	JISHNU BANERJEE	6500	PROJECT
E09	PINAKI BOSE	5500	PROJECT
E04	JAYANTA GANGULY	6000	RESEARCH
E06	JISHNU BANERJEE	6500	RESEARCH
E09	PINAKI BOSE	5500	RESEARCH
E04	JAYANTA GANGULY	6000	PERSONNEL
E06	JISHNU BANERJEE	6500	PERSONNEL
E09	PINAKI BOSE	5500	PERSONNEL

9 rows selected.

Some more examples of equijoin are as follows:

Query: List the code, name of the employees with their corresponding project ids in which they are assigned

Using the WHERE clause	Using the ON clause
<pre>SELECT e.ecode, ename, a.pid FROM emp e,assign a WHERE e.ecode=a.ecode;</pre>	<pre>SELECT e.ecode, ename, a.pid FROM emp e JOIN assign a ON e.ecode=a.ecode;</pre>

Output:

ECODE	ENAME	PID
E01	KOUSHIK GHOSH	P01
E01	KOUSHIK GHOSH	P02
E01	KOUSHIK GHOSH	P03
E02	JAYANTA DUTTA	P01
E08	GOUTAM DEY	P01

NOTES

E02	JAYANTA DUTTA	P02
E08	GOUTAM DEY	P02
E07	RANI BOSE	P02
E03	HARI NANDAN TUNGA	P03
E09	PINAKI BOSE	P03
E06	JISHNU BANERJEE	P03

11 rows selected.

Query: List the name of the projects and department names located at same city

Using the WHERE clause	Using the ON clause
<pre>SELECT pname, dname FROM project p, dept d WHERE p.location=d.city;</pre>	<pre>SELECT pname, dname FROM project p JOIN dept d ON p.location=d.city;</pre>

Output:

PNAME	DNAME
BANKING INFORMATION SYSTEM	RESEARCH
HOSPITAL MANAGEMENT SYSTEM	PROJECT
ACCOUNTING SYSTEM	PROJECT
HOSPITAL MANAGEMENT SYSTEM	PERSONNEL
ACCOUNTING SYSTEM	PERSONNEL

Apart from the join conditions, the WHERE clause or the ON clause can also contain other conditions that refer to columns of only one table.

Query: List the employee code, name and department name of each employee getting salary more than Rs 5000

Using the WHERE clause	Using the ON clause
<pre>SELECT ecode, ename, dname FROM emp, dept WHERE emp.dno=dept.dno AND salary>5000;</pre>	<pre>SELECT ecode, ename, dname FROM emp JOIN dept ON emp.dno=dept.dno AND salary>5000;</pre>

Output:

ECODE	ENAME	DNAME
E04	JAYANTA GANGULY	PERSONNEL
E06	JISHNU BANERJEE	RESEARCH
E09	PINAKI BOSE	RESEARCH

Query: List the employee code, name and department name of all PROGRAMMERS

Form of Basic SQL Query

Using the WHERE clause	Using the ON clause
<pre>SELECT ecode, ename, dname FROM emp, dept WHERE emp.dno=dept.dno AND desg='PROGRAMMER';</pre>	<pre>SELECT ecode, ename, dname FROM emp JOIN dept ON emp.dno=dept.dno AND salary>5000;</pre>

NOTES

Output:

ECODE	ENAME	DNAME
E02	JAYANTA DUTTA	PROJECT
E03	HARI NANDAN TUNGA	RESEARCH
E08	GOUTAM DEY	PROJECT
E09	PINAKI BOSE	RESEARCH

Sorting a Join

We can use ORDER BY clause also in join.

Query: List the code, name of the employees with their corresponding project ids in which they are assigned in descending order of their employee code

Using the WHERE clause	Using the ON clause
<pre>SELECT e.ecode, ename, pid FROM emp e, assign a WHERE e.ecode=a.ecode ORDER BY e.ecode DESC;</pre>	<pre>SELECT e.ecode, ename, pid FROM emp e JOIN assign a ON e.ecode=a.ecode ORDER BY e.ecode DESC;</pre>

Output:

ECODE	ENAME	PID
E09	PINAKI BOSE	P03
E08	GOUTAM DEY	P01
E08	GOUTAM DEY	P02
E07	RANI BOSE	P02
E06	JISHNU BANERJEE	P03
E03	HARI NANDAN TUNGA	P03
E02	JAYANTA DUTTA	P01
E02	JAYANTA DUTTA	P02
E01	KOUSHIK GHOSH	P01
E01	KOUSHIK GHOSH	P02
E01	KOUSHIK GHOSH	P03

11 rows selected.

NOTES

Join Conditions Involving Multiple Columns

If a join condition consists of multiple columns from each table, it is needed to specify all the predicates in the `ON` clause or in the `WHERE` clause. For example, if tables A and B are joined based on columns `c1` and `c2`, the join condition would be as follows:

Using the WHERE clause	Using the ON clause
<pre>SELECT FROM A , B WHERE A.c1=B.c1 AND A.c2=B.c2;</pre>	<pre>SELECT FROM A JOIN B ON A.c1=B.c1 AND A.c2=B.c2;</pre>

If the column names are identical in the two tables, the `USING` clause can be used as follows:

```
SELECT .....
FROM A JOIN B
USING(c1,c2);
```

For illustration, say there is a `CITY` column in the `EMP` table specifying the city in which the employee is staying.

Query: Find the employees who are working in the same city as their dept

Using the WHERE clause	Using the ON clause	Using the USING clause
<pre>SELECT ecode, ename, salary FROM emp a, dept b WHERE a.city=b.city AND a.dno=b.dno;</pre>	<pre>SELECT ecode, ename, salary FROM emp a JOIN dept b ON a.city=b.city AND a.dno=b.dno;</pre>	<pre>SELECT ecode, ename, salary FROM emp a JOIN dept b USING(city,dno);</pre>

Joining Multiple Tables

In a join more than two tables can participate. This is basically an extension of a two-table join. Three tables, *A*, *B* and *C* can be joined in the following ways:

- *A* joins *B* which joins *C*.
- *A* joins *B* and the join of *A* and *B* joins *C*.
- *A* joins *B* and *A* joins *C*.

Besides the above, several other variations are available. Oracle first joins two of the tables that are based on the join conditions, comparing their columns, and then joins the result to another table based on join conditions containing columns of the joined tables and the new table. Till the point all tables are joined into the result this process continues. The order in which Oracle joins tables based on the join conditions, indexes on the tables, etc., is determined by the optimizer.

NOTES

Query: Display codes, names of employees with their corresponding assigned project name and location of the project

Using the WHERE clause	Using the ON clause
<pre>SELECT e.ecode,ename,pname,location FROM emp e, project p, assign a WHERE e.ecode=a.ecode AND p.pid=a.pid;</pre>	<pre>SELECT e.ecode,ename,pname,location FROM (emp e JOIN assign a ON e.ecode=a.ecode) JOIN project p ON p.pid=a.pid;</pre>

Output:

ECODE	ENAME	PNAME	LOCATION
E01	KOUSHIK GHOSH	HOSPITAL	MANAGEMENT SYSTEM KOLKATA
E01	KOUSHIK GHOSH	ACCOUNTING	SYSTEM KOLKATA
E01	KOUSHIK GHOSH	BANKING	INFORMATION SYSTEM CHENNAI
E02	JAYANTA DUTTA	HOSPITAL	MANAGEMENT SYSTEM KOLKATA
E08	GOUTAM DEY	HOSPITAL	MANAGEMENT SYSTEM KOLKATA
E02	JAYANTA DUTTA	ACCOUNTING	SYSTEM KOLKATA
E08	GOUTAM DEY	ACCOUNTING	SYSTEM KOLKATA
E07	RANI BOSE	ACCOUNTING	SYSTEM KOLKATA
E03	HARI NANDAN TUNGA	BANKING	INFORMATION SYSTEM CHENNAI
E09	PINAKI BOSE	BANKING	INFORMATION SYSTEM CHENNAI
E06	JISHNU BANERJEE	BANKING	INFORMATION SYSTEM CHENNAI

11 rows selected.

It is to be noted that when joining more than two tables, parentheses are used to control the join order. In the absence of parentheses, the joins are processed from left to right. The example above uses parentheses for explicitly specifying the default join order.

The following query will give the same output:

```
SELECT e.ecode,ename,pname,location
FROM emp e JOIN assign a ON e.ecode=a.ecode
JOIN project p ON p.pid=a.pid;
```

However, the following one is incorrect:

```
SELECT e.ecode,ename,pname,location
FROM emp e JOIN assign a JOIN project p
ON p.pid=a.pid
AND e.ecode=a.ecode;
```

Output:

```
AND e.ecode=a.ecode
*
ERROR at line 4:
ORA-00905: missing keyword
```

NOTES

Query: Retrieve the code, name and department number of the employees who are working in the project in the same city as their department

```
SELECT DISTINCT E.ecode, ename, E.dno
FROM assigns A,dept D,project P,emp E
WHERE A.pid=P.pid
AND e.ecode=A.ecode
AND D.city=P.location;
```

Output:

ECODE	ENAME	DNO
E01	KOUSHIK GHOSH	D01
E02	JAYANTA DUTTA	D01
E03	HARI NANDAN TUNGA	D02
E06	JISHNU BANERJEE	D02
E07	RANI BOSE	D01
E08	GOUTAM DEY	D01
E09	PINAKI BOSE	D02

7 rows selected.

More than two tables can be used in the FROM clause. There is no theoretical limit; however, the system might place some limit. If there are N tables in the FROM clause, then normally (N – 1) join conditions are needed.

Natural Join

The natural join is based on table columns with the same data and name type. This join automatically integrates into the join condition all columns with the similar data and name type. Here join condition need not be specified. The SELECT * syntax does not, however, return columns doubly. Obviously, it is particularly Oracle 9i feature. For illustration, a previously discussed query is taken.

Query: List the employee code, name and department name of each employee

```
SELECT ecode,ename, salary, desg
FROM emp NATURAL JOIN dept;
```

Output:

ECODE	ENAME	SALARY	DESG
E01	KOUSHIK GHOSH	5000	SYSTEMANALYST
E02	JAYANTA DUTTA	3500	PROGRAMMER
E03	HARI NANDAN TUNGA	4000	PROGRAMMER
E04	JAYANTA GANGULY	6000	ACCOUNTANT
E05	RAJIB HALDER	4000	CLERK

E06	JISHNU BANERJEE	6500	SYSTEM MANAGER
E07	RANI BOSE	3000	PROJECT ASSISTANT
E08	GOUTAM DEY	5000	PROGRAMMER
E09	PINAKI BOSE	5500	PROGRAMMER

NOTES

9 rows selected.

Table aliases are not used for the join columns in the select list. In case of natural join, Oracle recognizes only one version of each join column.

If we write the above query as follows, it will generate an error:

```
SELECT ecode, ename, salary, d.dno, desg
  FROM emp NATURAL JOIN dept;
```

Output:

```
SELECT ecode, ename, salary, d.dno, desg
*
```

ERROR at line 1:

ORA-00904: invalid column name

As usual, one or more WHERE conditions may follow the join clause:

```
SELECT ename, dno, dname
  FROM emp NATURAL JOIN dept
 WHERE dno = 'D02';
```

Output:

ENAME	DNO	DNAME
HARI NANDAN TUNGA	D02	RESEARCH
JISHNU BANERJEE	D02	RESEARCH
PINAKI BOSE	D02	RESEARCH

The USING clause is another shortcut for performing an equijoin. The difference between USING and NATURAL is that with the former, the join columns are specified explicitly. This case is also true for join with the ON clause where join condition needs to specify explicitly, i.e. the ON clause is mandatory here. Look at the following query:

```
SELECT ecode, ename, salary, desg
  FROM emp JOIN dept;
```

Output:

```
FROM emp JOIN dept
*
```

ERROR at line 2:

ORA-00905: missing keyword

NOTES

Inner Join in SQL

An **inner join** (sometimes called ‘simple join’) is a join of two or more tables that returns only those rows that satisfy the join condition.

INNER JOIN keywords can be specified in the FROM clause to perform an inner join between the two tables. The ON clause used to specify the join conditions.

Query: List the names of each employee with his/her department name

```
SELECT ename, dname
FROM emp INNER JOIN dept
ON emp.dno=dept.dno;
```

Output:

ENAME	DNAME
KOUSHIK GHOSH	PROJECT
JAYANTA DUTTA	PROJECT
HARI NANDAN TUNGA	RESEARCH
JAYANTAGANGULY	PERSONNEL
RAJIB HALDER	PERSONNEL
JISHNU BANERJEE	RESEARCH
RANI BOSE	PROJECT
GOUTAMDEY	PROJECT
PINAKI BOSE	RESEARCH

9 rows selected.

Other clauses like WHERE and ORDER BY can be used, which come after the FROM clause.

```
SELECT ename, dname
FROM emp INNER JOIN dept
ON emp.dno=dept.dno
WHERE dname='PROJECT'
ORDER BY ename;
```

Output:

ENAME	DNAME
GOUTAM DEY	PROJECT
JAYANTA DUTTA	PROJECT
KOUSHIK GHOSH	PROJECT
RANI BOSE	PROJECT

Outer Join in SQL

An *inner join* does not include rows from either table that does not have a matching row in the other table. A basic goal of an outer join is to extend the result of a simple join to include rows from one table that may not have matching rows in another table. This means that an **outer join** will returns all rows that satisfy the

NOTES

join condition and in addition will include those rows from one table for which no rows from the other table to satisfy the join condition in its result set. The outer join combines the unmatched row in one of the tables with an artificial row for the other table. This artificial row has all columns set to *null*.

The ANSI syntax identifies three types of outer join: right outer joins, left outer joins, and full outer joins. The right and left outer joins are the same thing—all rows from one table are included, along with any matching rows from the other table. The one and only difference between a right and a left outer join is the order in the tables are listed by the programmer. The full outer join returns all rows from both the tables. Rows are matched on the join columns wherever possible, and *NULLs* are used to fill up the empty columns for any rows that do not have a match in the other table.

Pre-Oracle 9i syntax does not provide a full outer join. We need to use a work around that performs the *UNION* of left and right outer joins. Oracle 9i allows for ANSI compatible syntax for full outer joins.

(Pre-Oracle 9i Syntax for Outer Join)

To perform outer join, we have to place a plus sign (+) enclosed by parenthesis after the name of the *WHERE* clause join condition argument for the table that is deficit in rows (i.e. that has the missing rows). How a join operator is used to perform left outer join, right outer join and full outer join (using *UNION*) is discussed below in details.

Oracle 9i or higher syntax for outer join

The outer join is specified in the *FROM* clause. It has the following general format:

```
<table_name> { LEFT | RIGHT | FULL } OUTER JOIN <table_name>  
ON <predicate>
```

Predicate is a join predicate for the outer join. It can only reference columns from the joined tables. The *LEFT*, *RIGHT* or *FULL* specifiers give the type of join:

- *LEFT*—only unmatched rows from the left side table are to be retained.
- *RIGHT*—only unmatched rows from the right side table are to be retained.
- *FULL*—unmatched rows from both tables are to be retained.

Left Outer Join

A left outer join is a join where records from the left table that have no matching key in the right table are included in the result set.

Query: List the employee codes, names and project ids assigned to him/her of all employees of the *EMP* table

In Pre-Oracle 9i

The left table for this discussion is the table on the left side of the join condition and the right table is the table on the right of the join condition. The (+) outer join

NOTES

operator is placed to the right of the right table to signify that all columns returned from the right table corresponding to non-matching records from the left table are returned as NULL values.

```
SELECT E.ecode,ename,pid FROM emp E,assign A  
WHERE E.ecode=A.ecode (+);
```

Output:

ECODE	ENAME	PID
E01	KOUSHIK GHOSH	P01
E01	KOUSHIK GHOSH	P02
E01	KOUSHIK GHOSH	P03
E02	JAYANTA DUTTA	P01
E02	JAYANTA DUTTA	P02
E03	HARI NANDAN TUNGA	P03
E04	JAYANTA GANGULY	
E05	RAJIB HALDER	
E06	JISHNU BANERJEE	P03
E07	RANI BOSE	P02
E08	GOUTAM DEY	P01
E08	GOUTAM DEY	P02
E09	PINAKI BOSE	P03

13 rows selected.

In Oracle 9i

To write a query that performs an outer join of tables A and B and returns all rows from A (a **left outer join**), the LEFT [OUTER] JOIN syntax in the FROM clause can be used. For all rows in A that have no matching rows in B, null is returned for any select list expressions containing columns of B.

```
SELECT E.ecode,ename,pid  
FROM emp E LEFT OUTER JOIN assign A  
ON A.ecode=E.ecode;
```

Output:

ECODE	ENAME	PID
E01	KOUSHIK GHOSH	P01
E01	KOUSHIK GHOSH	P02
E01	KOUSHIK GHOSH	P03
E02	JAYANTA DUTTA	P01
E08	GOUTAM DEY	P01
E02	JAYANTA DUTTA	P02
E08	GOUTAM DEY	P02
E07	RANI BOSE	P02

E03	HARI NANDAN TUNGA	P03
E09	PINAKI BOSE	P03
E06	JISHNU BANERJEE	P03
E05	RAJIB HALDER	
E04	JAYANTA GANGULY	

*Form of Basic SQL
Query*

NOTES

13 rows selected.

Notice that the PIDs for employees with ecode E04 and E05 are null as they have assigned no projects.

The above query can also be written through the USING clause provided the USING clause cannot have qualifier.

```
SELECT ecode, ename, pid
FROM emp emp LEFT OUTER JOIN assign USING (ecode);
```

Right Outer Join

A right outer join is a join where records from the right table that have no matching key in the left table are included in the result set. We can rewrite the above query using right outer join.

In Pre-Oracle 9i

The left table for this discussion is the table on the left side of the join condition and the right table is the table on the right of the join condition. The (+) outer join operator is placed to the right of the left table to signify that all columns returned from the left table corresponding to non-matching records in the right table are returned as NULL values.

```
SELECT E.ecode, ename, PID FROM emp E, assign A
WHERE A.ecode (+)=E.ecode;
```

In Oracle 9i

To write a query that performs an outer join of tables A and B and returns all rows from B (a **right outer join**), the RIGHT [OUTER] JOIN syntax in the FROM clause can be used. For all rows in B that have no matching rows in A, null is returned for any select list expressions containing columns of A.

```
SELECT E.ecode, ename, pid FROM assign A RIGHT OUTER JOIN emp E
ON A.ecode=E.ecode;
OR
SELECT ecode, ename, pid FROM assign RIGHT OUTER JOIN emp
USING (ecode);
```

Both of the queries give the same output given in the left outer join because left and right outer joins are similar; the difference is in the ordering of the tables in the FROM clause.

NOTES

Full Outer Join

A full outer join is a join where records from the first table are included that have no corresponding record in the other table and records from the other table are included that have no records in the first table.

To illustrate full outer join, DEPT table contains a department and there is no employee in that department, the tuple would not be selected in a query joining the DEPT and EMP tables.

We insert a row in the dept table as follows:

```
INSERT INTO dept VALUES ('D04', 'EDUCATION', 'BANGALORE');  
1 row created.
```

Furthermore, in emp table there is an employee to whom no dept has been assigned. For that we insert the following row:

```
INSERT INTO emp VALUES ('E10', 'RAJA SEN', 4500, NULL, 'JR  
PROGRAMMER', '12-APR-99');  
1 row created.
```

This row also will not appear in the inner join. Now if outer join is performed, result set will contain these newly added rows from both the table.

Query: List the information of all employees of all departments

In Pre-Oracle 9i

We can use the outer join on only one side of the join condition. Using the outer join on both sides will cause an error to be issued. Since Oracle will issue an error, then another way to get an outer join is to use UNION.

```
SELECT ecode, ename, D.dno, dname  
FROM emp E, dept D  
WHERE E.dno (+) = D.dno  
UNION  
SELECT ecode, ename, E.dno, dname  
FROM emp E, dept D  
WHERE E.dno=D.dno (+);
```

Output:

ECODE	ENAME	DNO	DNAME
E01	KOUSHIK GHOSH	D01	PROJECT
E02	JAYANTA DUTTA	D01	PROJECT
E03	HARI NANDAN TUNGA	D02	RESEARCH
E04	JAYANTA GANGULY	D03	PERSONNEL
E05	RAJIB HALDER	D03	PERSONNEL
E06	JISHNU BANERJEE	D02	RESEARCH
E07	RANI BOSE	D01	PROJECT

E08	GOUTAM DEY	D01	PROJECT
E09	PINAKI BOSE	D02	RESEARCH
E10	RAJA SEN	D04	EDUCATION

*Form of Basic SQL
Query*

11 rows selected.

Notice the last two rows that are in bold font. The first query gets the employees of all departments including department ‘D04’ in which no employee has been assigned. The second query gets all the employees including employee ‘E10’ who has not been assigned to any department. In the case of the first query, any dept without employee would return `NULL` values for all two columns, ecode and ename. In the case of the second query, any employee without dno would return `NULL` values for two columns, dno and dname. The `UNION` puts both together and eliminates any repetitions.

It is to be noted that since ‘dno’ is a foreign key in the `EMP` table and a primary key in the `DEPT` table, the combination of entity integrity and referential integrity tells us that only one of the queries is necessary.

In Oracle 9i

In Oracle 9*i*, we can use the `FULL OUTER JOIN` keyword to perform the same thing:

```
SELECT ecode, ename, D.dno, dname
  FROM emp E FULL OUTER JOIN dept D
    ON E.dno=D.dno;
```

It will give the same output as above.

Self-Join in SQL—Joining Tables to Itself

A **self-join** is a join of a table to itself. This table appears twice in the `FROM` clause and is followed by table aliases that qualify column names in the join condition.

Query: List the code and name of all the employees who work in the same department as employee code ‘E03’

In this case, the two ‘versions’ of the `emp` table must be used, one for employees other than E03, and one for employee E03:

Using the WHERE clause	Using the ON clause
<pre>SELECT x.ecode, x.ename FROM emp x, emp y WHERE x.dno = y.dno AND y.ecode = 'E03' AND x.ecode != 'E03';</pre>	<pre>SELECT x.ecode, x.ename FROM emp x JOIN emp y ON x.dno = y.dno AND y.ecode = 'E03' AND x.ecode != 'E03'</pre>

NOTES

NOTES

Output:

ECODE	ENAME
E06	JISHNU BANERJEE
E09	PINAKI BOSE

One version of the table **emp** is needed so that we can find the department number of employee E03. In the above example, this table is called **y**. We then look through another version of the table emp, here called **x**, to find people who are in the same department. Finally, we do not want employee code E03 to be displayed, so we should eliminate this case by adding `x.empno != 'E03'` in the query.

Query: Find out the employees who have exactly the same salary

Using the WHERE clause	Using the ON clause
<pre>SELECT a.ecode, a.salary FROM emp a, emp b WHERE a.salary=b.salary AND a.ecode<>b.ecode;</pre>	<pre>SELECT a.ecode, a.salary FROM emp a JOIN emp b ON a.salary=b.salary AND a.ecode<>b.ecode;</pre>

Output:

ECODE	SALARY
E05	4000
E03	4000
E08	5000
E01	5000

The second condition is important. If the above query is written omitting the second condition, it will give the following output:

ECODE	SALARY
E07	3000
E02	3500
E03	4000
E05	4000
E03	4000
E05	4000
E01	5000
E08	5000
E01	5000
E08	5000
E09	5500
E04	6000
E06	6500

13 rows selected.

NOTES

If the condition is omitted, then it yields duplicate tuples. As the tables are same, obviously salary of any employee must be the same with himself, and then those tuples will appear in the output. The second condition (`a.ecode<>b.ecode`) will remove those tuples.

Query: Find out the employees in each department getting exactly the same salary

Using the WHERE clause	Using the ON clause
<pre>SELECT a.ecode, a.ename, a.salary, a.dno FROM emp a , emp b WHERE a.salary=b.salary AND a.dno=b.dno AND a.ecode<>b.ecode;</pre>	<pre>SELECT a.ecode, a.ename, a.salary, a.dno FROM emp a JOIN emp b ON a.salary=b.salary AND a.dno=b.dno AND a.ecode<>b.ecode;</pre>

Output:

ECODE	ENAME	SALARY	DNO
E08	GOUTAM DEY	5000	D01
E01	KOUSHIK GHOSH	5000	D01

Query: Select the pair of employees for each department to whom projects may be assigned

Using the WHERE clause	Using the ON clause
<pre>SELECT a.ecode, b.ecode, a.dno FROM emp a, emp b WHERE a.dno=b.dno AND a.ecode>b.ecode ORDER BY a.dno;</pre>	<pre>SELECT a.ecode, b.ecode, a.dno FROM emp a JOIN emp b ON a.dno=b.dno AND a.ecode>b.ecode ORDER BY a.dno</pre>

Output:

ECODE	ECODE	DNO
E02	E01	D01
E07	E01	D01
E08	E01	D01
E07	E02	D01
E08	E02	D01
E08	E07	D01
E06	E03	D02
E09	E03	D02
E09	E06	D02
E05	E04	D03

10 rows selected.

The last condition $a.ecode > b.ecode$ may also be written as $a.ecode < b.ecode$.

Let us illustrate the above query in more details. If the query is written omitting the second condition, it will give the following output:

NOTES

E CODE	E CODE	D NO
E01	E01	D01
E02	E01	D01
E08	E01	D01
E07	E01	D01
E01	E02	D01
E02	E02	D01
E08	E02	D01
E07	E02	D01
E01	E08	D01
E02	E08	D01
E08	E08	D01
E07	E08	D01
E01	E07	D01
E02	E07	D01
E08	E07	D01
E07	E07	D01
E03	E03	D02
E09	E03	D02
E06	E03	D02
E03	E09	D02
E09	E09	D02
E06	E09	D02
E03	E06	D02
E09	E06	D02
E06	E06	D02
E04	E04	D03
E05	E04	D03
E04	E05	D03
E05	E05	D03

29 rows selected.

The output contains some rows implying logically the same combinations, say E01 E02 D01 and E02 E01 D01. Some rows are meaningless as codes are combined with themselves like E01 E01 D01.

If the query is written as follows:

Then the output will be:

Using the WHERE clause	Using the ON clause
SELECT a.ecode, b.ecode, b.dno FROM emp a, emp b WHERE a.dno=b.dno AND a.ecode<>b.ecode;	SELECT a.ecode, b.ecode, b.dno FROM emp a JOIN emp b ON a.dno=b.dno AND a.ecode<>b.ecode;

NOTES

ECODE	ECODE	DNO
E02	E01	D01
E08	E01	D01
E07	E01	D01
E01	E02	D01
E08	E02	D01
E07	E02	D01
E01	E08	D01
E02	E08	D01
E07	E08	D01
E01	E07	D01
E02	E07	D01
E08	E07	D01
E09	E03	D02
E06	E03	D02
E03	E09	D02
E06	E09	D02
E03	E06	D02
E09	E06	D02
E05	E04	D03
E04	E05	D03

20 rows selected.

The rows with the same ecode are eliminated. But 10 rows with bold fonts are extraneous because their equivalent rows are already present in the output. To eliminate one of the pair, the condition

a.ecode>b.ecode or
a.ecode<b.ecode
has to be specified

Similarly,

NOTES

Query: Get all pairs of employees such that they are working at the same department

Using the WHERE clause	Using the ON clause
<pre>SELECT a.ecode, b.ecode, a.dno FROM emp a, emp b WHERE a.dno=b.dno AND a.ecode>b.ecode ORDER BY a.dno</pre>	<pre>SELECT a.ecode, b.ecode, a.dno FROM emp a JOIN emp b ON a.dno=b.dno AND a.ecode>b.ecode ORDER BY a.dno</pre>

Output:

ECODE	ECODE	DNO
E02	E01	D01
E07	E01	D01
E08	E01	D01
E07	E02	D01
E08	E02	D01
E08	E07	D01
E06	E03	D02
E09	E03	D02
E09	E06	D02
E05	E04	D03

10 rows selected.

Query: Which salary is common to more than one department?

Using the WHERE clause	Using the ON clause
<pre>SELECT a.dno,a.salary FROM emp a, emp b WHERE a.salary=b.salary AND a.dno<>b.dno;</pre>	<pre>SELECT a.dno,a.salary FROM emp a JOIN emp b ON a.salary=b.salary AND a.dno<>b.dno;</pre>

Output:

DNO	SALARY
D03	4000
D02	4000

NOTES

Query: Which salary is common to more than one employee in each department?

Using the WHERE clause	Using the ON clause
<pre>SELECT a.dno,a.salary FROM emp a ,emp b WHERE a.salary=b.salary AND a.dno=b.dno AND a.ecode<>b.ecode;</pre>	<pre>SELECT a.dno,a.salary FROM emp a JOIN emp b ON a.salary=b.salary AND a.dno=b.dno AND a.ecode<>b.ecode;</pre>

Output:

DNO	SALARY
D01	5000
D01	5000

7.2.7 Nested Query/Subquery in SQL

To illustrate the SUBQUERY, we consider the instances of the following tables:

EMP Table:

E CODE	E NAME	SALARY	D NO	D E S G	D T _ J N
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93
E02	JAYANTA DUTTA	3500	D01	PROGRAMMER	15-JAN-94
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E07	RANI BOSE	3000	D01	PROJECT ASSISTANT	17-JUN-97
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

DEPT Table:

DNO	DNAME	CITY
D01	PROJECT	KOLKATA
D02	RESEARCH	CHENNAI
D03	PERSONNEL	KOLKATA

PROJECT Table:

P ID	P NAME	L O C A T I O N
P01	HOSPITAL MANAGEMENT SYSTEM	KOLKATA
P02	ACCOUNTING SYSTEM	KOLKATA
P03	BANKING INFORMATION SYSTEM	CHENNAI

ASSIGN Table:

NOTES

ECODE	PID
E01	P01
E01	P02
E01	P03
E02	P01
E08	P01
E02	P02
E08	P02
E07	P02
E03	P03
E09	P03
E06	P03

So far we have worked with queries that have either a comparison statement or a compound comparison statement in the WHERE clause. We can have a query in the WHERE clause of a SELECT statement. The query in the WHERE clause is called a subquery. The subquery is often referred to as the inner query and the surrounding query is called the outer query.

These subqueries may be written into the following two forms:

- Non-correlated
- Correlated

In a *non-correlated subquery*, the first inner query is executed first. The outer query takes an action based on the results of the inner query. In a *correlated subquery*, the inner query needs values from the outer query and passes results to the outer query. It is important to know that for a simple (or correlated subquery), the subquery is executed only once.

Let us illustrate the above forms with an example:

Query: Find the employees who are working in department located at KOLKATA

Using non-correlated subquery, the SQL statement will be as follows:

OUTER QUERY

```
SELECT ecode, ename, dno FROM emp
WHERE dno IN (SELECT dno FROM dept WHERE city='KOLKATA');
```

INNER QUERY

The inner query is independent and gets executed first, passing results to the outer query. Using correlated subquery, the above one can be rewritten as follows:

```
SELECT ecode, ename, dno
FROM emp outer
WHERE dno IN (SELECT dno FROM dept WHERE city = 'KOLKATA'
    AND dno = outer.dno);
```

Output:

ECODE	ENAME	DNO
E01	KOUSHIK GHOSH	D01
E02	JAYANTA DUTTA	D01
E04	JAYANTA GANGULY	D03
E05	RAJIB HALDER	D03
E07	RANI BOSE	D01
E08	GOUTAM DEY	D01

6 rows selected.

Correlated subqueries operate in a repetitive manner. Those familiar with looping control structures of a programming language will see the similarity. In a correlated subquery, the outer query runs producing one test record at a time. For each record that the outer query produces, the inner query is supplied one or more columns from the outer query. The inner query runs when it receives column(s) from the outer query and produces a result set. After the inner query produces a result set, the outer query evaluates the comparison statements containing the inner query. If the comparison statements evaluate favourably, the record produced by the outer query is returned and displayed. This process continues until the outer query tests each test record. Note that a test record is a record that is produced by the outer query by relaxing the conditions containing the inner query.

The full set of *test* records for this correlated subquery produced are listed below by running the relaxed query. The relaxed query is the original outer query with the condition imposed by the inner query having been removed (remove the inner query and its condition to get the relaxed query).

Relaxed Query

```
SELECT ecode, ename, dno FROM emp;
```

Test records

ECODE	ENAME	DNO
E01	KOUSHIK GHOSH	D01
E02	JAYANTA DUTTA	D01
E03	HARI NANDAN TUNGA	D02
E04	JAYANTA GANGULY	D03
E05	RAJIB HALDER	D03

NOTES

NOTES

E06	JISHNU BANERJEE	D02
E07	RANI BOSE	D01
E08	GOUTAM DEY	D01
E09	PINAKI BOSE	D02

9 rows selected.

HOW IT WORKS: Steps involved.

Step 1: First the outer query runs and supplies a column from the first test record to the inner query

The first record is

ECODE	NAME	DNO
E01	KOUSHIK GHOSH	D01

Column outer.dno = 'D01' is supplied to inner query.

Step 2: Then the outer query shares the outer.dno column value of 'D01' with the inner query and the inner query runs. Let us see the result of the inner query.

SELECT dno FROM dept WHERE city = 'KOLKATA' AND dno = 'D01';	Column outer.dno is supplied to the inner query. The inner query runs and produces the result set on the right.	DNO D01
---	---	------------

Step 3: The inner query can now provide a value to the outer query's WHERE clause. Now the outer query can evaluate the comparison condition in the WHERE clause. Notice that we have to simulate the current test record by adding the first and last name conditions to *freeze* the outer query. Let us run it and see what we get.

SELECT ecode, ename, dno FROM emp outer WHERE dno = 'D01';	Inner query return value replaces the subquery in the outer query evaluation.
--	---

This test record satisfies the WHERE clause condition and is returned

ECODE	ENAME	DNO
E01	KOUSHIK GHOSH	D01

Step 4: The outer query has found a match so E01's information is returned.

First returned record

ECODE	ENAME	DNO
E01	KOUSHIK GHOSH	D01

Continue Step 1 through Step 4 for each record in the test record set.

NOTES

As we have seen, a simple subquery is executed once prior to the execution of the outer query. With a correlated subquery, the inner query is executed once for each record that is returned by the outer query. The inner query is driven by a correlation between the outer query and the inner query. The correlation provided by an exchange of column(s) from the outer query to the inner query. A table alias is used to share the outer query column(s) with the inner query.

The general form of subquery is as follows:

```
SELECT [DISTINCT] <column_list>
      FROM <table_list>
      WHERE <column_name> | <expression> {[NOT] IN | 
          <comparison_operator>
          [ANY | ALL] | [NOT] EXISTS}
          (SELECT [DISTINCT] <column_list>
              FROM <table_list>
              WHERE <condition>)
              GROUP BY <group_by_list>
              HAVING <condition>
              ORDER BY <order_by_list>;
```

Types of Subqueries

SQL has the following three basic types of subqueries:

- **Predicate Subquery:** These are extended logical constructs in the WHERE (and HAVING) clause.
- **Scalar Subquery:** These are standalone queries that return a single value; they can be used anywhere a scalar value is used.
- **Table Subquery:** These queries are nested in the FROM clause.

All subqueries should be enclosed in parentheses.

Predicate Subqueries

Predicate subqueries are used in the WHERE (and HAVING) clause. Predicate subqueries may be classified into the following categories:

- Single value subquery (introduced with relational operator)
- Multiple value subquery (introduced with IN)
- Quantified Subqueries (introduced with comparison operator accompanied by ANY/ALL)
- Subqueries that are an existence test (introduced with EXISTS)
- Single value subquery (introduced with relational operator)

NOTES

Subqueries returning single value from single column

The following generic SELECT statement depicts the syntax for a simple subquery returning a single value:

```
SELECT <column_list>
      FROM <table_list>
     WHERE COLUMN <relational_operator> (SELECT statement);
```

Illustration 7.3:

(NON-CORRELATED SUBQUERY)

Query: List the employees working in the personnel department

```
SELECT * FROM emp WHERE dno=(SELECT dno FROM dept WHERE
          dname='PERSONNEL');
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E04	JAYANTAGANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95

The above query may also be written as follows:

```
SELECT * FROM emp WHERE 'PERSONNEL'=(SELECT dname FROM dept
          WHERE dept.dno=emp.dno);
```

Query: Display the information of the employee(s) getting the highest salary

```
SELECT * FROM emp WHERE salary = (SELECT MAX(salary) FROM emp);
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96

Query: Retrieve the names and salaries for all employees who earn more than the average salary

```
SELECT ENAME, SALARY FROM EMP
      WHERE SALARY > (SELECT AVG(SALARY) FROM EMP);
```

Output:

ENAME	SALARY
KOUSHIK GHOSH	5000
JAYANTAGANGULY	6000
JISHNU BANERJEE	6500
GOUTAM DEY	5000
PINAKI BOSE	5500

Non-Correlated Subquery in HAVING—Selecting ONLY the group with the maximum Sum in a group query

Form of Basic SQL Query

Query: Find a designation with the lowest average salary

```
SELECT desg, AVG (salary)
      FROM emp
      GROUP BY desg
      HAVING AVG (salary) =
        (SELECT MIN (AVG (salary)) )
      FROM emp
      GROUP BY desg) ;
```

NOTES

Output:

DESG	AVG(SALARY)
PROJECT ASSISTANT	3000

Query: Display the name of the department and sum of salaries of all employees of that department, which is spending maximum sum of salaries

```
SELECT a.dname, sum(b.salary)
      FROM dept a, emp b
      WHERE a.dno = b.dno
      GROUP BY a.dname
      HAVING SUM(b.salary) = (SELECT MAX (SUM(c.salary))
      FROM emp c GROUP BY c.dno) ;
```

Output:

DNAME	SUM(B.SALARY)
PROJECT	16500

Correlated Subquery

Query: List the employees who get the same salary as KOUSHIK GHOSH

```
SELECT * FROM emp A WHERE salary = (SELECT salary FROM emp B
      where A.ecode=>B.ecode AND b.ename='KOUSHIK GHOSH') ;
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97

Query: List the details of the employees getting the highest salary in each department

```
SELECT * FROM emp A WHERE salary= (SELECT MAX (salary) FROM emp
      B
      WHERE A.dno=B.dno) ORDER BY dno;
```

NOTES

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10 - MAR - 93
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96

Query: Retrieve the details of department having more than 2 employees

```
SELECT * FROM dept
WHERE 2 < (SELECT COUNT(*) from emp WHERE dept.dno=emp.dno);
```

OR

```
SELECT * FROM dept
WHERE (SELECT COUNT(*) from emp WHERE dept.dno=emp.dno) > 2;
```

Output:

DNO	DNAME	CITY
D01	PROJECT	KOLKATA
D02	RESEARCH	CHENNAI

Listing a certain number of records with the highest values for a certain column using SQL only

There are times where it is needed to simply return the rows with a certain number of the highest (or lowest) values for a certain column. For example,

Query: Display the code and salary of the first three employees getting the highest salaries

One solution for this problem is to use a correlated subquery to the same table. The following select will return the correct rows:

```
SELECT ecode, salary FROM emp e1
WHERE 3 > (SELECT COUNT(*) FROM emp e2 WHERE e1.salary < e2.salary)
ORDER BY salary desc;
```

Output:

ECODE	SALARY
E06	6500
E04	6000
E09	5500

For every row processed by the main query, the correlated subquery returns a count (*COUNT(*)*) of the number of rows with higher salaries (*WHERE e1.salary < e2.salary*). Then the main query only returns rows that have fewer than three salaries that are higher (*WHERE 3 > ...*). For example, for ECODE = E09, the salary is '5500'. There is only 2 rows with a higher salary (ECODE = E06, E04),

so the subquery returns '2', which is less than 3, causing the 'WHERE 3 > ...' to evaluate to TRUE, thereby returning the row.

- Retrieving nth maximum salary from EMP Table

Query: Retrieve the third highest salary of the EMP table

```
SELECT DISTINCT (e1.salary) FROM emp e1 WHERE 3 = (SELECT COUNT
(DISTINCT (e2.salary)) FROM emp e2 WHERE
e1.salary<=e2.salary);
```

Output:

SALARY

5500

Query: Display the details of the employee(s) getting the third highest salary

```
SELECT * FROM emp WHERE salary=( SELECT DISTINCT (e1.salary)
FROM emp e1 WHERE 3 = (SELECT COUNT (DISTINCT (e2.salary))
FROM emp e2 WHERE e1.salary<=e2.salary));
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

Subqueries returning single value from each of the multiple columns (Oracle 9i feature)

It is possible for a subquery to return more than one column for each row that it returns. In the case that a simple subquery returns multiple columns a comparison can be made by creating a list in comparing the return values of the subquery and the columns to be compared. The following is the general syntax for a subquery returning multiple columns:

```
SELECT <column_list>
FROM <table_list>
WHERE (column_name1, ..., column_nameK) = (SELECT ....);
```

Illustration 7.4:

Query: List the details of the employees getting the highest salary in each department

```
select * from emp
where (dno, salary) in (select dno, max(salary) from emp
group by dno);
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96

Correlated subquery in HAVING

NOTES

NOTES

Query: Select department wise total salary, eliminating all those departments in which total salary was not at least Rs 11000 more than the maximum salary for the department

```
SELECT dno, SUM(salary) FROM emp A GROUP BY dno  
HAVING SUM(salary) > (SELECT 11000+MAX(salary) FROM emp B  
WHERE A.dno=B.dno);
```

Output:

DNO	SUM(SALARY)
D01	16500

Multiple value subquery (introduced with IN)

A subquery can return multiple values. In that case relational operator cannot be used for comparison. Multi-row subqueries require a multi-value comparison operator such as IN. The IN subquery is used for testing whether a scalar value matches the single query column value in any subquery result row. The general form of such a query is as follows:

```
SELECT <column list>  
FROM <table_list>  
WHERE <column_name> [NOT] IN (SELECT statement);
```

Note: The comparison operator can use IN or NOT IN because the inner query can return a list. Single value comparison operators such as >, <, <=, >= and = are not allowed.

A. Subqueries returning multiple values from single column

NON-CORRELATED SUBQUERY

Query: List the departments having more than 2 employees working in that department

```
SELECT * FROM dept WHERE dno IN (SELECT dno FROM emp GROUP BY  
dno HAVING COUNT(*)>2);
```

Output:

DNO	DNAME	CITY
D01	PROJECT	KOLKATA
D02	RESEARCH	CHENNAI

Query: Retrieve employee details who are not assigned in any project

```
SELECT * FROM emp  
WHERE ecode NOT IN (SELECT DISTINCT ecode FROM assign);
```

Output:

E CODE	E NAME	SALARY	D NO	DESG	DT_JN
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95

Query: Retrieve department details in which no PROGRAMMER is working

Form of Basic SQL Query

```
SELECT * FROM DEPT  
WHERE dno NOT IN (SELECT DISTINCT dno FROM emp WHERE  
desg='PROGRAMMER') ;
```

Output:

DNO	DNAME	CITY
D03	PERSONNEL	KOLKATA

CORRELATED SUBQUERY

Query: Which designations are common to more than one department?

```
SELECT DISTINCT A.desg  
FROM emp A WHERE A.desg IN (SELECT B.desg  
FROM emp B WHERE A.dno <> B.dno) ;
```

Output:

DESG

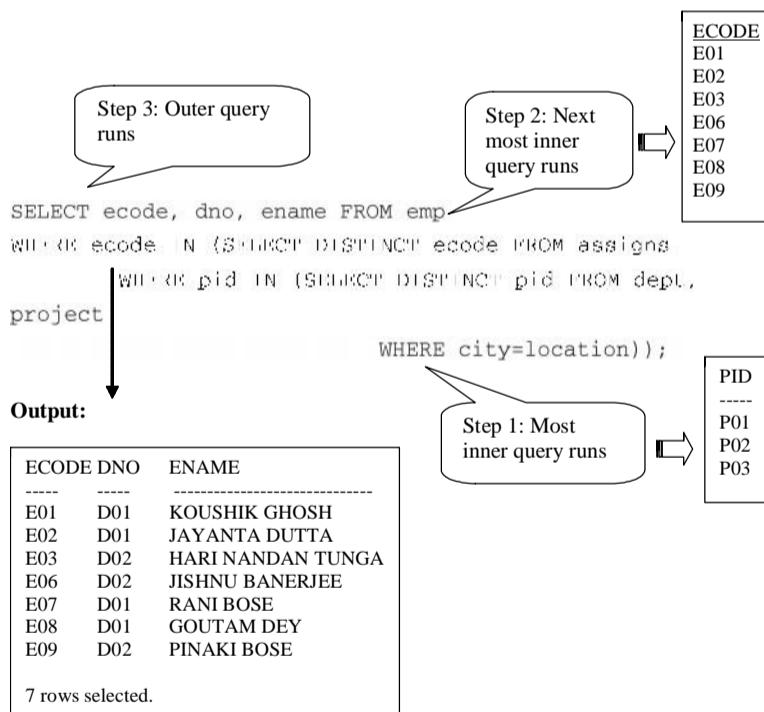
PROGRAMMER

NESTED SUBQUERY

It is possible to nest subqueries but it is not a practice that makes sense beyond three or four levels deep. The following example demonstrates a nested subquery. Both subqueries are multi-row subqueries.

Query: Retrieve the code, department number and name of each employee who are working in the project located in the same city as their departments

NOTES



NOTES

Query: List the department details having maximum number of employees

```
SELECT * FROM dept
WHERE dno IN (SELECT dno FROM emp GROUP BY dno
HAVING COUNT(*) = (SELECT MAX(COUNT(*)) FROM emp
GROUP BY dno));
```

Output:

DNO	DNAME	CITY
D01	PROJECT	KOLKATA

Query: List the employees who are working on all the projects

```
SELECT * FROM emp WHERE ecode IN ( SELECT ecode FROM assign
GROUP BY ecode HAVING COUNT(ecode) = (SELECT COUNT(pid) FROM
project));
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93

B. Subqueries returning multiple values from the multiple columns (Oracle 9i feature)
Multi-row subquery syntax for multiple columns is as follows:

```
SELECT <column_list>
FROM <table_list>
WHERE (<column_name1>, ..., <column_nameK>) IN (
SELECT ....);
```

Query: Retrieve the code, name and department number of the employees who are working in the project in the same city as their department

```
SELECT DISTINCT E.ecode, ename, E.dno
FROM emp E, dept D, assigns A
WHERE E.dno=D.dno
AND E.ecode=A.ecode
AND (pid,city) IN (SELECT pid,location FROM project);
```

Output:

ECODE	ENAME	DNO
E01	KOUSHIK GHOSH	D01
E02	JAYANTA DUTTA	D01
E03	HARI NANDAN TUNGA	D02
E06	JISHNU BANERJEE	D02
E07	RANI BOSE	D01
E08	GOUTAM DEY	D01
E09	PINAKI BOSE	D02

7 rows selected.

If we use the equality operator(=) instead of the IN operator in the third predicate, then Oracle generates an error as inner subquery returns multiple rows.

```
SELECT DISTINCT E.ecode, ename, E.dno
  FROM emp E, dept D, assigns A
 WHERE E.dno=D.dno
   AND E.ecode=A.ecode
  AND (pid,city) = (SELECT pid, location FROM project)
```

Output:

```
AND (pid,city) = (SELECT pid, location FROM project)
*
ERROR at line 5:
ORA-01427: single-row subquery returns more than one row
```

NOTES

• Quantified Subqueries (introduced with comparison operator accompanied by ANY/ALL)

Subqueries introduced with the comparison operator and ANY/ALL. ALL implies that all the values results from the inner query must be taken into account. ANY, SOME and ALL are multi-valued comparison operators that allow the use of single valued comparison operators such as >, <, <=, >= and = on a list of values. These comparison operators can be used in any multi-row subquery.

Syntax

```
SELECT <select_list>
  WHERE <expression> <relational_operator> [ANY | ALL] (subquery);
```

ALL

ALL works with a comparison operator and a list. The ALL comparison operator is true whenever all the values, produced by the subquery, compares favourably with the column being compared.

>ALL means greater than all the values these return from the subquery, i.e. greater than the largest value among the values returns from the inner query.

>ALL (1, 2, 3) implies >3

Similarly <ALL (1, 2, 3) implies <1

=ALL(1, 2, 3) means =1 AND =2 AND =3, which is meaningless.

ALL is used basically with in equalities rather than equalities because a value cannot be 'equal to all' the values. A value can be equal to all of the results of a subquery if all the said results are infact identical.

Query: Find the employee getting more salary than every employee of the 'PROJECT' department

```
SELECT * FROM emp
```

```
WHERE salary > ALL (SELECT salary FROM emp
WHERE dno = (SELECT dno FROM dept WHERE dname = 'PROJECT')) ;
```

Output:

NOTES

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E04	AYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

Query: Find the details of the employee(s) who is/are getting the highest salary

It can be written as using MAX() function

```
SELECT * FROM emp WHERE salary = (SELECT MAX (salary) FROM
emp) ;
```

Using ALL it can be rewritten as

```
SELECT * FROM emp
WHERE sal >= ALL (SELECT salary FROM emp) ;
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96

Query: List the department details having the maximum number of employees

```
SELECT * FROM dept
WHERE dno IN (SELECT dno FROM emp GROUP BY dno
HAVING COUNT (*) >= ALL (SELECT COUNT (*) FROM emp
GROUP BY dno)) ;
```

Output:

DNO	DNAME CITY
D01	PROJECT KOLKATA

It is to be noted that if the inner query introduced with ALL and a comparison operator, returns NULL as one of its values, the entire query fails.

ANY (SOME is the same as ANY)

ANY works with a comparison operator and a list. The ANY comparison operator is true whenever one or more of the list elements compares favourably with the column being compared.

>ANY means that greater than at least one value, i.e. greater than the minimum.

>ANY (1, 2, 3) is equivalent to >1

<ANY means that less than the maximum, i.e. <ANY(1,2,3) implies <3
 =ANY(1,2,3) means =1 OR =2 OR =3
 <> ANY (1, 2, 3) means NOT 1 OR NOT 2 OR NOT 3

Form of Basic SQL Query

The following equivalences are listed:

- >=ALL (LIST) is equivalent to = MAX(LIST)
- <=ALL (LIST) is equivalent to = MIN (LIST)
- <> ANY is equivalent to = NOTIN (LIST)
- =ANY is equivalent to = IN (LIST)

NOTES

Query: Find the employees getting salary larger than the minimum salary of the PERSONNEL department

It can be written as follows using MIN() group function:

```
SELECT * FROM emp
WHERE salary> (SELECT MIN(salary) FROM emp
WHERE dno = (SELECT dno FROM dept WHERE dname='PERSONNEL' )) ;
Using ANY query will be
SELECT * FROM emp
WHERE salary>ANY (SELECT salary FROM emp , dept
WHERE emp .dno = dept .dno AND dname = 'PERSONNEL' ) ;
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

Query: Find the employees getting same salary as the salary of any employee of the PERSONNEL department

```
SELECT * FROM emp
WHERE salary=ANY (SELECT salary FROM emp , dept
WHERE emp .dno = dept .dno AND dname = 'PERSONNEL' ) ;
OR
SELECT * FROM emp
WHERE salary=ANY (SELECT salary FROM emp
WHERE dno = (SELECT dno FROM dept WHERE dname = 'PERSONNEL' ))
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96

NOTES

“=ANY” is exactly equivalent to IN. The above query can be replaced as follows:

```
SELECT * FROM emp
WHERE salary IN (SELECT salary FROM emp, dept
WHERE emp.dno = dept.dno AND dname='PERSONNEL');
```

Notice that the records of the PERSONNEL department (with dno='D03') appear in the output. These records should be removed from the result set. Therefore, the query will be as follows:

```
SELECT * FROM emp
WHERE salary=ANY (SELECT salary FROM emp, dept
WHERE emp.dno = dept.dno AND dname='PERSONNEL')
AND dno <> (SELECT dno FROM dept WHERE dname='PERSONNEL');
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95

It is very important to note that “<> ANY” is not equivalent to NOT IN. Consider the following example:

Query: Find the employees who do not work in any project

Look at the ASSIGN table. The employee with codes ‘E04’ and ‘E05’ are absent in the ASSIGN table. Obviously their records will be the output. However, the following query will give all the rows of the EMP table:

```
SELECT * FROM emp
WHERE ecode <>ANY (SELECT DISTINCT ecode FROM assign);
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93
E02	JAYANTA DUTTA	3500	D01	PROGRAMMER	15-JAN-94
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E07	RANI BOSE	3000	D01	PROJECT ASSISTANT	17-JUN-97
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

Because <> ANY (1, 2, 3) means NOT 1 OR NOT 2 OR NOT 3, whereas NOT IN (1, 2, 3) implies NOT 1 AND NOT 2 AND NOT 3. Therefore, the query should be as follows:

```
SELECT * FROM emp
WHERE ecode NOT IN (SELECT DISTINCT ecode FROM assign);
```

NOTES

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95

Comparison of ALL and ANY

Whenever a legal subquery fails to produce output, ALL is automatically TRUE but ANY is automatically FALSE.

Illustration 7.5:

Look at the DEPT table; there is no department at BANGALORE.

Query: Find the employees getting salary larger than the minimum salary of the department located at BANGALORE

The following query would result no output:

```
SELECT * FROM emp
WHERE salary > ANY (SELECT salary FROM emp, dept
WHERE emp.dno = dept.dno AND city= 'BANGALORE' );
```

Output:

no rows selected

But the following query would produce the entire emp table:

```
SELECT * FROM emp
WHERE salary > ALL (SELECT salary FROM emp, dept
WHERE emp.dno = dept.dno AND city= 'BANGALORE' )
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93
E02	JAYANTA DUTTA	3500	D01	PROGRAMMER	15-JAN-94
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96
E07	RANI BOSE	3000	D01	PROJECT ASSISTANT	17-JUN-97
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94

• Subqueries that are an existence test (introduced with EXISTS)

The keywords EXISTS and NOT EXISTS are designed for use only with subqueries. They are equivalent to the existential quantifier (and its negated form) that is seen in tuple calculus. Both versions return either true or false only. The EXISTS keyword is a WHERE clause tests for the existence or non-existence of data that meets the criteria of the subquery.

NOTES

Existence or non-existence implies presence or absence of the ‘empty set’ of rows. Even if the subquery returns at least one row, the subquery evaluates to TRUE. The EXISTS operator is used with correlated subqueries. The general format is as follows:

```
SELECT <select_list>
WHERE [NOT] EXISTS (subquery);
```

Any valid EXISTS subquery must contain an *outer reference* (the subquery where clause references a column in the outer query), which means it must be a *correlated subquery*.

Rules with EXISTS

- EXISTS is not preceded by a column name, constant or other expression.
- The SELECT list of the subquery introduced by EXISTS almost always contains an asterisk (*) OR ‘X’. Because we are testing for the existence of rows that meets the subquery conditions.

Query: Find the names of the employees who work in the RESEARCH department

```
SELECT DISTINCT    ename   FROM   emp
WHERE EXISTS
  (SELECT * FROM dept
  WHERE dept.dno = emp.dno AND      dname = 'RESEARCH' );
```

Output:

```
ENAME
HARI NANDAN TUNGA
JISHNU BANERJEE
PINAKI BOSE
```

Query: Which designations are common to more than one department?

```
SELECT DISTINCT desg FROM emp A
WHERE EXISTS ( SELECT * FROM emp B
  WHERE A.desg=B.desg
  AND A.dno<>B.dno) ;
```

Output:

```
DESG
PROGRAMMER
```

Query: Show all employees names and salaries where there is at least one employee who makes more salary and at least one employee who makes less salary

```
SELECT ecode, ename, salary
FROM emp
WHERE EXISTS
  (SELECT ename
```

```

FROM emp e2
WHERE e2.salary > emp.salary)
AND EXISTS
(SELECT ename
FROM emp e3
WHERE e3.salary < emp.salary);

```

*Form of Basic SQL
Query*

Output:

ECODE	ENAME	SALARY
E01	KOUSHIK GHOSH	5000
E02	JAYANTA DUTTA	3500
E03	HARI NANDAN TUNGA	4000
E04	JAYANTA GANGULY	6000
E05	RAJIB HALDER	4000
E08	GOUTAM DEY	5000
E09	PINAKI BOSE	5500

7 rows selected.

NOT EXISTS acts as the reverse of EXISTS.

Query: Retrieve employee details who are not assigned in any project

```

SELECT * FROM emp
WHERE NOT EXISTS (SELECT * FROM assign WHERE
emp.ecode=assign.ecode);
OR
SELECT * FROM emp
WHERE NOT EXISTS (SELECT 'X' FROM assign WHERE
emp.ecode=assign.ecode);

```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E04	JAYANTAGANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95

Query: Show all employees for whom there does not exist an employee who is paid less; in other words, the highest paid employee

It can be easily written using simple subquery as follows:

```

SELECT *
FROM emp
WHERE salary=(SELECT MAX(salary) from emp);
But using NOT EXISTS it can be written as follows:
SELECT *
FROM emp
WHERE NOT EXISTS
(SELECT *

```

NOTES

*Form of Basic SQL
Query*

```
FROM emp e2
WHERE e2.salary > emp.salary);
```

Output:

NOTES

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96

Scalar Subqueries

A scalar subquery can appear as a scalar value in the select list and in the WHERE predicate of another query.

For a scalar subquery,

- The subquery should reference just one column in the select list.
- It should also retrieve no more than one row. If the subquery retrieves more than one row, a *run-time* error is generated and query execution is aborted.

Query: List the code, name of the employees along with the corresponding city of the department in which they are working

```
SELECT ecode, ename, (SELECT city FROM dept
WHERE emp.dno=dept.dno) "CITY OF DEPT"
FROM emp;
```

Output:

ECODE	ENAME	CITY OF DEPT
E01	KOUSHIK GHOSH	KOLKATA
E02	JAYANTA DUTTA	KOLKATA
E03	HARI NANDAN TUNGA	CHENNAI
E04	JAYANTA GANGULY	KOLKATA
E05	RAJIB HALDER	KOLKATA
E06	JISHNU BANERJEE	CHENNAI
E07	RANI BOSE	KOLKATA
E08	GOUTAM DEY	KOLKATA
E09	PINAKI BOSE	CHENNAI

9 rows selected.

When the subquery does not return a row, a database *null* is used as the result of the subquery.

Query: Find the employees of the department located at BANGALORE

```
SELECT ecode, ename, (SELECT city FROM dept
WHERE emp.dno=dept.dno AND city = 'BANGALORE') "CITY OF DEPT"
FROM emp;
```

Output:

ECODE	ENAME	CITY OF DEPT
E01	KOUSHIK GHOSH	
E02	JAYANTA DUTTA	
E03	HARI NANDAN TUNGA	
E04	JAYANTA GANGULY	

E05	RAJIB HALDER
E06	JISHNU BANERJEE
E07	RANI BOSE
E08	GOUTAM DEY
E09	PINAKI BOSE

Form of Basic SQL
Query

NOTES

9 rows selected.

Query: List the employees working in the personnel department

```
SELECT * FROM emp WHERE 'PERSONNEL' = (SELECT dname FROM dept
WHERE emp.dno=dept.dno);
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95

If the scalar subquery returns multiple rows, then Oracle generates an error. For example, to list the code, name of the employees along with the corresponding project id, which they are assigned.

```
SELECT ecode, ename,
(SELECT pid FROM assign WHERE emp.ecode=assign.ecode)
"PROJECT ID"
FROM emp;
```

Output:

```
(SELECT pid FROM assign WHERE emp.ecode=assign.ecode) "PROJECT
ID" *
ERROR at line 2:
ORA-01427: single-row subquery returns more than one row
```

Here multiple PIDs results from the scalar query. Therefore, the whole query fails.

Table Subqueries

Table subqueries are queries used in the FROM clause for replacing a table name. The result set of a table subquery acts like a base table in the FROM list. Table subqueries can have a correlation name in the FROM list. It is also known as *inline view*. They can also be in outer joins.

Query: List the details of the employees along with the corresponding location (city) of the RESEARCH department

```
SELECT E.* , city
FROM emp E, (SELECT dno , city From dept WHERE dname='RESEARCH')
WHERE E.dno=D.dno;
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E03	HARI NANDAN TUNGA	4000	D02	PROGRAMMER	01-JUL-95 CHENNAI
E06	JISHNU BANERJEE	6500	D02	SYSTEM MANAGER	19-SEP-96 CHENNAI
E09	PINAKI BOSE	5500	D02	PROGRAMMER	26-AUG-94 CHENNAI

NOTES

There is another application of an inline query. If it is tried to use ROWNUM to restrict a query by a range that does not start with 1, it is found that it does not work. For example:

```
SELECT * from emp WHERE ROWNUM BETWEEN 5 AND 9 ;
```

Output:

no rows selected

The reason for this is that ROWNUM is a psuedo-column produced AFTER the query returns. Normally, it can only be used to restrict a query to return a rownumber range that starts with 1 (like *rownum < 5*). An ‘Inline View’ can be used to get around this limitation.

```
SELECT rn, ecode, ename, salary
FROM (SELECT ROWNUM rn, ecode, ename, salary
      FROM emp)
WHERE rn BETWEEN 5 AND 9;
```

Output:

RN ECODE	ENAME	SALARY
5 E05	RAJIB HALDER	4000
6 E06	JISHNU BANERJEE	6500
7 E07	RANI BOSE	3000
8 E08	GOUTAM DEY	5000
9 E09	PINAKI BOSE	5500

This SELECT statement in the FROM clause basically does a full query of the table, and then returns the values (along with the pseudo-column *ROWNUM*) to the outer query. The outer query can then operate on the results of the inner query. Since ‘ROWNUM’ is a pseudo-column and therefore, a reserved word, it is needed to alias that column (here it is ‘rn’) in the inner query in order to refer to it in the outer query.

Query: Select EVERY 4th row from EMP table

```
SELECT*
  FROM (SELECT rownum rn, ecode, ename, salary
        FROM emp ) A
 WHERE MOD (A.ROWNUM, 4) = 0;
```

Output:

RN ECODE	ENAME	SALARY
4 E04	JAYANTA GANGULY	6000
8 E08	GOUTAM DEY	5000

Update through an Inline View

Through the inline view, updation is possible.

Query: Double the salary of the employees who are working in the department located at CHENNAI

```
UPDATE (SELECT ecode, emp.dno, salary, dept.dno, dname, city
       FROM dept, emp WHERE emp.dno=dept.dno and city='CHENNAI')
      SET salary=salary*2;
```

*Form of Basic SQL
Query*

Output:

3 rows updated.

Let's check the updation:

```
SELECT * FROM emp;
```

Output:

ECODE	ENAME	SALARY	DNO	DESG	DT_JN
E01	KOUSHIK GHOSH	5000	D01	SYSTEM ANALYST	10-MAR-93
E02	JAYANTA DUTTA	3500	D01	PROGRAMMER	15-JAN-94
E03	HARI NANDAN TUNGA	8000	D02	PROGRAMMER	01-JUL-95
E04	JAYANTA GANGULY	6000	D03	ACCOUNTANT	12-SEP-96
E05	RAJIB HALDER	4000	D03	CLERK	07-OCT-95
E06	JISHNU BANERJEE	13000	D02	SYSTEM MANAGER	19-SEP-96
E07	RANI BOSE	3000	D01	PROJECT ASSISTANT	17-JUN-97
E08	GOUTAM DEY	5000	D01	PROGRAMMER	23-OCT-97
E09	PINAKI BOSE	11000	D02	PROGRAMMER	26-AUG-94

NOTES

WITH CHECK OPTION in Inline View

The WITH CHECK OPTION may be used to create the inline view as a constrained view and prohibit specific insert and update operations through the inline view. The WITH CHECK OPTION is used with the WHERE clause. For example, to curb users from altering data for any employee in the PROJECT department, the SQL statement will be as follows:

```
UPDATE (SELECT ecode, ename, dno, salary FROM emp
        WHERE dno != (SELECT dno FROM dept WHERE dname='PROJECT')
        WITH CHECK OPTION) empl
        SET empl.salary=7000
        WHERE empl.ecode='E01';
```

Output:

0 rows updated.

Now the following SQL statement is entered:

```
UPDATE (SELECT ecode, ename, dno, salary FROM emp
        WHERE dno != (SELECT dno FROM dept WHERE dname='PROJECT')
        WITH CHECK OPTION) empl
        SET empl.salary=7000
        WHERE empl.ecode='E03';
```

NOTES

Output:

1 row updated.

Here updation is successful as employee with employee code 'E03' is working in the RESEARCH department (dno is 'D02').

The WITH CHECK OPTION in an inline view also protects against data modification that would not be visible via the inline view. For example, Employee with employee code 'E03' is shifted from the RESEARCH department to the PROJECT department. To do this assignment, the following statement is entered:

```
UPDATE (SELECT ecode, ename, dno, salary FROM emp
WHERE dno != (SELECT dno FROM dept WHERE dname='PROJECT')
WITH CHECK OPTION) empl
SET empl.dno=(SELECT dno FROM dept WHERE dname='PROJECT')
WHERE empl.ecode='E03';
```

Output:

WHERE dno != (SELECT dno FROM dept WHERE dname='PROJECT')

ERROR at line 2:

ORA-01402: view WITH CHECK OPTION where-clause violation

The above statement generated an error as the data would no longer be visible via the inline view.

7.2.8 Impact on SQL Constructs

SQL constructs along with description are tabulated below.

S. NO.	Name	Syntax	Description	Syntax rules
1.	SET Statement	SET <target> = <source> target is an output link variable or column source is an expression or NULL.	Finds out the expression and assigns the output link to column and result to a variable.	<ul style="list-style-type: none">If the data types of the source and target are not alike then the source is automatically converted to the target's data type before assignment.If the source is not convertible to the target data type then an error will occur.The conversions which are supported are the same as which acceptable by the CAST function.
2.	BEGIN END Block	[<beginning label>:] BEGIN <block body> END [<ending label>] block body is an SQL statement list	It groups statements together. This statement acts as an organization tool which does not affect the business rule logic.	<ul style="list-style-type: none">A block label is an SQL identifier which gives a name for the block.If an ending label is specified then it must match the beginning label.Block labels are optional.
3.	DISPLAY Statement	DISPLAY (<value list>) value list is <expression> [,<expression>]...	This statement writes values to the SYSPRINT spooler.	<ul style="list-style-type: none">Non character values are displayed, they were converted to VARCHAR.Expression values which are written to the SYSPRINT spooler from left to right on one line and terminated by a carriage return.

NOTES

4.	INSERT Statement	INSERT INTO <output link>	This statement sends columns down an output link.	<ul style="list-style-type: none"> To verify that data has been successfully written to an output link, check the variable SQLCA. A business rule that must consists of at least one INSERT statement for each output link. SQLCODE after the INSERT statement. A nonzero value indicates an error.
5.	LEAVE Statement	LEAVE <loop label> loop label is an SQL identifier which gives a name for a loop	Exits a loop	<ul style="list-style-type: none"> The loop label must match the label of the nearest LOOP statement which contains the LEAVE statement. A LEAVE statement must be used in a loop body. It exits the loop which allows the program execution to resume at the statement following the END LOOP.
6.	LOOP Statement	<beginning label> : LOOP <loop body> END LOOP [<ending label>] loop body is an SQL statement list	This statement executes the loop body continuously until a LEAVE statement is executed to exit the loop.	<ul style="list-style-type: none"> The loop body must contain a LEAVE statement. If an ending label is specified, it must match the beginning label. A loop label is an SQL identifier that gives a name for the loop. The ending label is optional.
7.	IF Statement	IF <condition> <THEN clause> [<ELSE clause>] END IF condition is a comparison whose value determines the program flow. THEN clause is the statement to be executed if the condition is true. ELSE clause is the statement to be executed if the condition is false.	These statements allows conditional execution of SQL statements.	<ul style="list-style-type: none"> If the condition evaluates to TRUE then the business rule executes the SQL statements in the THEN clause. If the condition evaluates to UNKNOWN (NULL), FALSE or, the business rule executes the SQL statements in the ELSE clause if present. A condition must be a Boolean expression.
8.	EXIT Statement	Its value is returned to the operating system.	This statement achieves an implicit COMMIT	These statements terminate the job with the status.
9.	ROLLBACK Statement	ROLLBACK	This statement Cancels any inserts, updates, and deletes prepared to relational tables in a job. Since the last COMMIT statement.	<ul style="list-style-type: none"> A job must contain a target Relational or Relational stage. To use a ROLLBACK statement.
10.	COMMIT Statement	COMMIT	This statement commits any inserts, updates, and deletes prepared to relational tables in a job.	<ul style="list-style-type: none"> To use a COMMIT statement, a job must consists of a target Relational or Relational stage.

7.3 INTEGRITY CONSTRAINTS IN SQL TRIGGERS AND ACTIVE DATABASES

Integrity constraints enforce limitations on the allowable data in the database. It also includes the simple structure and type restrictions which are enforced by the basic schema definition.

NOTES

The integrity constraints are used:

- To implement consistency across data in the database.
- To handle data-entry errors.
- Analyze data that may choose to store the data or process queries accordingly
- When writing database updates take care about correctness criteria.

For example: consider a schema:

```
Student (ID, name, address, GPA, sizeHS)
Campus (location, enrollment, rank)
Apply (ID, location, date, major, decision)
```

Example of constraints are:

- A student with $GPA < 5.0$ can only apply to campuses with $rank > 5$
- All applications with $date < 1/1/19$ have non-NULL decision
- Apply.ID and Apply.location these should appear in Student.ID and Campus.location, respectively
- Campus.rank ≤ 5

Types of Integrity Constraints for Relational Databases:

There are different types of Integrity constraints for relational databases:

- Non-null
- Referential integrity
- Tuple-based
- Attribute-based
- General assertions

Declaring and Enforcing Constraints

To declare constraints is of different forms:

- **Declared with original schema:** these types of constraints must hold after bulk loading.
- **Declared later:** it means the constraints must hold on current database.

If any SQL statement causes a constraint violated then error will be generated.

Non-Null Constraints

Non-Null constraints allows the attributes to have non null values.

For example:

```
CREATE TABLE Student (ID integer NOT NULL,
```

```
name char(30) NOT NULL,  
address char(100),  
GPA float NOT NULL,  
sizeHS integer)
```

*Form of Basic SQL
Query*

NOTES

Key Constraints

There are two types of keys in SQL:

- **PRIMARY KEY:** It is the key that uniquely identified the records and at most one per table. It repeatedly non-null, automatically indexed.
- **UNIQUE:** It consists of any number per table which is automatically indexed. There are two ways to define keys in SQL:

For example:

```
CREATE TABLE Student (ID integer PRIMARY KEY,  
name char(30),  
address char(100),  
GPA float,  
sizeHS integer,  
UNIQUE (name, address))
```

Referential Integrity

It is the most important and common kind of constraint.

- The referenced attribute must be PRIMARY KEY or UNIQUE (e.g., Student.ID, Student.(name,address), Campus.location).
- Referencing attribute called FOREIGN KEY (e.g., Apply.ID, Apply.Location)

For example:

```
CREATE TABLE Apply(ID integer REFERENCES Student(ID),  
location char(25),  
date char(10),  
major char(10),  
decision char,  
FOREIGN KEY (location) REFERENCES Campus(location))
```

Referential Integrity Enforcement

Referential integrity Enforcement consist of the following:

- SET DEFAULT it means set all referencing values to default for that column.
- SET NULL that is set all referencing values to NULL.
- CASCADE means to delete all tuples with referencing values.
- Disallow values by default.

NOTES

Attribute-Based Constraints

These constraints have:

- Type specifications which are type of attribute-based constraint.
- Non-null constraints which are type of attribute-based constraint.

For example:

```
CREATE TABLE Student (...  
    GPA float CHECK(GPA <= 4.0 AND GPA > 0),  
    ...)  
  
CREATE TABLE Apply (...  
    decision char CHECK(decision in ('Y', 'N', 'U'))  
    ...)
```

Triggers

These are the most general than constraints. These are available in SQL-99 but not in SQL2.

Syntax:

```
CREATE TRIGGER <name>  
    BEFORE | AFTER | INSTEAD OF <events>  
    <referencing clause> // optional  
    FOR EACH ROW // optional  
    WHEN (<condition>) // optional  
    <action>  
    <events> can be:  
        INSERT ON R  
        DELETE ON R  
        UPDATE [OF A1, A2, ..., An] ON R  
    • Here <condition> implies that like general assertion.  
    • The <action> implies that the sequence of SQL statements.  
    • FOR EACH ROW  
        o If present, execute trigger once for each tuple changed and If absent,  
            execute trigger once for each relevant statement.  
        o It has rule that is “row-level” versus “statement-level”.  
        o Trigger executes after statement completes.  
    • <referencing clause>: REFERENCING <thing1> AS <var1>  
        <thing2> AS <var2>, etc.  
    <thing> which can be:
```

NOTES

- o NEW TABLE : The current values of inserted or updated tuples. It consists of row-level or statement-level that is INSERT or UPDATE.
- o NEW ROW : The current value of inserted or updated tuple. It consists of row-level only that is INSERT or UPDATE.
- o OLD TABLE: The previous values of deleted or updated tuples. It consists of row-level or statement-level that is DELETE or UPDATE
- o OLD ROW : The previous value of deleted or updated tuple. It consists of row-level only that is DELETE or UPDATE

Example: Consider an application tuple is inserted for a student with GPA > 3.9 and sizeHS > 1500 to Berkeley, then set decision to “Y”.

```
CREATE TRIGGER AutoAccept
AFTER INSERT ON Apply
REFERENCING NEW ROW AS NewApp
FOR EACH ROW
WHEN (NewApp.location = 'Berkeley' AND
      3.9 < (SELECT GPA FROM Student WHERE ID = NewApp.ID) AND
      1500 < (SELECT sizeHS FROM Student WHERE ID = NewApp.ID))
UPDATE Apply
SET decision = 'Y'
WHERE ID = NewApp.ID
AND location = NewApp.location
AND date = NewApp.date
```

Example: Same trigger without FOR EACH ROW

```
CREATE TRIGGER AutoAccept
AFTER INSERT ON Apply
REFERENCING NEW TABLE AS NewApps
UPDATE Apply
SET decision = 'Y'
WHERE (ID,location,date) IN (SELECT ID,location,date FROM
                               NewApps)
AND location = 'Berkeley'
AND 3.9 < (SELECT GPA FROM Student WHERE ID = Apply.ID)
AND 1500 > (SELECT sizeHS FROM Student WHERE ID = Apply.ID)
```

7.4 INTRODUCTION TO SCHEMA REFINEMENT

The schema refinement means to address and a refinement approach based on decompositions. Decomposition can eliminate redundancy but it can be the problems of its own and should be used with caution. Redundant storage of information is

NOTES

the root cause of decomposition. Storing the same information redundantly, can causes to several problems:

- Redundant storage: it means some information is stored repeatedly.
- Update anomalies: An inconsistency is created unless all copies are similarly updated, if one copy of such repeated data is updated,
- Insertion anomalies: Unless some other information is stored as well, it may not be possible to store some information
- Deletion anomalies: Without losing some other information, it may not be possible to delete some information..

For example: consider a relation: paid Emps(name, ssn, lot, rating, hours_paid wages, hours_paid worked)

Name	Ssn	lot	Rating	Hours_paid wages	Hours_paid worked
John	11001101	22	6	6	40
Smith	21001101	43	4	4	50
Robert	31001101	35	7	8	10
Samson	41001101	28	8	10	30

Here in above table, some information is stored multiple times like, the rating value 4 which is corresponding to the hours_paid wage 6, which is repeated 2 times.so here redundancy causes to potential inconsistency.

- **In case of updation:** for example, consider the hourly wages in the first tuple which can be updated without making the same change in the second tuple.
- **In case of insertion:** we cannot insert a tuple for an employee until we know about the hourly wage for the employee's rating value.
- **In case of deletion:** if we delete all tuples with a given rating value then we lose the association between that rating value and its hourly wage value.

Check Your Progress

1. Why SQL is easy to use?
2. What is data definition language?
3. What is the use of data control command?
4. How many constraints are there?
5. What happens when a column is declared as the PRIMARY KEY?
6. What is a UNIQUE constraint?

7.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. SQL is a ‘declarative language’ (non-procedural). This makes SQL relatively easy-to-use as compared to other programming languages. In SQL, the programmer only specifies what data is needed, but it is not required to specify how to retrieve it. The underlying DBMS analyses the SQL and formulates the way to retrieve the required information.
2. Data Definition Language (DDL) is the part of SQL that allows a database user to create and restructure database objects, such as the creation or the deletion of a table.
3. Data control commands in SQL allow to control access to data within the database. These DCL commands are normally used to create objects related to user access and also control the distribution of privileges among users.
4. There are two basic types of constraints—column constraints and table constraints. Column constraints apply only to individual columns and are provided with the column definition. Table constraints apply to groups of one or more columns and are defined separately from the definitions of the columns in the table.
5. When a column is declared as the PRIMARY KEY, an index on this column is automatically created and assigned a unique name by ORACLE. The additional constraints `UNIQUE` and `NOT NULL` are implied by the PRIMARY KEY constraint.
6. `UNIQUE` is a constraint that enforces distinct column values on a table column and can contain a `NULL` record also. PRIMARY KEY is a constraint similar to `UNIQUE`; however, it cannot have a Null value for the column. There can only be one primary key in a table. Moreover, a PRIMARY KEY can be the parent column for a Parent–Child relationship (Foreign Key).

NOTES

7.6 SUMMARY

- There are two forms of SQL—interactive and embedded. Interactive SQL operates on a database to produce output for user demand. In embedded SQL, SQL commands can be put inside a program written in some other language (called host language) like C, C++, etc. SQL commands are of varied types to suit different purposes. The primary types are Data Definition Language (DDL), Data Manipulation Language (DML), Data Query Language (DQL), Data Control Language (DCL) and Transactional Control Language (TCL).

NOTES

- A join is a query that retrieves rows from more than one table or view. Most join queries contain the WHERE clause conditions that compare two columns, each from a different table. Such a condition is known as join condition.
- If two tables in a join query have no join condition, their Cartesian product is returned. It is also known as cross-join.
- An *equijoin* is a join in which the join condition contains an equality operator.
- An inner join (sometimes called ‘simple join’) is a join of two or more tables that returns only those rows that satisfy the join condition.
- A left outer join is a join where records from the left table that have no matching key in the right table are included in the result set.
- A right outer join is a join where records from the right table that have no matching key in the left table are included in the result set. We can rewrite the above query using the right outer join.
- A full outer join is a join where records from the first table are included that have no corresponding record in the other table and records from the other table are included that have no records in the first table.
- A self-join is a join of a table to itself. This table appears twice in the FROM clause and is followed by table aliases that qualify column names in the join condition.
- The query in the WHERE clause is called a subquery. The subquery is often referred to as the inner query and the surrounding query is called the outer query.
- SQL has three basic types of subqueries: Predicate Subquery which are extended logical constructs in the WHERE (and HAVING) clause, Scalar Subquery which are stand-alone queries that return a single value; they can be used wherever a scalar value is used, Table Subquery which are queries nested in the FROM clause. All subqueries should be enclosed in parentheses.

7.7 KEY WORDS

- **Data Manipulation Language:** The part of SQL used to manipulate data within objects of a relational database.
- **Data Query Language (DQL):** The most concentrated focus of SQL comprising only one command for modern relational database users.
- **SELECT:** A command, accompanied by many options and clauses, used to compose queries against a relational database.
- **Data Control Language (DCL):** In SQL it allows us to control access to data within the database.

7.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Define the terms DDL and DML. What are the differences between them?
2. Define the term data dictionary. What is the difference between procedural and non-procedural DML?
3. What is the use of SELECT statement?
4. What is the difference between the IN and BETWEEN clauses?
5. How is it possible to select null values? Give examples.
6. What is embedded SQL?
7. What do you understand by schema refinement?

Long Answer Questions

1. What are different categories of SQL commands?
2. What are the different types of data types in SQL?
3. Explain the different types of outer join with the help of examples.
4. What are the different types of integrity constraints for relational database?
5. What are the different form of nested queries? Explain each of them.

NOTES

7.9 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.

NOTES

- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

UNIT 8 NORMAL FORMS

Structure

- 8.0 Introduction
 - 8.1 Objectives
 - 8.2 Problems Caused by Redundancy and Normalization
 - 8.3 Decomposition
 - 8.4 Reasoning about FDS
 - 8.4.1 Types of Dependency
 - 8.4.2 Armstrong Axioms—Inference Rules for Functional Dependencies
 - 8.4.3 Key and Functional Dependency
 - 8.5 First Normal Form
 - 8.6 Second Normal Form
 - 8.7 Third Normal Form
 - 8.8 BCNF
 - 8.9 Answers to Check Your Progress Questions
 - 8.10 Summary
 - 8.11 Key Words
 - 8.12 Self Assessment Questions and Exercises
 - 8.13 Further Readings
-

NOTES

8.0 INTRODUCTION

In order to minimize data duplication and increase data integrity, database normalization is the process of structuring a relational database in accordance with a sequence of so-called normal types. As a part of his relational model, it was first suggested by Edgar F. Codd.

Normalization requires arranging a database's columns (attributes) and tables (relations) to ensure that their dependencies are properly implemented by restrictions on database integrity. It is done either by a method of synthesis (creating a new database design) or decomposition by applying certain formal rules (improving an existing database design).

Normalization is a technique used to analyse the given relational schema to reduce redundancy, inconsistency, and deletion, updation and insertion anomalies. A set of relations that is normalized makes management and retrieval of data easy in a database system.

In this unit, you will study about the normal forms, problem caused by data redundancy, problem related to decompositions, reasoning about FDS, FIRST, SECOND, THIRD normal forms and BCNF.

NOTES

8.1 OBJECTIVES

After going through this unit, you will be able to:

- Define what normalization is
- Understand the need for normalization
- Explain the significance of functional dependency
- Understand different types of normal forms

8.2 PROBLEMS CAUSED BY REDUNDANCY AND NORMALIZATION

Redundancy can be defined as the database containing multiple copies of same data. It arises when the database is not normalized. Database design is essentially a business problem, not a data problem. Data modelling is primarily the result of a thorough understanding of information about an enterprise. The aim of a relational database design is to create a group of relation schemas that symbolize the real-world situation that is being modelled. The design must also allow the storage of information without redundancy as well as retrieval of information efficiently. A technique called normalization is used to analyse the given relational schema to get the following desired properties:

- Reducing redundancy
- Reducing inconsistency
- Reducing the deletion, updation and insertion anomalies

This is done by decomposing the original relation into certain smaller relation schemas each of which is in an appropriate normal form, which is a structural form that satisfies certain rules or restrictions. Normalization is a bottom-up approach for database design that begins by examining the relationships between attributes.

A string of tests is performed on relation schema that is individual in nature so that the relational database is normalized to some amount. A relation that violates a test must be decomposed into relations that meet the normalization tests individually. This occurs primarily if the test is unsuccessful. Therefore, we can define normalization as: ‘The process of normalization is a formal method that identifies relational schemas based upon their primary or candidate keys and the functional dependencies that exist among their attributes’.

McFadden has defined normalization in his book *Modern Database Management*. According to him, ‘Normalization is the process of decomposing relations [tables] with anomalies to produce smaller, well-structured relations.’

A fully normalized table should do the following:

- Avoid update, insert and delete anomalies.

- Describe the structure of relational tables.
- Prevent unnecessary data duplication.
- Avoid inconsistencies in the database.
- Prevent information loss.

Normal Forms

NOTES

Need for Normalization

A database system aims to reduce redundancy, whereby information is stored only once. Storage space is wasted and the total size of the data stored is increased as a result of storing information many times. The E-R model provides a starting point to identify flaws in table design, such as duplication that makes it hard to maintain relational integrity. Consider the following two sets of relation schemas and instances for each:

Set- I

EMPLOYEE (empno, name, desg, salary, deptno)
DEPT(deptno, deptname, location)

EMPNO	NAME	DESG	SALARY	DEPTNO
E01	KOUSHIK GHOSH	MANAGER	25000	D01
E05	PINAKI BOSE	PROGRAMMER	15000	D03
E06	GOUTAM DEY	FINANCE OFFICER	10000	D02
E07	RAJIB HALDER	SYSTEM ANALYST	20000	D03

EMPLOYEE

DEPTNO	DEPTNAME	LOCATION
D01	ADMINISTRATION	KOLKATA
D02	PERSONNEL	MUMBAI
D03	PROJECT	CHENNAI

DEPT

Set- II

EMPNO	NAME	DESG	SALARY	DEPTNO	DEPTNAME
E01	KOUSHIK GHOSH	MANAGER	25000	D01	ADMINISTRATION
E05	PINAKI BOSE	PROGRAMMER	15000	D03	PROJECT
E06	GOUTAM DEY	FINANCE OFFICER	10000	D02	PERSONNEL
E07	RAJIB HALDER	SYSTEM ANALYST	20000	D03	PROJECT

EMP

NOTES

The two different sets of relation schemas contain exactly the same information, though in slightly different ways. In Set II, the EMP relation contains much redundant data; the details of each department are repeated for every employee working at that branch. In contrast, in DEPT relation, the department details appear only once for each department and only the department-number is repeated in the EMPLOYEE relation to indicate where each employee is located. The relations that contain redundant data may have problems called update anomalies, which are classified into insertion anomalies, deletion anomalies and modification anomalies.

Insertion Anomaly

The database design prevents some data from being represented due to insertion anomalies. There are two main types of insertion anomalies, which can be illustrated using the relation schemas (Sets I and II) from above.

Suppose we want to insert the details for a new department ‘D04’ but no employee members are assigned to it. There is no way, in the EMP relation, to represent this information.

EMPNO	NAME	DESG	SALARY	DEPTNO	DEPTNAME
E01	KOUSHIK GHOSH	MANAGER	25000	D01	ADMINISTRATION
E05	PINAKI BOSE	PROGRAMMER	15000	D03	PROJECT
E06	GOUTAM DEY	FINANCE OFFICER	10000	D02	PERSONNEL
E07	RAJIB HALDER	SYSTEM ANALYST	20000	D03	PROJECT
NULL	NULL	NULL	NULL	D04	SALES

We cannot simply enter null values for the attributes belonging to employee information because one of these attributes is the key of the relation; and remember that no key attribute can be assigned a null value because this violates referential integrity. The relations in Set I, however, do not have this problem because the details for the departments and the employees are maintained in separate relations. The department details would have to be inserted first in the DEPT relation and after that employee information is to be inserted into the EMPLOYEE table.

Again, if we want to insert the details for a new employee in the EMP relation, the new tuple must also include correct information about the branch where he will be working. If not, the data will become inconsistent.

Deletion Anomaly

Suppose we want to delete a tuple from the EMP relation, say the employee whose employee number is E06. We now face a problem, which is just the reverse of the second type of insertion anomaly. Through the deletion of this tuple, we would delete information concerning this particular department. For example, if we wanted to delete information related to an employee whose employee code is

E06 in the EMP instance just discussed, then we would lose information concerning department D02.

Normal Forms

EMPNO	NAME	DESG	SALARY	DEPTNO	DEPTNAME
E01	KOUSHIK GHOSH	MANAGER	25000	D01	ADMINISTRATION
E05	PINAKI BOSE	PROGRAMMER	15000	D03	PROJECT
E07	RAJIB HALDER	SYSTEM ANALYST	20000	D03	PROJECT

NOTES

Instance of EMP after Deleting the Tuple of Employee E06

However, in Set I, we can safely delete the information of E06 without deleting information concerning department D02.

EMPNO	NAME	DESG	SALARY	DEPTNO
E01	KOUSHIK GHOSH	MANAGER	25000	D01
E05	PINAKI BOSE	PROGRAMMER	15000	D03
E07	RAJIB HALDER	SYSTEM ANALYST	20000	D03

Instance of EMPLOYEE after deleting the Information of Employee E06

DEPTNO	DEPTNAME	LOCATION
D01	ADMINISTRATION	KOLKATA
D02	PERSONNEL	MUMBAI
D03	PROJECT	CHENNAI

Instance of DEPT after Deleting the Information of Employee E06

Update Anomaly

Suppose we want to change the value of one of the attributes of a particular department. To change the address of department D03 in the Set II schema, we have to update on every tuple in the relation instance. If this modification is not carried out on all the appropriate tuples in the relation instance, the relation, and hence the database, will become inconsistent. Once again, however, in the Set I format, updating the location for all employees who work in department D03 requires the updation of only a single tuple in the DEPT relation. Modification anomalies are related to the first type of insertion anomaly.

It is to be noted that the EMPLOYEE relation still has insertion anomaly to some extent. We cannot insert employee information without knowing his/her department number. We assumed here that DEPTNO is the foreign key of EMPLOYEE relation referencing DEPT relation. The following set of relations (Set-III) may be the well-designed database, which suffers no insertion or deletion anomalies and has little updation anomaly.

Set- III

```
EMPLOYEE (EMPNO, NAME, DESG, SALARY)
DEPT (DEPTNO, DEPTNAME, LOCATION)
WORK_ON (EMPNO, DEPTNO)
```

Normal Forms

NOTES

EMPLOYEE

EMPNO	NAME	DESG	SALARY
E01	KOUSHIK GHOSH	MANAGER	25000
E05	PINAKI BOSE	PROGRAMMER	15000
E06	GOUTAM DEY	FINANCE OFFICER	10000
E07	RAJIB HALDER	SYSTEM ANALYST	20000

DEPT

DEPTNO	DEPTNAME	LOCATION
D01	ADMINISTRATION	KOLKATA
D02	PERSONNEL	MUMBAI
D03	PROJECT	CHENNAI

WORK-ON

EMPNO	DEPTNO
E01	D01
E05	D03
E06	D02
E07	D03

8.3 DECOMPOSITION

Decomposing relations is an accepted way of removing anomalies. Decomposition is the process of breaking a relation into multiple relations. Decomposition of relations must make certain that from the decomposed relations, an original relation can be reconstructed. If decomposition of a relation is treated carelessly, it can lead to information loss.

A relation R with schema $\{A_1, \dots, A_n\}$ can be decomposed into a couple of relations R_1 and R_2 with schemas $\{B_1, B_2, \dots, B_m\}$ and $\{C_1, C_2, \dots, C_k\}$, such that

- $\{A_1, A_2, \dots, A_n\} = \{B_1, B_2, \dots, B_m\} \dot{\times} \{C_1, C_2, \dots, C_k\}$
- The tuples in S are the projections onto $\{B_1, B_2, \dots, B_m\}$ of all the tuples in R.
- Same for R_2

It must be noted that the decomposition process is not disjoint necessarily.

8.4 REASONING ABOUT FDS

A database is a set of information that is related; and it is, therefore, inevitable that some items of information in the database would depend on some other items of information. For example, at any point of time in a database, an employee code

would always identify an employee of an organization. A person may change his/her name or his/her residence address but at any instant of time an employee can be identified with his/her employee code. The information is either single-valued or multi-valued. Information such as employee code, his/her name, or his/her date of birth is single-valued facts; qualifications of an employee or telephone numbers of an employee, etc. are multi-valued facts. The single-valued facts formalize the concept of functional dependency and the multi-valued facts define the concept of multi-valued dependency, which will be discussed in the next unit. The Entity–Relationship model provides a starting point for identifying schemas and integrity constraints. Functional dependencies offer approaches for refining the modelling by analysing integrity constraints. Actually, not all schemas are equal. Some are bad, some are good and some are superior to others. The functional dependencies theory gives us a technique of recognizing defective schemas (those that possess particular anomalies), and changing them to high-quality schemas. This is the fundamental theory of relational database design. Other database models have no counterparts which makes the relational model powerful: It gives a tougher base for database design.

A Functional Dependency (FD) is derived from mathematical theory. It implies the dependency of values of a single attribute or a collection of attributes, on another attribute or a collection of attributes.

Functional dependencies are dependant on the information of what can be stored in the relation and serve as integrity constraints, which should be validated when adding or updating tuples in a relation. A relation state r of R that satisfies the functional dependency constraints is called a legal relation state (or legal extension) of R .

Formal Definition: Let $A_1, A_2 \dots A_k$, and B be attributes of a relation R . One can say that B is *functionally dependent* on $\{A_1, A_2 \dots A_k\}$ if and only if whenever two tuples t and u agree on $A_1, A_2 \dots A_k$, they also agree on B .

That is, if $t.A_1 = u.A_1$

$$t.A_2 = u.A_2$$

.

.

$$t.A_k = u.A_k$$

then, $t.B = u.B$

More formally, we can define as follows:

Consider a relation scheme $R (A_1, A_2 \dots A_n)$ and X and Y subsets of $\{A_1, A_2 \dots A_n\}$. Y is functionally dependent on X , denoted by $X \circledast Y$, if and only if for every relation instance r of R , for every pair of two tuples t and u in r , $t[X] = u[X]$ implies $t[Y] = u[Y]$ where $X \subseteq R$ and $Y \subseteq R$.

NOTES

NOTES

Pictorially we can represent this as follows:

R	X	Y	
t			
S			
	If t and u agree here	Then they must agree here	

An FD tells us about *any* two tuples t and u in relation R.

We can conclude that if X and Y are attributes or sets of attributes of relation $r(R)$, then Y is functionally dependent on X, if each value of X determines exactly one value of Y, i.e. the values in tuples corresponding to the attributes in X uniquely determine the values corresponding to the attributes in Y. Diagrammatically, an FD can be represented as follows:

- The single arrow denotes ‘functional dependency’.
- $X \rightarrow Y$ can also be read as ‘X determines Y’.
- The \models denotes ‘logical implication’.

Determinant of an FD: An attribute is a determinant if it occurs on the left-hand side of a functional dependency statement (Determinant \rightarrow Dependent attribute). That is, a determinant is an attribute on which another attribute is functionally dependent.

We can make it more understandable with the help of the EMPLOYEE table given below. An employee code uniquely determines an employee, denoted by $E\text{CODE} \rightarrow E\text{NAME}$; salary is also functionally dependent on employee code, denoted by $E\text{CODE} \rightarrow \text{SALARY}$.

E\text{CODE}	E\text{NAME}	SALARY
E01	Koushik Ghosh	12000
E02	Goutam Dey	8000
E03	Jayanta Das	9000
E04	Pinaki Bose	15000
E05	Rajib Halder	10000

Two employees can have the same name or salary but they might not be the same person. To differentiate them, they should have different employee codes. This reflects in FDs $E\text{CODE} \rightarrow E\text{NAME}$ and $E\text{CODE} \rightarrow \text{SALARY}$. $E\text{CODE}$ can be treated as candidate key.

8.4.1 Types of Dependency

- **Full Functional Dependency:** $X \rightarrow Y$ is a **full functional dependency** if Y is *not* dependent on any proper subset of X and Y is said to be **fully**

functionally dependent upon X. That is, in $X \rightarrow Y$ where X is the minimal set of attributes that uniquely determines Y. If we remove an attribute from X, it no longer determines Y.

Normal Forms

- **Partial Dependency:** $X \rightarrow Y$ is a **partial dependency** if Y is dependent on a proper subset of X and Y is said to be **partially dependent** upon X. $X \rightarrow Y$ where X is not the minimal set of attributes that uniquely determines Y. Some attributes could be removed from X, and the dependency would still hold. Partial dependencies must be decomposed into separate relations. It is **Partial** if removal of any attribute from X **does not result** in the violation of the rule.

NOTES

- **Trivial Functional Dependency:** Trivial functional dependencies are those FDs, which are impossible **not** to satisfy. A dependency is trivial, if and only if the right-hand side (the consequent) is a subset (not necessarily a proper subset) of the left-hand side (the determinant). That is, an FD $A_1, A_2, \dots, A_n \rightarrow B_1, B_2, \dots, B_m$ is said to be a **trivial FD** if and only if $\{B_1, B_2, \dots, B_m\} \subseteq \{A_1, A_2, \dots, A_n\}$.

Some of the examples of trivial FD include the following:

$ABC \rightarrow A$, $ABC \rightarrow BC$, $BC \rightarrow BC$

$E CODE, ENAME, DT_JN \rightarrow ENAME$

The FD $A_1, A_2, \dots, A_n \rightarrow B_1, B_2, \dots, B_m$ is equivalent to $A_1, A_2, \dots, A_n \rightarrow C_1, C_2, \dots, C_k$ if C's are all those B's those are not in A's. Pictorially,

			C
	A		B
t			
u			
	If t and u agree on A	They must agree in B	So surely they agree in C

Therefore, a functional dependency $A_1, A_2, \dots, A_n \rightarrow B_1, B_2, \dots, B_m$ is

- **TRIVIAL** if the B's are a subset of A's.
- **NON-TRIVIAL** if at least one of the B's is not among A's
- **COMPLETELY TRIVIAL** if none of the B's is one of the A's.

can always remove from the right-hand side of a functional dependency those attributes that appear on the left.

NOTES**Some observations/comments on FD**

- When two rows in a relation have the same values for X, these rows will have the **same** values for Y.
- When two rows in a table have the same values for Y, these rows *may have different* values for X.
- A functional dependency is a property of the relation schema (intension). They must hold on all relation states (extensions) of R. However, an FD is not of a particular relation state/instance. A certain relation case for a relation schema R cannot be considered to infer what FDs hold for R. The sole means to find the FDs that hold for R is to cautiously consider what attributes of R mean. FDs are not provable; therefore, they should make it compulsory by a DBMS as a limitation.
- Let X and Y are subsets of the attributes {A₁, A₂... An}.
 - X → Y indicates that there is a 1:M relationship between the set of attributes Y and the set X.
 - X → Y and Y → X indicate that there is a 1:1 relationship between the set Y and the set X.
 - If Y = {C, D}, then X → Y is equivalent to the two functional dependencies X → C. X → D. Thus, X → CD is equivalent to X → C and X → D. Concatenation of sets of attributes denotes the union of the sets, i.e. XY stands for X ∪ Y.
- An FD changes with its real-world meaning.
- X and Y may be composite.
- X and Y could sometimes be mutually dependent on each other; for example., Husband → Wife, Wife → Husband.
- The same Y value may occur in multiple tuples.

8.4.2 Armstrong Axioms—Inference Rules for Functional Dependencies

Let F be a collection of functional dependencies that are stated on a relation schema R. and X and Y: subsets of R. F rationally means a FD X → Y, indicated by F |= X → Y, if every relation occurrence of R that satisfies all FDs in F also satisfies X → Y. The closure of F, denoted by F+, is the set of all FDs that are rationally indicated by F. In other words, it is a collection of all functional dependencies that may be rationally obtained from F.

To determine F+, Armstrong proposed a complete set of rules in 1974 for deriving all possible functional dependencies that are implied by F. The rules are as follows:

Reflexivity Rule:

$$Y \subseteq X \models X \rightarrow Y.$$

It can also be stated as $X \rightarrow X$, i.e., each subset of X is functionally dependent on X . It is the formal statement of *trivial dependencies*.

Normal Forms

Augmentation Rule:

$$X \rightarrow Y \models XZ \rightarrow YZ.$$

We can conclude that if a dependency is able to hold, then it can easily increase its left-hand side.

It should be noted here that the notation XZ is used to denote the collection of all attributes in X and Z and write XZ rather than the more conventional (X, Z) for convenience.

Transitivity Rule:

$$\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$$

It is the ‘most powerful’ inference rule, which is useful in multi-step derivations.

These rules are called **Armstrong’s Axioms**. Each of these rules can be proved from the definition of the functional dependency.

These rules are said to be **complete** in the sense that given a set of FDs, all FDs implied by F can be deduced from the given set using these rules.

These set of inference rules are said to be **sound** in the sense that no additional FDs other than complete set of implied FDs be derived.

It should be possible to simplify an existing set of dependencies to derive a result. Let us illustrate with the aid of an example:

Let us consider a relation $R(X, Y, Z, W, V, U)$

Given FDs are as follows:

1. $XY \rightarrow V$
2. $YV \rightarrow U$
3. $V \rightarrow Z$
4. $ZU \rightarrow W$

A derivation results $XY \rightarrow ZW$ in the following way:

- a. $Y \rightarrow V$ (given: 1)
- b. $XY \rightarrow XY$ (By reflexivity)
- c. $XY \rightarrow Y$ (From decomposition of b)
- d. $XY \rightarrow YV$ (By union of a and c)
- e. $YV \rightarrow U$ (given: 2)
- f. $XY \rightarrow U$ (By transitivity from d and e)
- g. $V \rightarrow Z$ (given: 3)
- h. $XY \rightarrow Z$ (By transitivity from a and g)
- i. $XY \rightarrow ZU$ (Applying union: f and h)

NOTES

- j. $ZU \rightarrow W$ (given: 4)
- k. $XY \rightarrow W$ (By transitivity: i and j)
- l. $XY \rightarrow ZW$ (By union: h and k)

NOTES**8.4.3 Key and Functional Dependency****Key Revisited**

The actual data stored in relations or collected for entry to the database may violate a stated data constraint, an FD.

If this happens, we have to decide whether the FD is incorrect and we need to change our relational model (and our table structures) or a data constraint needs to be enforced by the database processing system.

Functional Dependencies (FD) are types of constraints that are based on *keys*. Let R be a relation schema represented by its set of attributes. A subset k of R, $k \subseteq R$, for all pairs, t_1 and t_2 in tuples r such that $t_1 \neq t_2$ then $t_1[K] \neq t_2[K]$ or no two rows (tuples) have the same value in the attribute(s) K, which is the key. In other words, if there are two attributes X and Y that are legal in relation schema R, $X \rightarrow Y$ implies all pairs of two tuples t_a & t_b such that if $t_a[X] = t_b[X]$ then $t_a[Y] = t_b[Y]$. This allows us to state that k is a super key of R if k implies R, i.e. $k \rightarrow R$.

Therefore, A set of one or more attributes $\{A_1, A_2, \dots, A_n\}$ is a key for a relation R if:

- Those attributes functionally determine all other attributes of the relation. That is, it is impossible for two distinct tuples of R to agree on all of $\{A_1, A_2, \dots, A_n\}$.
- No proper subset of $\{A_1, A_2, \dots, A_n\}$ functionally determines all other attributes of R, i.e. a key must be minimal.

Super key is a ‘superset of keys’. It should be noted that every super key satisfies the first condition for a key but it is not needed to satisfy the second condition, i.e. minimality.

Check Your Progress

1. Define normalization.
2. What does functional dependency implies?

8.5 FIRST NORMAL FORM

The aim of the first normal form (1NF) is to make the structure of a relation simple by making sure that it does not have data aggregates or repeating groups. This means that none of the attribute values can contain a collection of values.

It specifies that the domains of attributes are inclusive of only atomic (straightforward, in dividable) values and that any attribute value in a tuple must be an individual value from the domain of that attribute.

Formal Definition: A relation is in the **first normal form** (1NF) only if all the basic domains hold atomic values simply.

Normalization Procedure: The following table is 1NF.

A	B	C
x	1	{a, b}
z	2	c

Transformation to 1NF:

A	B
x	1
z	2

Method 1: By splitting the relation into new relations.

B	C
1	a
1	b
2	c

Method 2: By adding new tuples in the same relation.

A	B	C
x	1	a
x	1	b
z	2	c

Method 3: By adding new attributes with NULL values.

A	B	C	C'
x	1	a	b
z	2	c	NULL

Illustration 8.1: In the example given below, any one employee has more than one telephone number.

EMP			
EPCODE	ENAME	TEL_NO	
E01	JAYANTA DUTTA	24185462	24181450
E02	KOUSHIK GHOSH	24148618	24731961
E03	SOURAV BASAK	25551189	24725782

Normal Forms

NOTES

This is thus not in 1NF. This can be achieved by ensuring that every tuple defines a single entity by containing only atomic values. One can either reorganize into one relation as in:

NOTES

EMP		
ECODE	ENAME	TEL_NO
E01	JAYANTA DUTTA	24185462
E01	JAYANTA DUTTA	24181450
E02	KOUSHIK GHOSH	24148618
E02	KOUSHIK GHOSH	24731961
E03	SOURAV BASAK	25551189
E03	SOURAV BASAK	24725782

or split into multiple relations as in:

EMP		
ECODE	ENAME	
E01	JAYANTA DUTTA	
E02	KOUSHIK GHOSH	
E03	SOURAV BASAK	

EMP_TEL		
ECODE	TEL_NO	
E01	24185462	
E01	24181450	
E02	24148618	
E02	24731961	
E03	25551189	
E03	24725782	

Reduction to 1NF

8.6 SECOND NORMAL FORM

Fully functional dependency forms the basis of the second normal form. If each nonprime attribute in relation is dependent functionally on the candidate key of the relation, a relation schema is said to be in 2NF. Therefore, if each non-prime attribute in relation is not partly dependent on any key of the relation, the a relation schema is in 2NF.

Formal Definition: Each non-key attribute is fully dependent functionally on the total candidate key and a relation is in the second normal form (2NF) if and only if it is in 1NF.

Normalization Procedure: Test left-hand side attributes as a component of the main key. If an attribute is dependant on parts of a multi-valued key, move it to a different table.

Normal Forms

Suppose, there is a relation R

	A	B	C	D
1	1	c1	d1	
1	2	c2	d2	
2	1	c3	d1	
2	2	c4	d2	

where the composite attribute (A, B) is the main key. Assume that the following functional dependencies exist:

(A, B) → C

i.e., complete functional dependency on the composite keys (A, B).

B → D

i.e., part functional dependency on the composite keys (A, B).

The steps are:

1. A new relation R2 from R are created due to the functional dependency B → D, R2 which contains B and D as attributes. B, the determinant, becomes the key of R2.
2. Minimize the original relation R by eliminating the attribute on the right-hand side of B → D from it. The minimized relation R1 thus has all the original attributes, however, D is absent.
3. Steps 1 and 2 are repeated if multiple functional dependencies stop the relation from converting to a 2NF.

If one relation contains similar determinant with the other relation, the dependent attributes of the relation to be non-key attributes is placed in the other relation for whom the determinant is a key.

B	D	R2	A	B	C
1	d1		1	1	c1
2	d2		1	2	c2
1	d1		2	1	c3
2	d2		2	2	c4

Reduction of 1NF into 2NF

Thus, ‘A relation R is in 2NF if it is in 1NF and every non-key attribute is fully functionally dependent on the primary key.’

NOTES

Reduction of 1NF to 2NF may be represented pictorially as follows:

NOTES

A	B		C	

1NF but Not in 2NF.

A	B	

B	C

Illustration 8.2: Consider the following relation:

EMP

EMPNO	EName	ProjID	ProjName	ProjLocation	Hrs_worked

Where

- EMPNO = Employee Number
 - EName = Name of the Employee
 - ProjID = Project Identification Number
 - ProjName = Name of the Project
 - ProjLocation = Location of the Project
 - Hrs_worked = Hours spent on the Project
- {EMPNO, ProjNum} is the primary key.

Instance of EMP relation is given as follows:

EMPNO	Ename	ProjID	ProjName	ProjLocation	Hrs_worked
E01	JAYANTA DUTTA	P01	HMS	KOLKATA	20
E02	PLNAKIBOSE	P02	SIS	MUMBAI	30
E03	KOUSHIK GHOSH	P01	HMS	KOLKATA	40
E04	GOUTAM DEY	P02	SIS	MUMBAI	30
E05	SOURAV BASAK	P03	RRS	KOLKATA	25

INSERT

We introduce a new employee who has not been assigned to any project yet. We cannot do so unless he/she has been assigned to a project.

EMPNO	Ename	ProjID	ProjName	ProjLocation	Hrs_worked
E01	JAYANTA DUTTA	P01	HMS	KOLKATA	20
E02	PINAKI BOSE	P02	SIS	MUMBAI	30
E03	KOUSHIK GHOSH	P01	HMS	KOLKATA	40
E04	GOUTAM DEY	P02	SIS	MUMBAI	30
E05	SOURAV BASAK	P03	RRS	KOLKATA	25

Normal Forms

NOTES

Entity Integrity Violation: ProjID is a Part of Primary Key

DELETE

If the data about a project is deleted, the information about the employee worked on that project is also deleted. If we delete the data about project P01, information about employees E01 and E03 will be deleted.

EMPNO	Ename	ProjID	ProjName	ProjLocation	Hrs_worked
E01	JAYANTA DUTTA	P01	HMS	KOLKATA	20
E02	PINAKI BOSE	P02	SIS	MUMBAI	30
E03	KOUSHIK GHOSH	P01	HMS	KOLKATA	40
E04	GOUTAM DEY	P02	SIS	MUMBAI	30
E05	SOURAV BASAK	P03	RRS	KOLKATA	25

UPDATE

If project location for a particular project has been changed and we have updated the change only in a single tuple, this will lead to update anomaly due to partial update resulting data inconsistencies.

Although it is normalized to 1NF, the EMP relation still contains storage anomalies. Partial dependency on the primary key leads to violations to the database's integrity and consistency rules.

Recall, the functional dependencies discussed as follows.

The determinant (EMPNO, ProjID) is the composite key of the EMP relation; its value will establish the value of other non-key attributes in a tuple of the relation uniquely. Note that while Hrs_worked is fully functionally dependent on all of (EMPNO, ProjID), Ename is partly functionally dependent on the composite key (as each of them depend only on the EMPNO part of the key only but not on ProjID). ProjName, ProjLocation are only partially functionally dependent on the composite key (as they each depend only on the ProjID part of the key only but not on EMPNO).

Functional Dependencies:

FD1: {EMPNO, ProjID} → Hrs_worked Primary Key

FD2: EMPNO → Ename Partial Dependency

FD3: ProjID → ProjName, ProjLocation Partial Dependency

NOTES

These issues can be kept away by removing partial key dependence in lieu of full functional dependence, and this can be done by making dependencies separate. This is as follows:

ASSIGN: Considering FD1

<u>EMPNO</u>	<u>ProjNum</u>	Hrs_worked
--------------	----------------	------------

EMP: Considering FD2

<u>EMPNO</u>	EName
--------------	-------

PROJECT: Considering FD3

<u>ProjID</u>	<u>ProjName</u>	<u>ProjLocation</u>
---------------	-----------------	---------------------

The source relation is divided into three relations and every resultant relation is no longer dependant on partial key dependencies:

ASSIGN

<u>EMPNO</u>	<u>ProjID</u>	Hrs_worked
E01	P01	20
E02	P02	30
E03	P01	40
E04	P02	30
E05	P03	25

EMP:

<u>EMPNO</u>	EName
E01	JAYANTA DUTTA
E02	PINAKI BOSE
E03	KOUSHIK GHOSH
E04	GOUTAM DEY
E05	SOURAV BASAK

PROJECT

<u>ProjID</u>	<u>ProjName</u>	<u>ProjLocation</u>
P01	HMS	KOLKATA
P02	SIS	MUMBAI
P03	RRS	KOLKATA

There are two relations in the second normal form now. Test the aforementioned operations on the resultant relations.

Update anomaly

Redundant/duplicate tuples are not present in the relation and the updates are completed at one place without worrying for inconsistencies in the database.

Insertion anomaly

Adding a new employee can be done in the EMP relation without concern whether he/she is assigned to a project or not.

Deletion anomaly

Normal Forms

When a tuple is deleted in PROJECT, it does not lead to information loss on details of all the employees.

8.7 THIRD NORMAL FORM

NOTES

Consider a relation R(A,B,C) where the given FDs are $A \rightarrow B$, $B \rightarrow C$. Then an instance of this relation can be as follows:

A	B	C
a ₁	b ₁	c ₁
a ₂	b ₁	c ₁
a ₃	b ₂	c ₂

Check for the redundancies in the relation. As the dependency $B \rightarrow C$ holds we know from the first tuple that when B is b₁, C is c₁. Therefore, we can say that this information b₁, c₁ is repeated in the second tuple. These repetitions had to be made to provide information a₁, b₁ and a₂, b₁.

Check out that this relation is in 2NF and both B and C are fully functionally dependent on the candidate key A.

Normalization Procedure: If we decompose the table such that no non-candidate keys become functionally dependant on another non-candidate key, then the redundancy is removed. So, here we decompose the relation as R₁(A, B) and R₂(B, C). Now the given instance will become

A	B
a ₁	b ₁
a ₂	b ₁
a ₃	b ₂

with A as the candidate key and

B	C
b ₁	c ₁
b ₂	c ₂

with B as the candidate key

Therefore, the redundancy is no longer present and we say that the relation is in 3NF.

NOTES

Illustration 8.3: Let us consider the following EMP relation:

EMPNO	NAME	DESG	SALARY	DEPTNO	DEPTNAME
E01	KOUSHIK GHOSH	MANAGER	25000	D01	ADMINISTRATION
E05	PINAKI BOSE	PROGRAMMER	15000	D03	PROJECT
E06	GOUTAM DEY	FINANCE OFFICER	10000	D02	PERSONNEL
E07	RAJIB HALDER	SYSTEM ANALYST	20000	D03	PROJECT

Consider again operations that we may want to do on the data.

UPDATE

Can we change the employees servicing in department ‘D03’, i.e., in PROJECT department. In this department, there are many employees belonging to department ‘D03’ (for example, RAJIB HALDER and PINAKI BOSE). Thus, we must make certain that all tuples are updated or there will be problem with database inconsistency.

INSERT

Suppose we want to add a new employee in the EMP relation with empno as its primary key. The new tuple must also include correct information about the department where he will be working. If not, the data will become inconsistent.

EMPNO	NAME	DESG	SALARY	DEPTNO	DEPTNAME
E01	KOUSHIK GHOSH	MANAGER	25000	D01	ADMINISTRATION
E05	PINAKI BOSE	PROGRAMMER	15000	D03	PROJECT
E06	GOUTAM DEY	FINANCE OFFICER	10000	D02	PERSONNEL
E07	RAJIB HALDER	SYSTEM ANALYST	20000	D03	PROJECT
E08	JAYANTA DUTTA	PROGRAMMER	12000	NULL	NULL

If we want to add a new department and no employees are assigned there, then as we do not know the information about the employees, we will have to put NULLs as their attributes including the primary key, which is an absolutely not permitted.

EMPNO	NAME	DESG	SALARY	DEPTNO	DEPTNAME
E01	KOUSHIK GHOSH	MANAGER	25000	D01	ADMINISTRATION
E05	PINAKI BOSE	PROGRAMMER	15000	D03	PROJECT
E06	GOUTAM DEY	FINANCE OFFICER	10000	D02	PERSONNEL
E07	RAJIB HALDER	SYSTEM ANALYST	20000	D03	PROJECT
NULL	NULL	NULL	NULL	D01	SALES

DELETE

What happens if we were to delete the employee data from the relation? In this case, information about the department is also lost, which must be avoided. For example, if we want to delete information of employee whose employee code is

E06 in the EMP instance shown above, then we would lose information concerning the department D02.

Normal Forms

EMPNO	NAME	DESC	SALARY	DEPTNO	DEPTNAME
E01	KOUSHIK GHOSH	MANAGER	25000	D01	ADMINISTRATION
E05	PINAKI BOSL	PROGRAMMER	15000	D03	PROJECT
E07	RAJIB HALDER	SYSTEM ANALYST	20000	D03	PROJECT

Instance of EMP after deleting the tuple of employee e06

The above example makes it clear that in spite of a relation existing in 2NF, issues can still occur and they should be removed. They need to be normalized further, i.e., there is a need for a third normal form to remove these anomalies. Examine the functional dependencies of the above example, the presence of ‘transitive’ dependencies are evident.

Here $\text{empno} \rightarrow \text{deptname}$ is only an transitive or indirect dependency. It is treated as indirect as $\text{empno} \rightarrow \text{deptno}$ and $\text{deptno} \rightarrow \text{deptname}$ and thus $\text{empno} \rightarrow \text{deptname}$

NOTES

8.8 BCNF

Check the following example. Consider a relation R (A, B, C) where the given FDs are $AB \rightarrow C$, $C \rightarrow B$. Then an instance of this relation can be as follows:

A	B	C
A ₁	b ₂	c ₁
A ₄	b ₂	c ₁
A ₁	b ₃	c ₂
A ₃	b ₃	c ₃

Check for the redundancies in the relation. As the dependency $C \rightarrow B$ holds, we know from the first tuple that when C is c_1 , B is b_2 . Therefore, we can say that this information c_1, b_2 is repeated in the second tuple. These repetitions had to be made to provide information a_1, b_2, c_1 and a_4, b_2, c_1 .

Check out that this relation is in 3NF and as C is the only candidate key in the relation, there is no chance that a non-candidate key is dependant on another non-candidate key.

Normalization Procedure

If we decompose the table such that no part of the candidate key becomes functionally dependant on a non-candidate key, then the redundancy is removed. Therefore, we decompose the relation as $R_1(A, C)$, $R_2(B, C)$. Now the given instance will become as follows:

NOTES

A	C
a ₁	c ₁
a ₄	c ₁
a ₁	c ₂
a ₃	c ₃

with AC as the candidate key and

B	C
b ₂	c ₁
b ₃	c ₂
b ₃	c ₃

with C as the candidate key

Therefore, the redundancy is no longer present and we say that the relation is in BCNF.

May be, one question have come to your mind. **What is the problem with the redundancy?** Total size of the database does not decrease after the removal of the redundancy. On the contrary, it increases. Actually, this question is answered right at the beginning of the discussion of DBMS. The problem with the redundancy is that it may lead to inconsistency. One can very easily change one occurrence of an instance and forget changing the other occurrence.

How to check that a relation is in BCNF

For a relation R to be in BCNF, one of the following conditions should hold for all dependencies: A→B in a relation where A and B are a set of attributes within the relation.

- The dependency A→B is trivial.
- A is a super key of the relation R.

Now if none of these conditions holds for at least one of the dependencies A→B in R, we say that R is not in BCNF.

Illustration 8.4: When a relation has more than one candidate key, anomalies may result though the relation is in 3rd **Normal Form**. Anomalies that occur when a table is unsuccessful to possess the property that each determinant is a candidate key leads to the Boyce–Codd Normal Form (BCNF). This example cites the failure to possess this property.

ECODE	PID	P_LEADER
E01	P01	MG
E02	P02	JDR
E03	P03	SS
E04	P02	JDR
E05	P01	MG

Where

ECODE = Employee Code

PID = Project Identification Number

P_LEADER = Project Leader

Normal Forms

NOTES

The only assumption is that a project leader manages every project. This is in 3NF because it has no partial functional dependencies and no transitive dependencies. It doesn't have the much needed property that each determinant be a candidate key. Name the determinants in R? A pair of attributes is determinant, ECODE and PID. Every one distinctive pair of values of ECODE and PID establishes a value for the attribute which is unique, P_LEADER. Another determinant is the pair, ECODE and P_LEADER, which establishes values of the attribute that are unique, PID. Still another determinant is the attribute, P_LEADER, for each different value of P_LEADER determines a unique value of the attribute, PID. These observations about the relation R correspond to the real-world facts that each employee has a single project leader (P_LEADER) for each of his or her project, and just one project leader (P_LEADER) manages each project.

These three determinants need to be examined whether they are candidate keys or not. The response is that the pair, ECODE and PID, is a candidate key, since each pair identifies a row in R uniquely. Similarly, the pair, ECODE and P_LEADER, is a candidate key. Because the value MG appears in two rows of the P_LEADER column, the determinant, P_LEADER, is not a candidate key. Therefore, the relation R is unsuccessful in fulfilling the condition that every determinant in it is a candidate key.

Even though R is in 3NF, there are still anomalies in it.

INSERTION: If we want to add a new project with a project leader, we cannot until we have an employee assigned in that project.

DELETION: If we delete employee E03, we lose all information that SS manages the project P03.

UPDATE: If AC replaces MG as project leader of project P01, we have to update multiple rows.

The problem occurs because there is a determinant that is not a candidate key

Now we formulate the whole thing discussed above.

R (ECODE, PID, P_LEADER)

FDs: ECODE, PID → P_LEADER

P_LEADER → PID.

By decomposition we get

R1 (ECODE, PID)

R2 (P_LEADER, PID)

Normal Forms

NOTES

R1	ECODE	PID	R2	P_LEADER	PID
	E01	P01		MG	P01
	E02	P02		JDR	P02
	E03	P03		SS	P03
	E04	P02			
	E05	P01			

Check Your Progress

3. What is the main aim of First Normal Form (1NF)?
4. Write the condition for a relation schema to be in 2NF.

8.9 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. McFadden has defined normalization in his book *Modern Database Management*. According to him, ‘Normalization is the process of decomposing relations [tables] with anomalies to produce smaller, well-structured relations.’
2. A Functional Dependency (FD) is derived from mathematical theory. It implies the dependency of values of a single attribute or a collection of attributes, on another attribute or a collection of attributes.
3. The aim of the First Normal Form (1NF) is to make the structure of a relation simple by making sure that it does not have data aggregates or repeating groups. This means that none of the attribute values can contain a collection of values.
4. If each nonprime attribute in relation is dependent functionally on the candidate key of the relation, a relation schema is said to be in 2NF. Therefore, if each non-prime attribute in relation is not partly dependent on any key of the relation, then a relation schema is in 2NF.

8.10 SUMMARY

- Database design is essentially a business problem, not a data problem.
- Data modelling is primarily the result of a thorough understanding of information about an enterprise.
- A technique called normalization is used to analyse the given relational schemas to achieve the desirable properties of minimizing redundancy as well as inconsistency besides minimizing the insertion, deletion, and update anomalies.

- Normalization is a bottom-up approach to database design that begins by examining the relationships between attributes.
- The process of normalization is a formal method that identifies relational schemas based upon their primary or candidate keys and the functional dependencies that exist among their attributes.
- Decomposition is the process of breaking a relation into multiple relations.
- A Functional Dependency (FD) is derived from mathematical theory. It implies the dependency of values of a single attribute or a collection of attributes, on another attribute or a collection of attributes.
- The aim of the first normal form (1NF) is to make the structure of a relation simple by making sure that it does not have data aggregates or repeating groups. This means that none of the attribute values can contain a collection of values.
- If each nonprime attribute in relation is dependent functionally on the candidate key of the relation, a relation schema is said to be in 2NF.
- For a relation R to be in BCNF, one of the following conditions should hold for all dependencies: $A \rightarrow B$ in a relation where A and B are a set of attributes within the relation.
 1. The dependency $A \rightarrow B$ is trivial.
 2. A is a super key of the relation R.

Normal Forms

NOTES

Now if none of these conditions holds for at least one of the dependencies $A \rightarrow B$ in R, we say that R is not in BCNF.

8.11 KEY WORDS

- **Normalization:** A technique used to analyse the given relational schema to reduce redundancy, inconsistency, and deletion, updation and insertion anomalies.
- **1NF:** A relation is in the 1NF if all the basic domains hold atomic values.
- **2NF:** A relation schema is said to be in 2NF if each non-prime attribute in a relation is dependent functionally on the candidate key of the relation.

8.12 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What is normalization?
2. What is Functional Dependency (FD)?

NOTES

3. What is full functional dependency? Give an example.
4. What is partial dependency? Give an example.
5. What is transitive dependency? Give an example.
6. What is trivial dependency? Give an example.
7. What are Armstrong's axioms? Give an example.
8. What are the advantages of normalized relations over unnormalized relations?
9. What are the main rules for normalization?
10. What is an insertion anomaly?
11. What do you mean by a deletion anomaly?

Long Answer Questions

1. Discuss the role of functional dependency in DBMS.
2. Discuss the different types of FD with the help of examples.
3. Describe in your own words the purpose of the process of normalization.
4. Discuss the basic rules for 1NF and 2NF considering only primary keys. Define any terms that you feel necessary for these definitions.
5. Describe Second Normal Form. Give an example of a relation in 1NF but not in 2NF. Transform the relation into relations in 2NF.
6. Describe Third Normal Form. Give an example of a relation in 2NF but not in 3NF. Transform the relation into relations in 3NF.
7. How does the definition for BCNF differ from the general definition for 3NF?
8. Why is BCNF a stricter normal form than 3NF? Explain.

8.13 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.

- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

Normal Forms

NOTES

UNIT 9 JOIN

NOTES

Structure

- 9.0 Introduction
- 9.1 Objectives
- 9.2 Lossless Join and Dependency Preservation
 - 9.2.1 Dependency Preservation and Multi-Valued Dependencies
- 9.3 Schema Refinement in Database Design
- 9.4 Fourth Normal Form (4NF)
- 9.5 Answers to Check Your Progress Questions
- 9.6 Summary
- 9.7 Key Words
- 9.8 Self Assessment Questions and Exercises
- 9.9 Further Readings

9.0 INTRODUCTION

A lossless join decomposition in database design is a decomposition of a connection into relationships. In such a way that the original bond is restored by a natural union of the two smaller connections. This is critical in the secure removal of redundancy from databases while maintaining the original knowledge.

Decomposition is the process of breaking a relation into multiple relations. Decomposition of relations must make certain that from the decomposed relations, an original relation can be reconstructed.

In this unit, you will study about the joins, lossless join decomposition, dependency preservation decomposition, schema refinement in database design, multi-valued dependencies, and FORTH normal form.

9.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the lossy and lossless decomposition
- Understand the dependency preservation
- Describe schema refinement in database design
- Describe the basics of multivalued dependencies
- Explain the fourth normal form

9.2 LOSSLESS JOIN AND DEPENDENCY PRESERVATION

It is clear that we cannot always perform decomposition, which is in BCNF and preserves dependencies. We may only be able to get to 3NF, which satisfies lossless join and dependency preservation. It may not be possible to achieve all the following three goals of database design with functional dependencies:

- BCNF
- Lossless Join
- Dependency Preservation

We may have only two choices, either BCNF or Dependency Preserving 3NF.

We must then decide whether to leave it there and build protection for update anomalies or to decompose even further with the resulting loss of performance. There is no concrete answer to this situation. First we have to estimate how important the FDs are, which are not preserving in BCNF decomposition. If the FDs in question are important to become unenforceable, we have to leave the schema in 3NF. If they were not so important that their unenforceability would not cause serious problems, then the BCNF decomposition would be beneficial to removing further redundancy.

Lossy and Lossless Decomposition

Loss of information due to decomposition is called lossy decomposition. When all information found in the original database is preserved after decomposition then it is termed as lossless decomposition or non-lossy decomposition. One of the drawbacks of decomposition into two or more relational schemas or tables is that some information is lost during retrieval of original relation or table. It is not easy to know about the starting table content, the loss of information due to decomposition and the subsequent join operation. Hence, this phenomenon is termed as lossy decomposition or lossy-join decomposition.

Typically, the concept of lossless-join decomposition is essential in removing redundancy safely from databases while preserving the original data. Lossless-join decomposition is also sometimes termed as non-additive. If a relation R is decomposed into relations R_1 and R_2 , then there will be a lossless-join if $R_1 \cup R_2 = R$.

Let R be a relation schema and F be a set of functional dependencies on R. Also, let R_1 and R_2 form a decomposition of R. The decomposition is a lossless-join decomposition of R if at least one of the following functional dependencies is in F^+ where F^+ stands for the closure for every attribute in F:

- $R_1 \cap R_2 \subseteq R_1 - R_2$
- $R_1 \cap R_2 \subseteq R_2 - R_1$

NOTES

NOTES

The process of database design also involves the decomposition. Decomposition is somehow similar to the projection operation in RDBMS. As you learnt earlier that in projection operation required columns are selected and the rest of the columns will be discarded, i.e., they are placed in another table logically.

To completely understand the concept of decomposition you must know about redundancy. Redundancy refers to the unwanted and uncontrolled duplicity of data. Redundancy also leads to the deletion of loss of consistency and integrity of data.

Loss of information during decomposition is called lossy decomposition. If the decomposition leads to loss of information then it cannot be used for redundancy. Alternatively, you can say that lossy decomposition is not acceptable in any case. Therefore, the decomposition should be performed in such a way that any information will not be lost either data of the table or the inter-relationships between the various data items of tables.

You can easily figure out if the decomposition is lossy. For this you need to check if, after performing decomposition, you can resemble the data in the original form, the process called as recomposition. If you are not able to get the data in the original form then it is meant that you have lost the information from the original database design or you performed a lossy decomposition. If you are able to reassemble the split tables so that the original data will be recreated, then the decomposition is desired and successful.

If $r = \{R_1, R_2, \dots, R_k\}$ is a decomposition of a relation scheme R and D is a set of dependencies then the decomposition is the lossless decomposition with respect to D , if for every relation r for R satisfies D such that:

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \dots \bowtie \pi_{R_k}(r)$$

i.e., $r = m_p(r)$.

If $r \subset \pi_{R_1}(r) \bowtie \dots \bowtie \pi_{R_n}(r)$, then the decomposition is lossy.

9.2.1 Dependency Preservation and Multi-Valued Dependencies

It is evident that complexities can occur when an entity has to represent multi-valued attributes of an entity type from an E-R model in the relational model. If all information about such an entity is represented in an individual relation, it is essential to replicate all information rather than the multi-valued attribute that represents all information about the entity. Alternatively, this multi-valued information can be represented in a separate relation. The circumstances would become worse if an entity has multiple multi-valued attributes and are represented in one relation by various tuples for every entity instance. In such a situation, each value of the one multi-valued attribute emerges with each value of the second multi-valued attribute to keep it consistent.

A multi-valued dependency forms the basis of the fourth and fifth normal forms. The following example illustrates the concept of multi-valued dependency. Fourth normal form will be discussed later.

Join

PROGRAMMER (emp_name, languages, qualifications)

This relation comprises two multi-valued attributes of the entity programmer, languages and qualifications. Functional dependencies are absent.

The attributes, languages and qualifications are supposedly not dependent on one another. If languages and qualifications are considered separately, then there are two relationships, i.e., one between employees and languages, and the other between employees and qualifications. This means that one programmer may have multiple qualifications and may have the knowledge of multiple programming languages. However, several programmers may have one qualification and the knowledge of only one programming language.

The above relation is, therefore, in 3NF (even in BCNF) but it still has some disadvantages. Suppose a programmer has several qualifications (BCA, MCA, etc.) and is proficient in several programming languages. How should this information be represented? There are several possibilities.

NOTES

Emp_name	qualifications	languages
MANAS GHOSH	BCA	C
MANAS GHOSH	BCA	C++
MANAS GHOSH	BCA	COBOL
MANAS GHOSH	MCA	C
MANAS GHOSH	MCA	C++
MANAS GHOSH	MCA	COBOL

Emp_name	qualifications	languages
MANAS GHOSH	BCA	NULL
MANAS GHOSH	MCA	NULL
MANAS GHOSH	NULL	C
MANAS GHOSH	NULL	C++
MANAS GHOSH	NULL	COBOL

Emp_name	qualifications	languages
MANAS GHOSH	BCA	C
MANAS GHOSH	MCA	C++
MANAS GHOSH	NULL	COBOL

Other variations may also be possible, but there are some disadvantages in these variations. In case of repeated information, the same problem occurs which specifies

repeated information and anomalies that take place when 2NF or 3NF conditions are violated. These issues can be resolved by decomposing a relation as follows:

P1:

NOTES

Emp_name	qualifications
MANAS GHOSH	BCA
MANAS GHOSH	MCA

P2:

Emp_name	languages
MANAS GHOSH	C
MANAS GHOSH	C++
MANAS GHOSH	COBOL

The basis of the above decomposition is the concept of Multi-Valued Dependency (MVD). Functional dependency $A \rightarrow B$ relates one value of A to one value of B while multi-valued dependency $A \rightarrow\rightarrow B$ defines a relationship in which a set of values of attribute B is determined by a single value of A.

The concept of multivalued dependencies was developed to provide a basis for decomposition of relations like the one above.

Let us now define the concept of multi-valued dependency. The multi-valued dependency $X \rightarrow\rightarrow Y$ is said to hold for a relation $R(X, Y, Z)$ if for a given set of values (a set of values if X is more than one attribute) for attributes 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.

X and Y are disjoint subsets of R, and let $Z = R - (XY)$.

A relation $r(R)$ satisfies the multi-valued dependency (MVD) $X \rightarrow\rightarrow Y$ if, $t_1(X) = t_2(X)$ and there exists a tuple t_3 in r with $t_3(X) = t_1(X)$, $t_3(Y) = t_1(Y)$, and $t_3(Z) = t_2(Z)$.

Trivial Multivalued Dependencies

The trivial type is the most common kind of multi-valued dependency. It has no relation with the fourth normal form apart from a few of the resulting tables from the process that has trivial multivalued dependencies.

When a table comprises of only two fields of which one is multi-valued and both fields form the primary key is termed as a trivial multi-valued dependency. In the example, it can be noticed that the double-headed arrow indicates an MVD or any type.

PARENT_NAME $\rightarrow\rightarrow$ CHILD_NAME

PARENT_NAME	CHILD_NAME
SUMAN	PRIYAM
SUMAN	SANDIP
SUMAN	RAJA

Join

NOTES

$X \rightarrow\!\!\rightarrow Y$ is a trivial MVD if Y is a subset of X or $R = X \cup Y$.

An FD $X \rightarrow Y$ is an MVD $X \rightarrow\!\!\rightarrow Y$ with the additional restriction that at most one value of Y is associated with each value of X .

When two multivalued dependencies are present and they are not dependant on the other MVD, it is termed a non-trivial multi-valued dependency.

Join Dependencies

SQL joins are used to query data from two or more tables based on a relationship between certain columns in these tables.

Joins between Tables

A query that accesses multiple rows of the same or different tables at one time is called a join query.

Storage in multitable relations and hence multitable queries involving more than one table are fundamental to relational databases.

To combine relations, i.e., to perform a join, we have to select data from the tables and relate the tables to each other with conditions on some attributes (often keys attributes).

The `JOIN` keyword is used in an SQL statement to query data from two or more tables, based on a relationship between certain columns in these tables.

Tables in a database are often related to each other with keys.

Check Your Progress

1. What do you understand by decomposition?
2. What is lossless decomposition or non-lossy decomposition?

9.3 SCHEMA REFINEMENT IN DATABASE DESIGN

Schema refinement is a terminology used for improving tables. It is the last step with classic workloads before considering the physical design. Following are the various steps for building a database.

1. Requirement analysis which identify the user requirements
2. Conceptual design involves high-level description, often using E/R diagrams

Join

NOTES

3. Logical design is the design from graphs to tables that consists of relational schema.
4. Schema refinement results in checking tables for anomalies and redundancies.

Consider an example which shows the case of anomalies and redundancies. In the following table, we are assuming the client's name (client_info) as primary key.

Emp_ID	First_Name	Last_Name	Client_info	Joining_date
EMP101	John	Smith	Jarbin H.	3/12/17
EMP102	Martin	John	Clark L.	4/11/16
EMP103	Sunchty	Trag	Marlin G.	4/8/18
EMP104	Ristag	Jonoy	Nautj P.	3/7/18

This table shows the information regarding employees and their clients.

- **Inserting data:** if we want to insert data, the following points should be kept in mind:
 - o Every row requires a mandatory entry in the client field.
 - o It is not possible to insert data for newly hired sales representative till they have been allotted to one or more clients.
 - o Consider if sales representatives are in a training process, even if they've been already hired. But they can't really join the database. The reason behind is that they required to have a delegated client, till the "dummy" clients are formed.
- **Updating Data:** If we want to update data, the following points should be kept in mind:
 - o For each client, the sales representative name is repeated.
 - o In case, if for a given client, we misspelled the name of client as mergin instead of merlin. We can't edit that without affecting all the sales reps called merlin.
- **Deleting Data:** If we want to delete data, the following points should be kept in mind:

If we want to **delete data**, what will be Marlin doesn't have a client anymore because he has gone abroad.so we are required to:

- We have to create a dummy client.
- Wrongly showing him with a client he has no longer handled.
- Delete Marlin's record, in case he's still an employee.
- There can't be client as a null because as we know that the primary field keys cannot store null.

The main problem with schema refinement is redundancy. In order to handle these problems, there is a concept of functional dependencies which shows the

relationship that occurs only when one attribute exclusively determines another attribute. So, it is a new type of restriction between two attributes. Consider, R is a relation having attributes A and B. Also assume that there is a functional dependency $A \rightarrow B$. Here B is functionally dependent on A.

Join

For example:

Consider a situation where the designer did not notice dependencies between columns:

- Data (stu_ID, stu_Name, stu_address, course_ID, course_Name, grade)

But the better arrangement is given below:

- Student(stud_ID, stud_Name, stu_address)
- Course(course_ID, course_Name)
- Enrolled(stud_ID, course_ID, grade)

So to pass from one to the other, there is a schema refinement using functional dependencies.

A student can be represented through his stud_ID. Each student has his own address. So we can say that stud_ID defines address. This statement can be written as:

- Stud_ID \rightarrow address

In the previous example, there were following FDs:

- Stud_ID \rightarrow stud_Name, stu_address
- Course_ID \rightarrow course_Name
- Stud_ID, course_ID \rightarrow grade

Properties of Functional Dependencies

There are different properties of functional dependencies. Here consider A, B, C are attributes which are belonging to a table R:

- **Transitivity:** if we consider that $A \rightarrow B$ and $B \rightarrow C$, then it's clear that $A \rightarrow C$.
- **Reflexivity:** consider, if B is a subset of A, then $A \rightarrow B$.
- **Augmentation:** if $A \rightarrow B$, then for any C, there will be, $A, C \rightarrow B, C$.
- **Union:** if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow B, C$.
- **Decomposition:** if $A \rightarrow B, C$ then $A \rightarrow B$ and $A \rightarrow C$.

Here, the first three properties are called the Armstrong's Axioms.

Consider, if F is a set of functional dependencies then it is clear that F^+ is the set of all FDs logically implied by F. Logically implied means another way of saying that is obtained from the properties of functional dependencies. F^+ is also called the closure of the set of functional dependencies

NOTES

For example:

Consider we have the following set of FDs, can we achieve that $A \rightarrow H$ is logically implied. So here:

NOTES

$A \rightarrow B$
 $A \rightarrow C$
 $C, G \rightarrow H$
 $C, G \rightarrow I$
 $B \rightarrow H$

So check, which properties are appropriate to our case:

- By the transitivity property, if $X \rightarrow Y$ and $Y \rightarrow Z$ then there is $X \rightarrow Z$.
- So here, we have $A \rightarrow B$ and $B \rightarrow H$.
- So, by transitivity rule, $A \rightarrow H$ is logically implied.
- The union rule, $CG \rightarrow HI$.
- The augmentation rule, $AG \rightarrow I$ by noticing that $A \rightarrow C$ holds, and then $AG \rightarrow CG$
- Transitivity rule, $AG \rightarrow$

Consider another example:

- Does $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D \rightarrow E\}$ imply $A \rightarrow E$?
- Is $A \rightarrow E$ in the closure F^+ ?
- Is E in A^+ ?

Consider here, A^+ the attribute closure of A w.r.t. F . This also helps to find out if $A \rightarrow E$ which is logically implied. For this some steps are given below:

1. Consider here, we are creating a temporary attribute closure of A which is called TMP. So assume, $TMP = A$.
2. Take 1st dependency that is, $F, A \rightarrow B$.
3. If A in the TMP, then $TMP = A$, and continue.
4. After that, union B with the current TMP that is A . here output will be AB .
5. Now consider 2nd given dependency, $B \rightarrow C$.
6. If B in the TMP, then we can say that, AB in the TMP and continue.
7. After that, union C with the current TMP, AB . The output will be the new TMP, ABC .
8. Now take the third dependency, $C \rightarrow D \rightarrow E$.
9. If CD in the TMP, then ABC in the current TMP, and finally stop.
10. So the attribute closure of A is then $A^+ = TMP = \{A, B, C\}$

To check if $A \rightarrow E$ is in the closure F^+ , we can determine that E is NOT in A^+ , then $A \rightarrow E$ is NOT in F^+ .

The above steps can be simplified into an algorithm:

Join

Algorithm

- Step 1:** Assume that the input of FDs is the first element of temporary attribute closure TMP.
- Step 2:** Assume the dependency $X \rightarrow Y$ of the given set of FDs.
- Step 3:** Now check whether X part of TMP, if yes then continue to step 4 and if not then continue to step 5.
- Step 4:** Yes: Union TMP is with Y.
- Step 5:** case No: means the attribute closure = TMP.
- Step 6:** if an attribute is in attribute closure, then it's logically implied.

NOTES

9.4 FOURTH NORMAL FORM (4NF)

The fourth normal form has a relation that possesses information about an individual entity. If a relation has multiple one multi-valued attribute, it should be decomposed so that problems can be eliminated with the multi-valued facts.

Intuitively R is in 4NF if all dependencies are a result of keys. When multivalued dependencies are present, a relation should not have multiple independent multi-valued attributes. The decomposition of a relation to attain 4NF would lead to minimizing redundancies and anomalies can be avoided.

Formal Definition: A relation R is in the **fourth normal form** (4NF) only if a multi-valued dependency exists in the R , say $A \rightarrow\rightarrow B$, then all attributes of R are dependent functionally on A .

A relation schema R is in 4NF with respect to a set F of functional and multivalued dependencies if for every non-trivial multi-valued dependency $X \rightarrow\rightarrow Y$ in F^* , X is a key for R .

Alternative Definition: A relation schema R is in 4NF with respect to a set of functional and multivalued dependencies if for every multi-valued dependency $X \rightarrow\rightarrow Y$ in F^* , at least one of the following holds:

- Y is a subset of X (trivial MVD)
- $R = X \cup Y$ (trivial MVD)
- X is a super key for R

Practical Rule: ‘Isolate Independent Multiple Relationships’, i.e. no table may contain two or more 1:n or N:M relationships that are not directly related.

Check Your Progress

3. What is schema refinement?
4. Write the condition for a relation to be in 4NF.

9.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

NOTES

1. Decomposition is the process of breaking a relation into multiple relations. Decomposition of relations must make certain that from the decomposed relations, an original relation can be reconstructed.
 2. When all information found in the original database is preserved after decomposition then it is termed as lossless decomposition or non-lossy decomposition.
 3. Schema refinement is a terminology used for improving tables.
 4. A relation R is in the fourth normal form (4NF) only if a multi-valued dependency exists in the R, say $A \rightarrow\!\!\rightarrow B$, then all attributes of R are dependent functionally on A.
-

9.6 SUMMARY

- We cannot always perform decomposition, which is in BCNF and preserves dependencies. We may only be able to get to 3NF, which satisfies lossless join and dependency preservation.
 - Decomposition is the process of breaking a relation into multiple relations.
 - Loss of information due to decomposition is called lossy decomposition. When all information found in the original database is preserved after decomposition then it is termed as lossless decomposition or non-lossy decomposition.
 - A multi-valued dependency forms the basis of the fourth and fifth normal forms.
 - Schema refinement is a terminology used for improving tables.
 - A relation R is in the fourth normal form (4NF) only if a multi-valued dependency exists in the R, say $A \rightarrow\!\!\rightarrow B$, then all attributes of R are dependent functionally on A.
-

9.7 KEY WORDS

- **Decomposition:** It is the process of splitting a relation into two or more relations.
- **Lossless decomposition:** It is the decomposition process in which all information found in the original database is preserved.

9.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Discuss the concept of lossy and lossless decomposition.
2. How will you find that a relation is in fourth normal form?

Long Answer Questions

1. Explain the term multi-valued dependencies.
2. Describe the schema refinement in database design.
3. Explain how the multi-valued dependencies form the basis of fourth normal form.

NOTES

9.9 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

BLOCK - IV

TRANSACTION

UNIT 10 BASIC CONCEPT OF TRANSACTION

Structure

- 10.0 Introduction
- 10.1 Objectives
- 10.2 Transaction Management
 - 10.2.1 Transaction
 - 10.2.2 Transaction Processing Steps
 - 10.2.3 Interleaved vs Simultaneous vs Serial Transaction
 - 10.2.4 Transaction Action
 - 10.2.5 Transaction States and Additional Operations
 - 10.2.6 Concept of System Log
 - 10.2.7 Commit Point of a Transaction
 - 10.2.8 Acid Properties of a Transaction
 - 10.2.9 Concurrent Execution of Transactions
 - 10.2.10 Motivation for Concurrent Execution of Transactions
- 10.3 Concurrency Control
 - 10.3.1 The Lost Update Problem
 - 10.3.2 Uncommitted Dependency—The Dirty Read Problem
 - 10.3.3 Unrepeatable Read or Inconsistent Retrievals Problem
 - 10.3.4 Phantom Reads; 10.3.5 Schedule
 - 10.3.6 Serializability; 10.3.7 Recoverability
- 10.4 Answers to Check Your Progress Questions
- 10.5 Summary
- 10.6 Key Words
- 10.7 Self Assessment Questions and Exercises
- 10.8 Further Readings

10.0 INTRODUCTION

There are three closely-related functions designed to ensure that a database is reliable and remains in a consistent state. These three functions are transaction management, concurrency control, and recovery. The reliability and consistency of the database must be maintained even in the presence of failures in both hardware and software components, and when multiple users access the database. Although each of these three functions can be examined independently, they are mutually dependent. Both concurrency control and recovery are required to protect the database from data inconsistencies and loss of data. Most DBMS allow concurrent access to the database. If these concurrent operations are not controlled, the

various accesses may interfere with one another and the database could arrive at an inconsistent state. To prevent this from occurring, the DBMS implements a concurrency control protocol that prevents concurrent accesses from interfering with one another. Recovery is the process of restoring the database to a correct state following some type of failure. The failure may have occurred in either/both hardware or software. Malicious corruption or destruction of a component of the database system is also a problem associated with the reliability and consistency of the database. While recovery from such an event certainly must be handled by the DBMS, its prevention falls under DBMS security issues. Therefore, this unit will discuss transaction management, concurrency control and database security and authorization.

A transaction is a small logical unit of program that possibly results in modification of content of database. It must maintain the atomicity, consistency, isolation and durability generally known as ACID properties. You will also learn about serializability and recoverability in transactions.

In this unit, you will study about the basic concept of transaction, its transition states, implementation of atomicity and durability (ACID properties), concurrent executions, serializability, recoverability, and implementation of isolation and testing for serializability.

10.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the concept of transaction in DBMS
- Discuss the various transaction processing steps
- Explain the ACID properties of transaction
- Describe the process of concurrency control
- Explain the serializability and recoverability in scheduled transactions

10.2 TRANSACTION MANAGEMENT

Transaction management is the process of controlling the execution of transactions.

10.2.1 Transaction

A transaction is a logical unit of work in a DBMS that includes one or more database access operations. A transaction may involve an arbitrary number of operations on the database which brings about changes in the state of the database while preserving system consistency. It may be specified using SQL in an interactive way or it may be embedded or contained in an application program. In a database environment, an application program is generally considered as a series of transactions with non-database processing taking place in between the operations on the database.

Basic Concept of Transaction

NOTES

NOTES

Credit card approvals, hotel check-ins, registrations (for courses), billing, phone calls, ATM transactions and supermarket canning are some examples.

Formal definition: It is a unit of work involving read, write or control operations on database elements that must be completed as a whole or not at all.

Let $\text{Read}(X)$ and $\text{Write}(X)$ denote read and write operations on a set of database items $X = \{A, B, C, \dots\}$.

Then a sequence of actions on X is given by

$$S = \{ [\text{Read}(X)]^*; [\text{Write}(X)]^* \}$$

A Transaction is a sequence followed by COMMIT or a ROLLBACK

$$T = \{ S^*, \{ \text{ROLLBACK} | \text{COMMIT} \} \}$$

COMMIT operation signals the successful end of a transaction. It informs that the transaction has been successfully completed and the database is in a consistent state and all the updates are permanent.

ROLLBACK operation, in contrast, signals the unsuccessful-end-operation of a transaction.

10.2.2 Transaction Processing Steps

It is a must to understand the steps followed during a transaction execution for planning and implementing the custom applications. Also, the database administrator should have a clear idea of the steps as it is helpful in understanding and tuning the database parameters and processes. It should be very clear to the readers that the discussion is only valid for the normal transactions. Other transactions—such as discreet and distributed—are treated in a different manner. These transactions are out of our discussion. The steps in processing are as follows:

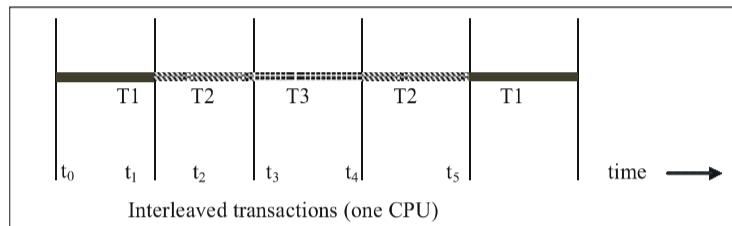
1. Entering the DDL/DML statements.
2. Assigning the ROLLBACK segment.
3. Optimizing the statement.
4. Generating the execution plan by the optimizer.
5. Manipulating/returning data following the execution plan.
6. Looping the above steps from 1 to 5.

10.2.3 Interleaved vs Simultaneous vs Serial Transaction

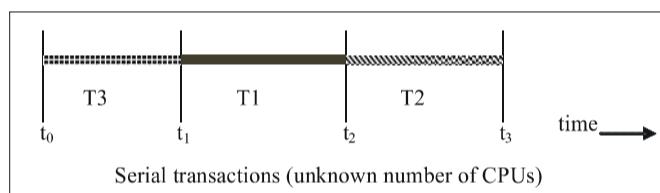
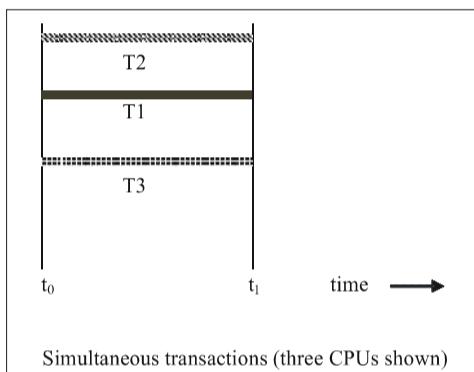
Interleaved transactions take place when two or more transactions are processed concurrently with only one transaction at a time making progress toward completion. This typically occurs in a single CPU environment using multi-programming techniques. Simultaneous transactions take place when two or more transactions are parallelly processed. The number of CPUs online determines the number of transactions progressing. Serial transactions may occur in either a single CPU or multiple CPU environments. They occur when a transaction is executed from start

to finish before any other transaction may begin execution. The following illustrations should clarify the differences in the three basic types of transactions:

Basic Concept of Transaction



NOTES



10.2.4 Transaction Action

Any transaction can access the database either by reading a database item or writing a database item.

The main area of concern with write operations is when to reflect the changes made by a transaction into the database. In many cases, the write-back protocol is handled by the recovery portion of the DBMS possibly working in conjunction with the OS (operating system). In general, there are two basic approaches to handling the write-back operation—the pessimistic approach and the optimistic approach. The pessimistic approach basically forces the write-back to occur immediately after the transaction performs the write. The problems with this approach are as follows:

- (i) If the transaction uses the same database item again, the buffer will need to be reloaded from the disk memory system
- (ii) If the transaction ultimately aborts, the write operation must be undone.

Alternately, the optimistic approach basically waits for a certain amount of time (which can be dependent upon many different factors including system loading,

NOTES

frequency of access to database items, etc.) before actually performing the write-back operation. This potentially eliminates both the problems that the pessimistic approach suffers. However, it is not without problems of its own. For example, if too much time is allowed to lapse before the write-back operation is performed, it is very likely that transactions waiting to access the item will be unnecessarily blocked if the writing transaction never uses the same item again. The current trend in DBMS favours the optimistic approach coupled with interaction with the OS in order to optimize the time delay before write-back occurs. This approach tends to delay waiting transactions for a minimum amount of time and at the same time eliminates the problems of the pessimistic approach.

A transaction can have only one of two outcomes. It can either successfully complete execution and be committed enabling the database to assume a new consistent state, or it can fail to successfully complete execution (for any number of reasons) and get aborted. If the transaction is aborted, the database must be returned to the last consistent state which existed before the aborted transaction began execution. The aborted transaction is said to be rolled-back or undone.

A transaction which has been committed cannot be aborted. If a committed transaction is subsequently found to be a mistake, then another transaction called a compensating transaction must be executed to reverse the effects of the transaction that was committed in error.

An aborted transaction can be restarted later, and depending upon the cause of the failure, may successfully execute to completion and commit at that time.

10.2.5 Transaction States and Additional Operations

If a transaction does not fail, it must complete successfully. If a transaction fails to complete successfully it is said to be aborted. Changes or modifications performed by an aborted transaction must be cancelled. Changes made by the successful transactions must be made permanent.

In a simple transaction model, the transaction takes place in any one of the following ways:

- (i) **Active:** This is the initial state of the transaction where it stays during execution.
- (ii) **Partially Committed:** A transaction stays in this state after it has executed the final statement.
- (iii) **Failed:** This is a state reached by the transaction when it cannot continue normal execution anymore.
- (iv) **Terminated:** The terminal state is reached once the transaction has been aborted or rolled back and the database has been restored to the state it was in before the beginning of the transaction.
- (v) **Committed:** A transaction reaches this state after the completion of successful execution.

The most fundamental of all database operations are read and write operations. But these two operations are not sufficient. A system needs some additional

NOTES

operations too. Suppose, for instance, there is a system crash. To recover from that situation, the system needs to maintain a full record about the beginning, termination and abortion of the transactions. For example:

- **BEGIN TRANSACTION:** This denotes the beginning of an execution process.
- **END TRANSACTION:** This denotes completion of execution of a transaction.
- **COMMIT:** This indicates that the modifications or changes made by the successful transactions are to be incorporated permanently onto the database and it will never be undone in future.
- **ROLLBACK:** This indicates that the modifications (if any) made by any unsuccessful transaction needs to be undone.

In a real multiprogramming environment, some systems even keeps track of the present status of transactions running in the system. Sometimes, the system even tries to predict the next possibilities for the transaction to proceed and also says how to roll it back in case of failure. This is diagrammatically represented by a state transition diagram which can appear as follows:

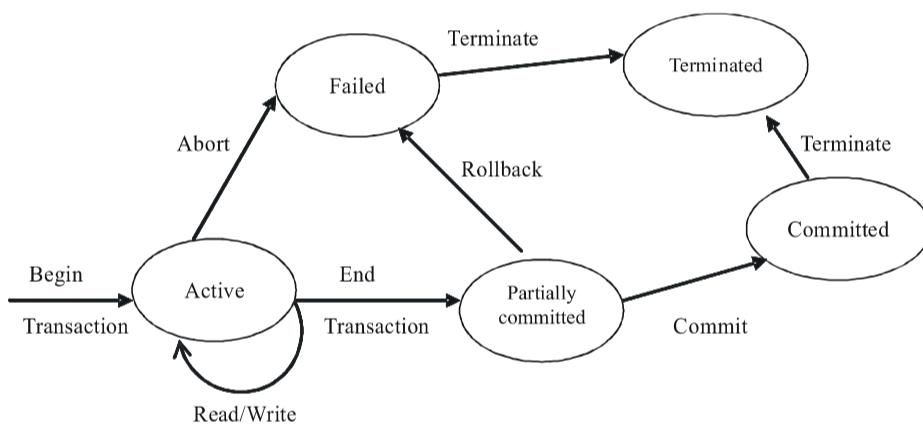


Fig. 10.1 State Transition Diagram of a Transaction

The arrows indicate the change of transaction from one state to another. Immediately following the start of execution, a transaction is said to be in an active state. In this state, the transaction performs read and write operations. The protocols in this state also ensure that no inconsistent entries into the database occur as a result of system failure. After this, the system enters the COMMITTED state wherein a transaction automatically moves on to the TERMINATED state. However, a transaction may fail due to many reasons before entering the COMMITTED state.

Once a transaction fails, it tries to undo the modification performed by the write operations. After this is done, the transaction will enter the terminated state to move out of the system. It is possible to restart a failed transaction later.

Figure 10.1 represents a state transition diagram for a transaction. In addition to the obvious states—active, committed and failed—there are two additional states:

NOTES

- (i) Partially Committed: This state occurs after the final statement has been executed. At this point, it may be found that the transaction has violated serializability (discussed later in this unit) or has violated an integrity constraint and the transaction needs to be aborted. Alternatively, the system may fail and any data updated by the transaction may not have been safely written back to the disk at this point in time. In such cases, the transaction would go into the Failed state and would have to be aborted. If the transaction has been successful, any updates can be safely written-back and the transaction can go to the Committed state.
- (ii) Failed: This state occurs when the transaction cannot be committed or the transaction is aborted while in the active state. This could be because the user aborted the transaction or possibly as a result of a concurrency control protocol which selects the transaction to be aborted to ensure serializability.

10.2.6 Concept of System Log

This system is capable of recording all the operations that take place and that are likely to impact the database status so that it can recover from any catastrophe. This information is known as the system log and is of great use when the system tries to recover from failures. Log information is stored permanently so that it does not get affected by normal system crashes or power failures. We will now discuss the kind of data that needs to be logged into the system. Suppose we denote a transaction by T which we call the transaction-id. This is generated whenever a new transaction is encountered. The transaction-id is unique for each transaction. The following information is stored in the log:

- (i) [Start, T]: Denotes the start of a transaction
- (ii) [Write, T, X, old, new]: Denotes that a transaction changes the value of a data item X from its old value to new value.
- (iii) [Read, T, X]: Denotes that the transaction has read the value of a data item X from the database.
- (iv) [Commit, T]: Denotes that T has completed all operations successfully and confirms that the changes should be made permanent to the database.
- (v) [Abort, T]: Denotes that transaction T has aborted due to some reason.

These entries are incomplete. In order to fulfil special requirements, certain modifications are made to the format and purpose, in some cases. It is to be noted that apart from being used for recovering, logs are also made use of in activities such as auditing and reporting.

‘Redo’ and ‘undo’ are the most commonly used operations related to logs. An undo operation sequentially traces the log details back and undoes the updates performed by the transaction in case of failures. Undo is performed if the transaction fails before system is asked to commit. But if the transaction fails between the system is asked to commit and the actual commit operation then redo operation is performed and the data items are assigned new values according to the log record.

10.2.7 Commit Point of a Transaction

Basic Concept of Transaction

The operations performed by a transaction on data items do not get reflected unless and until the transaction is committed. A transaction is said to have reached the ‘commit point’ once the operations accessing the database are carried out with success and the alterations/modifications made are recorded in the log. A transaction reaches ‘commit point’ when all operations accessing the database are successfully carried out and the modifications are recorded in the log. Once the commit point is reached the transaction is considered to be committed. This state is indicated by writing [commit, T] onto the log. Now the log contains the entire sequence of operations. A system failure at this point asks the system to perform redo operation. Using log for recovery purposes also have some serious disadvantages. Writing log records to secondary storage slows down the system operations. To solve this problem the most recent copy of log is maintained there in main memory. The records are written back to the disk at regular intervals. Once the transaction reaches the commit state, forceful writing of all records to the disk takes place and commit is performed. This is referred to as forceful writing of log file.

10.2.8 ACID Properties of a Transaction

Every transaction possesses four fundamental properties abbreviated as ACID properties.

- **Atomicity:** Being an atomic unit of work, a transaction can be either completed successfully or rolled back as a whole.
- **Consistency:** The transformation of a database from one consistent state to another by the transaction is guaranteed.
- **Isolation:** The transaction performs as though it is the only transaction in operation.
- **Durability:** After the successful completion of a transaction, it is guaranteed that the transaction will leave permanent effects even if there is a system failure.

Atomicity implies that each transaction is considered as an indivisible logical unit of work. An ideal example of that is a bank transaction. Here, the account of the debtor is debited and the account of the creditor is credited. Both these updates should take place to complete the whole transaction properly. The two operations are wrapped up into one single unit of work. Hence, either both credit and debit occur or none of them occur at all.

The isolation property can be ensured by guaranteeing that although many transactions take place in an interleaved manner, the overall effect is the same as execution of all transactions one after the other in a sequence. If two transactions, for instance, T1 and T2 are simultaneously executed, the net effect will be equivalent to the execution of T1 followed by T2 or execution of T2 followed by T1.

NOTES

NOTES

The DBMS provides no guarantees about which of these orders is effectively chosen. If each transaction transforms the database from one consistent state to another consistent state, executing several transaction one after the other (on a consistent initial database instance) will also cause a consistent final database instance.

Transactions can be incomplete due to three reasons:

- (i) A transaction can be aborted or terminated unsuccessfully, by the DBMS because of some anomaly arising during execution. If a transaction is aborted by the DBMS for some internal reason, it is automatically restarted and executed anew.
- (ii) The system may crash (for example, due to interrupted power supply) while one or more transactions are in progress.
- (iii) A transaction may encounter an unexpected situation (for example, read an unexpected data value or be unable to access some disk) and decide to abort (i.e., terminate itself).

Of course, since users think of transactions as being atomic, a transaction that is interrupted in the middle may leave the database in an inconsistent state. Thus, a DBMS must find a way to remove the effects of partial transactions from the database that is, it must ensure transaction atomicity. Either all the actions of a transaction are carried out, or none of them are carried out. A DBMS ensures transaction atomicity by undoing the actions of incomplete transactions. This means that users can ignore incomplete transactions in thinking about how the database is modified by transactions over time. To be able to do this, the DBMS maintains a record, called the log, of all writes to the database. The log is also used to ensure durability. In case a system happens to crash before the modifications made by a completed transaction are written to the disk, the log is employed, to not just remember but also restore the changes, on restarting.

Durability guarantees that once a transaction is successfully committed, the modifications made by the transaction are made permanent. Even in case of system or hardware failure, the system ensures that the data in the database is correct and the changes made by a committed transaction are still available after recovery from failure. Consistency and isolation are guaranteed by concurrency control whereas atomicity and durability are guaranteed by recovery.

Now, how can these desirable properties of transactions be enforced? During the implementation and designing of the transaction, the concept of atomicity is taken care of. On the other hand, if a transaction fails before completion of the task assigned, the recovery software is able to undo the partial effects of the transactions onto the database.

It is the database programmer's duty to preserve the consistency of the database before and after a transaction. 'Consistent state' of a database implies a state that satisfies the restrictions/limitations that the scheme specifies. It is the responsibility of the database programmer to write the database programs in such

a way that a transaction is only allowed to enter a database when it is in a consistent state and to leave only when it has reached another consistent state. This, of course means that there is no interference from any other transaction in the action of the concerned transaction.

Basic Concept of Transaction

Now come to the next concept, i.e., isolation. Isolation means that every transaction should carry on its job without being dominated by other transactions that may be operating on the same database. To implement isolation we should take care of the fact that no transaction makes its partial updates available to other transactions. Though this eliminates the temporary update problem it is not sufficient to solve several other problems.

10.2.9 Concurrent Execution of Transactions

Transaction-processing systems usually let many transactions run simultaneously. This can be achieved only through extra work. We have a convenient way to describe interleaved executions of transactions. The DBMS interleaves the actions of different transactions to improve performance.

10.2.10 Motivation for Concurrent Execution of Transactions

Ensuring transaction isolation, while permitting such concurrent execution, is difficult, but is essential for reasons of performance.

- (i) It is possible for the CPU to process another transaction while one particular transaction waits for a page to be read from the disk. This is because I/O activity can be done in parallel with CPU activity in a computer. Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle, and increases system throughput (the average number of transactions completed in a given time).
- (ii) When a short transaction is executed with a long transaction in an interleaved manner, the short transactions are completed fast. In serial execution, a short transaction could get stuck behind a long transaction which results in unnecessary delays in response time or increase in the average time required to finish a transaction.

Check Your Progress

1. What is transaction?
2. List the ACID properties of a transaction.

10.3 CONCURRENCY CONTROL

Concurrency control is the process of managing simultaneous execution of transactions without letting them interfere with one another. Concurrency control operates under one main assumption as follows:

NOTES

Given a consistent state of the database as input, an individually correct transaction will produce a correct state of the database as output if the transaction is executed in isolation. The goal of concurrency control is to allow multiple transactions to be executed concurrently, that is simultaneously, within a certain time period with all the transactions producing a correct state of the database at the end of their concurrent execution.

Although two transactions may be individually correct and produce a correct database state when executed in isolation, their concurrent (interleaved) execution may result in an inconsistent state. Let us look at an example of a concurrent execution that would result in an incorrect state:

T ₁	T ₂
Read(X) X = X - 50	
	Read(X) temp = X * 0.2 X = X - temp Write (X) Read(Y)
Write(X) Read(Y) Y = Y + 50 Write(Y)	
	Y = Y + temp Write(Y)

The first transaction transfers Rs 50 from the account of person X to the account of person Y. The second one transfers twenty percent from the account of person X to the account of person Y. In this example, the ten percent transfer from X is lost and the deposit of Rs 50 to Y is lost. You can prove it to yourself. The sum of X + Y should be the same before and after both transactions. Concurrency control can regulate the interaction among concurrent transactions to keep them from disturbing the database consistency.

There are several classical examples of the problems concurrent execution can cause in the correctness of a database. Problems of concurrency control that a DBMS must handle will be discussed in this section.

10.3.1 The Lost Update Problem

When two transactions access the same database items and also perform their own operations in a way that makes the value of some database item wrong, a lost update problem takes place.

Suppose that we have two distinct transactions called T1 and T2 whose function is to sell concert tickets. Considering only the fundamental database operations that would be required by this type of transaction, namely their read and write operations, examine the following concurrent execution of T1 and T2.

(Assume that the number of tickets, i.e., seats available, is represented by the variable n .)

Basic Concept of Transaction

Time	Action
t_0	transaction T1 performs Read(n)
t_1	transaction T2 performs Read(n)
t_2	transaction T1 checks n , issues ticket, performs Write($n - 1$)
t_3	transaction T2 checks n , issues ticket, performs Write($n - 1$)

NOTES

The lost update problem is as follows: The update performed by T1 at time t_2 is ‘lost’ due to the update performed by T2 at time t_3 . The correct value of n in the database should be $n-2$, but it clearly is not, so the database is incorrect.

The lost update problem can be handled in several different ways as follows:

- (i) Prevent T2 from reading the value of n at time t_1 on the basis that T1 has already read the value of n and may therefore update this value. If this is the case, then T2 would be operating with an obsolete value of n and this will not occur if T2 is blocked.
- (ii) Prevent T1 from writing the value of $n-1$ at time t_2 on the basis that T2 has also read the value of n and would then be operating with an invalid value of n , since clearly T2 cannot be forced to re-read the value of n .
- (iii) Prevent T2 from writing the value of $n-1$ at time t_3 on the basis that T1 has already updated the value of n and since T1 preceded T2 in terms of execution start time (thus T1 is the ‘older’ transaction) and thus T2’s write is based upon an obsolete value of n .

The first two solutions to the lost update problem can be implemented using locking protocols, while the third solution can be implemented using a time-stamping protocol.

10.3.2 Uncommitted Dependency—The Dirty Read Problem

A dirty read problem occurs when a transaction updates an item in the database and then there is a failure in the transaction for some reason. As the transaction fails, the updated item in the database is accessed by another transaction before it can go back to the original value.

In other words, a transaction T1 updates a record which is read by T2. Then T1 aborts so that T2 then has values which do not remain after T1 has rolled back. Consider the following concurrent schedule of transactions T1 and T2.

Time	Action
t_0	transaction T1 performs Read(n)
t_1	transaction T1 performs Write(n)
t_2	transaction T2 performs Read(n)
t_3	transaction T1 aborts

Self-Instructional Material

NOTES

The dirty read problem is as follows:

When T1 aborts, it must undo any changes it has made to the database. However, before T1 aborted, T2 read a value (at time t_2) that T1 wrote (at time t_1). Thus, T2 has read a ‘dirty’ value – one that will not remain in the database after T1 is rolled-back, thus T2 must undo its read, i.e., T2 must abort.

Typically, time-stamping protocols are employed to implement solutions to the dirty read problem as most locking protocols to eliminate this problem are too restrictive in the level of concurrency which they allow.

10.3.3 Unrepeatable Read or Inconsistent Retrievals Problem

Suppose two users X and Y access a department’s database concurrently. User X is updates the database to provide all employees with a salary raise of 5%. At the same time, user Y wants to know the total salary that has been paid to the department.

As the two transactions are being executed simultaneously on the same database, there is always a high probability to interfere with each other. As a result, the sum includes certain salaries before the raise and certain salaries subsequent to the raise. Such a sum cannot be considered as an acceptable value of the total salary (the value before the raise or after the raise would be different).

Table 10.1 An Example of Inconsistent Retrieval

X	Time	Y
Read(salary) of Employee 100	1	-
-	2	Sum = 0.0
Update Salary	3	-
-	4	Read(salary) of Employee 100
Write(salary)	5	-
-	6	Sum = Sum + Salary of Employee 100
Read(salary) of Employee 101	7	-
-	8	Read(salary) of Employee 101
Update Salary	9	-
-	10	Sum = Sum + Salary of Employee 101
Write(salary) of Employee 101	11	-
-	12	-
-	13	-
etc	-	etc

The problem illustrated in the last example 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.

10.3.4 Phantom Reads

Phantom reads take place when insertion or deletion is performed against a row belonging to a range of rows read by a transaction. The first read of the range of rows reflects a row that does not exist in the second or successive read because of a deletion by another transaction. In the same way, due to an insertion by another transaction, the second or successive read of the transaction shows a row that was non-existent in the original read.

An editor, for instance, modifies a document submitted by an author. However, when the changes are incorporated into the main/master copy of the document by the production department, it is discovered that new content which is unedited has been added by the author. Such a problem can be kept at bay if there is a provision to check anybody from adding new material. In other words, it would help to have a system wherein no one can add/modify a document till the editors and the production department have completed their work on the original document.

10.3.5 Schedule

One obvious way to prevent transactions from interfering with one another is to execute them serially. That is, a given transaction must be committed before the next transaction can begin execution. While this will certainly guarantee a consistent and correct database, it goes against one of the primary goals of a DBMS. In other words, multi-user access to the database should be maximized. In order to accomplish this goal, concurrent access to the database is clearly required. However, many of the concurrent operations on a database would be accessing different items in the database and would clearly not cause any conflicts and could thus be scheduled concurrently. Some of these operations however, will potentially conflict and cause consistency problems if left unresolved. It is the responsibility of the scheduler, in conjunction with the recovery subsystem, to determine if a concurrent schedule of transactions will leave the database in a correct state or if some intervention is required to ensure that it does. In order to do this, the scheduler needs to be able to determine if the concurrent schedule of transactions is equivalent to some serial ordering of the same transactions.

A schedule is a sequence of operations performed by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions. As before, we are only interested in the set of operations performed by the transactions which affect the database, so the schedule consists only of a sequence of read and/or write operations followed by either a commit or abort operation.

Serial Schedule

In a serial schedule, the transactions are performed in serial order. For example, two transactions T1 and T2 could be performed in serial order in two ways: T1

Basic Concept of Transaction

NOTES

NOTES

followed by T2 or T2 followed by T1. Similarly, for a set of three transactions T1, T2, and T3, the following are the possible ways in which transactions can be executed in serial order:

T1 followed by T2 followed by T3	T1 followed by T3 followed by T2
T2 followed by T1 followed by T3	T2 followed by T3 followed by T1
T3 followed by T1 followed by T2	T3 followed by T2 followed by T1

In a serial schedule, there is no possibility of interference between transactions since they are executed in isolation. However, there is no guarantee that the results of all serial executions of a given set of transactions will be identical. For example, in banking, whether interest is calculated on an account before a large deposit is made or after it is made is of great significance.

10.3.6 Serializability

Suppose two transactions, T_1 and T_2 are to be scheduled. They can be scheduled in a number of ways. To schedule them serially without being bothered about interleaving would be the most common way. In a serial schedule transaction, all operations of transaction T_2 should follow all the operations of T_1 or vice versa.

T_1	T_2
Read(X)	
$X=X+N$	
Write(X)	
Read(Y)	
$Y=Y+N$	
Write(Y)	
Time	Read(X)
	$X=X+P$
	Write(X)

Non-interleaved Serial Schedule A

T_1	T_2	T_1	T_2
Read(X)	read(X)		Read(X)
		→	
$X=X+N$	$X=X+P$		$X=X+P$
Write(X)	Write(X)		Write(X)
Read(Y)		Read(X)	
$Y=Y+N$			
Write(Y)			

Non-interleaved Serial Schedule B

Now, these can be termed as serial schedules, since all the operations of one transaction are being followed by the entire sequence of operations of the other transaction.

In the interleaved mode, the operations of T_1 are mixed with the operations of T_2 . This can be done in a number of ways. Two such sequences are given below:

T_1	T_2
Read(X)	
$X=X+N$	
	Read(X)
	$X=X+P$
Write(X)	
Read(Y)	
	Write(X)
	$Y=Y+N$
	Write(Y)

Interleaved non-serial schedule C

T_1	T_2
Read(X)	
$X=X+N$	
Write(X)	
	Read(X)
	$X=X+P$
	Write(X)
Read(Y)	
	$Y=Y+N$
	Write(Y)

Interleaved (Nonserial) Schedule D

If for every transaction T in a schedule S all the operations are executed consecutively, we call it a serial schedule; otherwise we call it a non-serial schedule. A non-interleaved schedule of independent transactions always results in a consistent database state as the transactions commit or abort before the beginning of the next transaction. A non-interleaved schedule is guaranteed to produce a correct result as long as the individual transactions are free of errors. However, non-interleaved schedules suffer from low utilization and wastage of resources. Often, a transaction waiting for an I/O makes the subsequent transactions wait causing wastage of resources or reduction in resource utilization. If the former transaction takes too long to execute, the latter one keeps waiting till its completion.

The problem with serial schedules is resource wastage. In case of a serial schedule, if a transaction is waiting for an I/O, the subsequent transactions will also wait causing wastage of resources. If a transaction T is extremely long, the other transactions will have to keep waiting till T is completed. In such a schedule, no concurrency is supported. Therefore, in general, the serial scheduling concept is unacceptable in practice.

NOTES

NOTES

Serial schedules are unacceptable in practice. To solve this problem, the operations need to be interleaved. But the interleaving sequence should be well planned. A wrong interleaving sequence may leave the schedule incorrect and the database inconsistent. Therefore, a methodology should be to determine which schedules produce correct results and which ones do not.

A schedule S of N transactions is said to be a serializable schedule if it is equivalent to a serial schedule which comprises the same N transactions. N transactions can produce $N!$ serial schedules. If you carry on interleaving them, the number of possible combinations becomes very large. To solve this problem, all the non-serial schedules are divided into two disjoint groups. One comprises those schedules that are equivalent to one or more serial schedules and the other consists of those schedules that are not. The first category of schedules is called ‘serializable schedules’ and the latter is called ‘non-serializable schedules’.

Conflicting Actions

Conflict between two or more actions takes place if:

1. The read or write action is performed on the same data object.
2. The actions are performed by different transactions.
3. At least one of the actions performed is a write operation.

The following set of actions is conflicting:

- T1: Read(X), T2:Write(X), T3:Write(X)

While the following sets of actions are not:

- T1: Read(X), T2:Read(X), T3:Read(X)
- T1: Read(X), T2:Write(Y), T3:Read(X)

Conflict Equivalence

Two schedules can be said to be conflict equivalent, if any two conflicting operations in both the schedules are executed in the same order. If somehow the order of conflicting operations in both the schedules is not the same then the schedules produces different database states at the end and hence they cannot be equivalent to each other.

Conflict-Serializable Schedule

A conflict-serializable schedule is one which is conflict-equivalent to other serial schedule(s).

You can say that a schedule is conflict-serializable only if an acyclic precedence graph or a serializability graph exists for the schedule.

NOTES

T1	T2
Read(A)	
	Read(A)
Write(B)	
COMMIT	
	Write(A)
	COMMIT

Which is conflict-equivalent to the serial schedule <T1, T2>

Testing for Conflict Serializability of a Schedule

To test a schedule for conflict serializability, an algorithm may be suggested:

1. For each transaction T_i , participating in the schedule S, create a node labelled T_i in the precedence graph.
2. For each case where T_j executes a Read(X) after T_i executes write(X), create an edge from T_i to T_j in the precedence graph.
3. For each case where T_j executes Write(X) after T_i executes a Read(X), create an edge from T_i to T_j in the graph.
4. For each case where T_j executes a Write(X) after T_i executes a Write(X), create an edge from T_i to T_j in the graph.
5. The schedule S is serializable if and only if there are no cycles in the graph.

If we apply these methods to write the precedence graphs for the above four cases, we get the following precedence graphs.

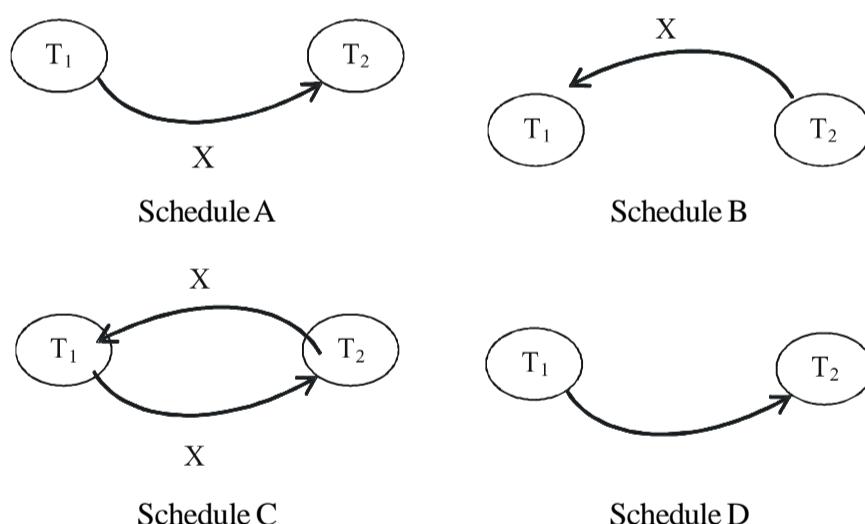


Fig. 10.2 Testing Serializability of a Schedule

We may conclude that schedule D is equivalent to schedule A.

NOTES

View Equivalence

Two schedules S1 and S2 are said to be view-equivalent when the following conditions are satisfied:

1. For every data item A if transaction T_i reads the initial value of A in S1, then in S2 also transaction T_i must read the initial value of A.
2. For each data item A if transaction T_i executes a Read(A) on data item A and if the value was produced by a Write(A) operation performed by another transaction T_j in S1, then in S2 also the same transaction T_i should perform Read(A) on the value produced by Write(A) operation by T_j .
3. For each data item A in S if the transaction T_i performs the final Write (A) operation, then in S2 also the final Write (A) operation must be performed by T_i .

View-Serializable Schedule

A view-serializable schedule is view-equivalent to some serial schedule. All conflict-serializable schedules are view-serializable.

G =	T1	T2
	Read(A)	
		Read(A)
	Write(B)	
	COMMIT	
		Write(A)
		COMMIT

Above is an example of a schedule which is both view-serializable as well as conflict-serializable. There are view serializable schedules which are not conflict serializable. Such a schedule contains '**blind writes**'.

H =	T1	T2	T3
	Read(A)		
		Write(A)	
		COMMIT	
	Write(A)		
	COMMIT		
			Write(A)
			COMMIT

The above example is not conflict-serializable, but it is view-serializable since it has a view-equivalent serial schedule $\langle T1, T2, T3 \rangle$.

Another Example:

$$S1 = \{[T3:\text{read}(A)], [T4:\text{write}(A)], [T3:\text{write}(A)], [T6:\text{write}(A)]\}$$

$$S2 = \{[T3:\text{read}(A)], [T3:\text{write}(A)], [T4:\text{write}(A)], [T6:\text{write}(A)]\}$$

Note that S1 is a concurrent schedule and S2 is a serial schedule. S1 is view equivalent to S2 since the one read(A) instruction reads the initial value of A in both schedules and T6 performs the last write(A) operation in both schedules. Therefore, S1 is a view serializable schedule.

Basic Concept of Transaction

The only difference between conflict serializability and view serializability is that the former satisfies ‘constrained write’ while the latter does not. This condition states that any write operation $W_i(A)$ in T_i is preceded by an $R_j(A)$ in T_j and that the value written by $W_i(A)$ in T_j depends only on the value of A read by $R_j(A)$. The assumption behind the concept of constrained write is that the new value of a data item A is a function based on the old value of the same data item in the database. So view serializability is less stringent than conflict serializability. View serializability applies the concept of ‘unconstrained write assumption’ where the value of a data item produced by a write operation performed by any transaction does not depend upon the old value of the data item.

The only problem with view serializability is its complexity with regard to computation. No efficient algorithm exists to do the same.

Uses of Serializability

If you say that S is correct, it implies or is equivalent to proving the serializability of schedule S.

Therefore, it is guaranteed that the schedule will provide correct results. However, being serial and being serializable are different things. A serial scheduling may be inefficient due to reasons explained earlier, which causes underutilization of the CPU, I/O devices and in some cases such as mass reservation system, becomes untenable. A serializable schedule, on the other hand, not only has the advantages of concurrent execution (ability to cater to numerous simultaneous users, efficient system utilization) but also guarantees correctness.

The scheduling process is done by the operating system routines after considering various factors such as the priority of the process in comparison to other processes, time of submission of transaction, system load, and many other factors. Also, since it is possible to have numerous interleaving combinations, it is very difficult to determine, in advance, the way in which the transactions are interleaved. That is, getting the various schedules itself is tough, leave alone testing them for serializability.

Therefore, most DBMS protocols employ a more practical technique. Instead of generating the schedules and checking for serializability before using them, they apply restrictions or controls on the transactions themselves. These restrictions are followed by each participating transaction, automatically ensuring serializability in all the schedules that the participating schedules create.

In addition, it is not easy to determine the start of a schedule and its finish, since transactions get submitted at different times.

NOTES

NOTES

Therefore, the theory of serializability can be used to tackle this problem by considering only the committed projection C(CS) of the schedule. Therefore, as an approximation, a schedule S can be defined as serializable if its committed C(CS) is equivalent to some serial schedule.

10.3.7 Recoverability

If a transaction T_i fails to commit, its effects should be undone to ensure the atomicity property of transaction. In an environment which supports concurrency, it is essential to ensure that any transaction T_j dependent on T_i (that is, T_j has read data written by T_i) is also aborted. This phenomenon is referred to as cascaded rollback. Cascading rollback is undesirable as it results in cancellation of a significant amount of work.

Transactions commit only after all transactions whose changes they read commit.

	T1	T2
F=	Read(A)	
	Write(A)	
		Read(A)
		Write(A)
	COMMIT	
		COMMIT

	T1	T2
F2=	Read(A)	
	Write(A)	
		Read(A)
		Write(A)
		Abort
		Abort

These schedules are recoverable. F is recoverable because T1 commits before T2, that makes the value read by T2 correct. Then T2 can commit itself. In F2, if T1 aborted, T2 has to abort because the value of A read by it is incorrect. In both cases, the database is left in a consistent state.

Unrecoverable

If a transaction T1 aborts, and a transaction T2 commits, but T2 relied on T1, we have an unrecoverable schedule.

G=	T1	T2
	Read(A)	
	Write(A)	
		Read(A)
		Write(A)
		COMMIT
	Abort	

In this example, G is unrecoverable, because T2 read the value of A written by T1, and committed. T1 later aborted, therefore the value read by T2 is wrong, but since T2 committed, this schedule is unrecoverable.

Avoiding Cascading Aborts (Rollbacks)

A series of transaction rollbacks can result from one transaction abort. Disallowing a transaction from reading uncommitted changes from a different transaction in the same schedule is a strategy that can be adopted to avoid cascading aborts. This is also referred to as cascadeless.

The following examples are the same as the one from the discussion on recoverable:

	T1	T2
F=	Read(A)	
	Write(A)	
		Read(A)
		Write(A)
	COMMIT	
		COMMIT

	T1	T2
F2=	Read(A)	
	Write(A)	
		Read(A)
		Write(A)
		Abort
		Abort

In this example, although F2 can be recovered, it does not avoid cascading aborts. It is clear that if T1 aborts, T2 will also have to be aborted to maintain the correctness of the schedule as T2 has already read the uncommitted value written by T1.

The following is a recoverable schedule, which avoids cascading abort. Note, however, that the update of A by T1 is always lost.

	T1	T2
F=		Read(A)
	Read(A)	
	Write(A)	
		Write(A)
	Abort	
		COMMIT

Cascading aborts avoidance is sufficient but not necessary for a schedule to be recoverable.

Strict

A schedule is strict if for any two transactions T1, T2, if a write operation of T1 precedes a *conflicting* operation of T2 (either read or write), then the commit event of T1 also precedes that conflicting operation of T2. Any strict schedule is cascadeless, but not the converse.

Hierarchical Relationship between Serializability Classes

The following subclass clauses illustrate the hierarchical relationships between serializability classes:

- Serial ⊂ commitment-ordered ⊂ conflict-serializable ⊂ view-serializable ⊂ all schedules
- Serial ⊂ strict ⊂ avoids cascading aborts ⊂ recoverable ⊂ all schedules

Basic Concept of Transaction

NOTES

NOTES

The following Venn diagram illustrates the above clauses graphically.

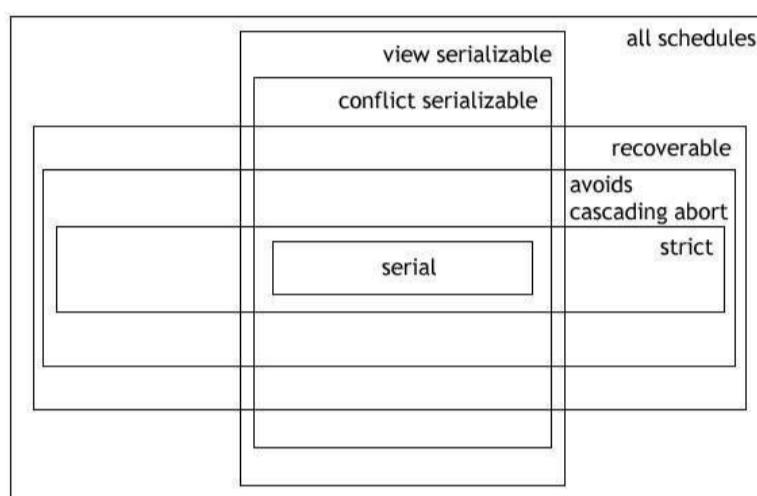


Fig. 10.3 Hierarchical Relationship between Serializability Classes

Check Your Progress

3. What do you understand by concurrency control?
4. What is conflict-serializable schedule?

10.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. A transaction is a logical unit of work in a DBMS that includes one or more database access operations.
2. ACID properties of transactions are:
 - i. Atomicity
 - ii. Consistency
 - iii. Isolation
 - iv. Durability
3. Concurrency control is the process of managing simultaneous execution of transactions without letting them interfere with one another.
4. A conflict-serializable schedule is one which is conflict-equivalent to other serial schedule(s).

10.5 SUMMARY

- A transaction is a logical unit of work in a DBMS that includes one or more database access operations. A transaction may involve an arbitrary number of operations on the database which brings about changes in the state of the database while preserving system consistency.
- Interleaved transactions take place when two or more transactions are processed concurrently with only one transaction at a time making progress toward completion.
- Simultaneous transactions take place when two or more transactions are parallelly processed. The number of CPUs online determines the number of transactions progressing.
- Serial transactions may occur in either a single CPU or multiple CPU environments.
- The system is capable of recording all the operations that take place and that are likely to impact the database status so that it can recover from any catastrophe. This information is known as the system log and is of great use when the system tries to recover from failures.
- The operations performed by a transaction on data items do not get reflected unless and until the transaction is committed. A transaction is said to have reached the ‘commit point’ once the operations accessing the database are carried out with success and the alterations/modifications made are recorded in the log.
- Every transaction possesses four fundamental properties abbreviated as ACID properties.
- Concurrency control is the process of managing simultaneous execution of transactions without letting them interfere with one another.
- Phantom reads take place when insertion or deletion is performed against a row belonging to a range of rows read by a transaction.
- One obvious way to prevent transactions from interfering with one another is to execute them serially. That is, a given transaction must be committed before the next transaction can begin execution.
- Two schedules can be said to be conflict equivalent, if any two conflicting operations in both the schedules are executed in the same order.
- A conflict-serializable schedule is one which is conflict-equivalent to other serial schedule(s).

NOTES

NOTES

10.6 KEY WORDS

- **Transaction:** It is a logical unit of work in a DBMS that includes one or more database access operations.
- **Atomicity:** It implies that each transaction is considered as an indivisible logical unit of work which can be either completed successfully or rolled back as a whole.
- **Durability:** It guarantees that once a transaction is successfully committed, the modifications made by the transaction are made permanent.

10.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What are the different transaction processing steps?
2. Discuss the concept of system log.
3. What is commit point of a transaction?
4. What do you understand by serial schedule?
5. Write the steps for testing conflict serializability of a schedule.

Long Answer Questions

1. Explain the transaction states with the help of state transition diagram.
2. What are the ACID properties of a transaction? Explain.
3. Explain the concept of concurrency control with the help of suitable example.
4. What is serializability? How will you test it?
5. How will you recover a transaction if it fails to commit?

10.8 FURTHER READINGS

Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.

Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.

Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.

Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.

NOTES

- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

UNIT 11 PROTOCOLS

NOTES

Structure

- 11.0 Introduction
- 11.1 Objectives
- 11.2 Concurrency Control Mechanism and Locking Protocols
 - 11.2.1 Locking Protocol for Concurrency
 - 11.2.2 Other Concurrency Control Protocols
 - 11.2.3 Two-Phase Locking Protocols
 - 11.2.4 Concurrency Control Protocols based on Timestamp
 - 11.2.5 Concurrency Control using Multi-Versioning
 - 11.2.6 Validation (Optimistic) Concurrency Control Techniques
- 11.3 Multiple Granularity Locking Protocol
- 11.4 Answers to Check Your Progress Questions
- 11.5 Summary
- 11.6 Key Words
- 11.7 Self Assessment Questions and Exercises
- 11.8 Further Readings

11.0 INTRODUCTION

The DBMS houses information that can communicate with each other and can be manipulated at any given moment. There is a probability of more than one user attempting to access a certain data item at the same time to create a concurrency in a few situations. Therefore, a need arises to deal with the said competition to handle the simultaneous execution of tractions between the various databases in the picture.

By providing the necessary isolation to the tractions involved within the DBMS, Lock Based Protocols remove the deficiencies caused by the concurrency access in DBMS.

Both concurrency control and recovery are required to protect the database from data inconsistencies and loss of data. Most DBMS allow concurrent access to the database. If these concurrent operations are not controlled, the various accesses may interfere with one another and the database could arrive at an inconsistent state. To prevent this from occurring, the DBMS implements a concurrency control protocol that prevents concurrent accesses from interfering with one another.

In this unit, you will study about the protocols, lock based protocols, timestamp based protocols, validation based protocols, and multiple granularity.

11.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the concept of locking
- Discuss the types of locks
- Explain the types of locking algorithms
- Understand the concurrency control protocols
- Understand how multiple granularity protocol works

NOTES

11.2 CONCURRENCY CONTROL MECHANISM AND LOCKING PROTOCOLS

As we have seen from our discussions regarding serializable schedules, concurrent access to the database will guarantee a consistent and correct database only if the concurrent schedule is equivalent to some serial schedule. What we now need is a technique that will ensure that the concurrent schedule is serializable. In other words, as transactions execute concurrently on the database, we will allow them to do so only with respect to some protocol that will guarantee that their schedule is serializable.

Serializability can be achieved by several different means. *Locking* and *time-stamping* protocols are two very common techniques. Locking is a more pessimistic approach to concurrency control while time-stamping is a more optimistic approach. *Versioning* is a more recent optimistic approach to concurrency control which is rapidly gaining popularity amongst DBMS developers. Locking protocols can be quite restrictive and may lead to unnecessary blocking, live lock or starvation, and in extreme situations deadlock. Time-stamping protocols, while not suffering from these problems, are more complicated to implement and require a higher-level of sophistication from the DBMS. We'll focus primarily on locking protocols which are the most common type of concurrency control mechanisms in standard DBMS.

11.2.1 Locking Protocol for Concurrency

Locking is the most popular concurrency control mechanism implemented by many important techniques. Lock is a variable associated with every data item to indicate whether the data item is available for an operation to be performed on it. Lock variable synchronizes the operations performed on these data items by the concurrent transactions. A planned and proper implementation of lock solves many problems due to concurrency listed earlier. However, there are some problems created by lock itself. We shall be discussing these in the subsequent sections.

NOTES**Lock Granularity**

Locking ensures conflict prevention by holding a lock on various parts of a database. Sometimes, these locks are held by the DBMS; sometimes the DBA or a user himself holds the lock explicitly. Applying the highest level of concurrency in databases with a high level of concurrency degrades the performance. Typical levels of locking are given as follows:

- (i) **Database locking:** The entire database is locked in this method during updating by any user. This is the easiest method to be implemented but not a very accepted one. This method applied only when the DBA is performing some maintenance task on the database like restoring, backing up etc.
- (ii) **Table locking:** Some operating systems consider tables to be equivalent to files. If it is so, then those systems can apply file locking systems to lock the tables of the database. This method is efficient but can lead to congestion in busy databases.
- (iii) **Page locking:** It is better to lock a separate page instead of locking the entire page. In fact, a page is a disk concept. A file stored on the disk is divided and written into physically separate sections called sectors. So, it is efficient to lock access to a disk page or sector permitting the users to access the rest of the table.
- (iv) **Row locking:** This locks a database row denying any request of modification by any transaction on the database. This technique involves high implementation overhead.
- (v) **Column locking:** Column locking is supported by very few databases. This method is generally considered unacceptable because of the high overhead involved in it.

Types of locks

- **Binary locks:** In this method, the lock variable can have only two states. This method is too simple, but too stringent. This is not a very good method to be accepted.
- **Shared/exclusive locks:** This method is more practical and accepted in general lock-based systems. This scheme locks the data items in two modes. Shared or read mode, and exclusive or write mode.
- **Certify lock:** This is used to improve the performance of locking protocols.

(i) Binary Locks

As we said previously, a binary locking scheme has two states. The states are designated by '0' and '1'. These two digits are used to indicate whether a data item is locked or unlocked. For example if '1' is presumed to indicate that a lock is held then '0' indicates that the lock is open. This means that if the value of the

lock associated with the data item X is ‘1’ then no operation can be performed on the data item. It indicates that the item is locked.

Protocols

The concept is that a data item cannot be modified by transactions if it is locked. If a transaction is modifying the value of a data item it cannot be accessed until the modification is completed. So, if a transaction wants to modify a data item it requests a lock on it. Now, the lock request is granted only if the data item is not already locked by some other transaction. Otherwise, it has to wait for the lock. After the lock is acquired, the transaction performs the modification and finally releases the lock.

NOTES

The two operations lockitem(X) and unlockitem(X) are required to be performed to implement this method. Lockitem(X) locks a data item and prevents it from being accessed by other transactions and unlockitem(X) releases the lock on a data item and makes it available to the other transactions. Therefore, from the point of view of the lock variable we can describe the procedure as follows:

Any transaction wanting to hold a lock on a data item first checks the value of the lock variable associated with it. If it finds the lock status of the variable to be ‘1’, then it is already locked. The transaction then waits. Once the lock status of the data item becomes ‘0’ the waiting transaction can hold a lock on it to perform the modifications. It then issues a lockitem(X) command. Once it completes its operation with the data item, it issues unlockitem(X) and resets the value of lock variable so that some other transaction waiting for acquiring a lock can proceed.

Notice that the binary lock essentially produces a ‘mutually exclusive’ type of situation for the data item, so that only one transaction can access it. These operations can be easily written as an algorithm as follows:

The Locking Algorithm

```
Lockitem(X):
    Start: if Lock(X)=0 /* item is unlocked*/
           Lock(X)=1 /*lock it*/
    Else
        {
            wait(until Lock(X)=0) and the lock manager wakes up the
            transaction)
            go to start;
        }
```

The Unlocking Algorithm

```
Unlock item(X):
    Lock(X) ← 0;
```

NOTES

{ If any transactions are waiting,
Wakeup one of the waiting transactions }

The only restriction on the use of the binary locks is that they should be implemented as indivisible units (also called ‘critical sections’ in operating systems terminology). That means, no interleaving operations should be allowed, once a lock or unlock operation is started, until the operation is completed. Otherwise, if a transaction locks a unit and gets interleaved with many other transactions, the locked unit may become unavailable for long times to come with catastrophic results.

To make use of the binary lock schemes, every transaction should follow certain rules:

1. The lockitem(X) command is to be issued before issuing a read(X) or write(X).
2. The unlockitem(X) command is to be issued after completion of all read(X) and write(X) operations on X.
3. If a transaction holds a lock on a data item X, it should not issue another lockitem(X) command.
4. If a transaction is not currently holding a lock on a data item X, it should not issue an unlockitem(X) command.

No other transaction Tj would be permitted to operate on a data item X between a lockitem(X) and an unlockitem(X) issued by Ti. Therefore, between these intervals only Ti holds the value of the data item X. Thus, many of the above mentioned problems are solved by this scheme.

(ii) Shared/Exclusive Locks

A binary lock is easy to implement, it also looks satisfactory enough but suffers from serious difficulties. It strictly prohibits more than one transaction to access a data item simultaneously. This scheme does not even permit more than one transaction to perform a read operation on the same data item simultaneously. This is where the problem lies. While one transaction is performing a write operation, the other transactions should not be permitted to perform a write or read operation on the same data item. But where is the harm in allowing two read operations to take place simultaneously? In fact, allowing simultaneous read operations on the same data item would increase the system’s performance without causing any harm to the database.

This concept gave rise to the idea of shared/exclusive locks. The notion of exclusive lock is too rigid and has performance side effects. There is a need to make the locking mechanism less stringent by introducing shared or read locks.

There are two types of shared locks:

- Exclusive locks/write locks (X locks).
- Shared locks/read locks (S locks).

NOTES**(a) Write Lock**

A write lock set up on a data item, allows a transaction to read and/or modify its value, exclusively. No other transaction can read or write to that data item while the write lock is in effect.

If transaction A holds an X lock on record p, then transaction B requesting a lock on the same record will be denied.

(b) Read Lock

A read lock is set up by a transaction. It is non-exclusive, i.e., it can be shared among many readers, allowing parallel reads to happen. Nobody can change the data item while the read lock is on.

Figure 11.1 shows the **lock compatibility matrix**.

Current State of Lock of Data Items			
Requested Lock	Exclusive	Shared	Unlocked
Exclusive	N	N	Y
Shared	N	Y	Y
Unlock	Y	Y	-

Fig. 11.1 Lock Compatibility Matrix

Normally, locks are implicit. A FETCH request is an implicit request for a shared lock whereas an UPDATE request is an implicit request for an exclusive lock. Explicit lock requests need to be issued if a different kind of lock is required during an operation. For example, if an X lock is to be acquired before a FETCH it has to be explicitly requested for.

We need to think of three operations, a read lock, a write lock and unlock. The algorithms can be as follows:

Read Lock(X):

```

Start: If Lock (X) = ‘unlocked’
{
    Lock(X) ← ‘readlocked’;
    no_of_reads(X) ← 1;
}
else if Lock(X) = ‘read locked’
    no_of_reads(X) = no_of_reads(X)+1;
else {
    wait until Lock(X) = “unlocked” and the lock manager
    wakes up the transaction);
}
go to start;
```

NOTES**Write Lock(X):**

```
Start : If lock(X) = 'unlocked'
        Lock(X) ← 'locked';
else {
```

```
    Wait until Lock(X) = 'unlocked' and the lock manager wakes up the
transaction;
}
```

```
Go to start;
```

Unlock(X):

```
If lock(X) = 'write locked'
```

```
{
```

```
    Lock(X) ← 'unlocked';
```

```
    Wakeup one of the waiting transaction, if any;
```

```
}
```

```
else if Lock(X) = 'read locked'
```

```
{
```

```
    No_of_reads(X) ← No_of_reads -1;
```

```
    if no of reads(X)=0
```

```
{
```

```
        Lock(X) = 'unlocked';
```

```
        Wakeup one of the waiting transactions, if any;
```

```
}
```

```
}
```

The algorithms are fairly straightforward, except that during the unlocking operation, if a number of read locks are there, then all of them are to be unlocked before the unit itself becomes unlocked.

To ensure smooth operation of the shared / exclusive locking system, the system must enforce the following rules:

1. Readlock(X) or writelock(X) commands must be issued before any read or write operations are performed.
2. Writelock(X) command must be issued before performing any writetr(X) operation on the same data item.
3. An unlock (X) command must be issued after completion of all readtr(X).
4. A readlock(X) command must not be issued while holding readlock or writelock on X.
5. A writelock(X) command must not be issued while holding readlock or writelock on X.

Conversion of Locks

In some cases, it is desirable to allow lock conversion by relaxing the conditions (4) and (5) of the shared/ exclusive lock mechanism. That is, there should be

provision for a transaction holding one type of lock on a data item to be converted to some other type of lock. For example, if a transaction is holding a read lock on a data item X, it may be permitted to upgrade it to writelock. If no other transaction is holding a lock on a data item X, it can upgrade its read lock to write lock on issuing a writelock(X) command. Otherwise the transaction waits for the others to release their readlocks on X. Similarly, any transaction holding a write lock on X is allowed to downgrade its write lock to read lock. The above algorithm can be amended to accommodate lock conversion.

It is important to note that use of binary locks does not guarantee serializability. The reason behind this is that in some situations or combinations of them, a key holding transaction may unlock the unit prematurely due to many reasons. One situation could be wherein a transaction does not need a certain data unit and therefore unlocks it but may be indirectly writing into it via some other unit at a later date. This would give rise to ineffective locking performance and loss of serializability. Such serializability can be guaranteed by implementing two-phase locking.

(iii) Optimistic Locking and Pessimistic Locking

Optimistic Locking: The data is locked only when data is being saved. The lock is released after data is saved.

Pessimistic Locking: The data is locked when editing begins. The lock is released after the data is saved or discarded.

Optimistic locking is only for solving physical I/O conflict. It cannot be used to handle the concurrency mentioned above. In other words, it cannot provide a solution to update loss problem and uncommitted dependency problem.

11.2.2 Other Concurrency Control Protocols

Concurrency control protocols are distinguished into *conservative* (or *pessimistic*) and *aggressive* (or *optimistic*).

(i) **Conservative/Pessimistic protocols** are based on the assumption that conflicts are frequent, so a transaction has to get permission before performing an operation on a data item.

A permission is denied if there would be a conflict that can potentially lead to a non-serialisable execution. Depending on the scheme, a requesting transaction is either made to wait (*blocked*) until the permission can be granted, as in the case of *locking* methods, or aborted and restarted, as in the case of *timestamp* methods.

(ii) **Aggressive/Optimistic protocols** are based on the assumption that conflicts are rare, therefore, it is possible for transactions to perform conflicting operations without having to wait. When they try to commit, serializability is ensured by validating or certifying the transactions. A transaction that is validated implies that no conflicting operations have been executed and commit can be done.

NOTES

NOTES

Each proposed concurrency control protocol has been shown to exhibit superior performance under certain situations, but no single protocol can perform best in all situations.

However, commercial DBMS, have adopted the *strict two-phase locking* protocol, because of its simplicity and ease of implementation compared to other alternatives.

11.2.3 Two-Phase Locking Protocols

In 2PL, a transaction requests an appropriate lock just before performing an operation. If a transaction requests a conflicting lock, it is *blocked*, awaiting the release of the conflicting lock. Otherwise, the appropriate lock is granted to the transaction.

Each transaction is executed in two phases:

Phase 1: The *growing phase* during which the transaction acquires all the required locks, but cannot release any lock.

Phase 2: The *shrinking phase* during which the transaction releases its locks, but cannot request additional locks on any data item.

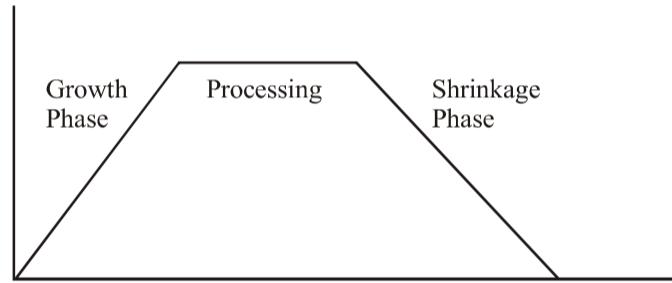


Fig. 11.2 Two-Phase Locking Strategy

readlock(Y)	
read(Y)	Growth Phase I
writelock(X)	
unlock(Y)	
read(X)	Shrinkage Phase II
X=X+Y	
write(X)	
unlock(X)	

No Downgrading Lock

According to these rules, a transaction cannot downgrade a write lock into a read lock, because downgrading is equivalent to releasing the write lock and subsequently requesting and acquiring a new read lock.

Upgrading a Lock is Accepted

Protocols

However, it is possible for a transaction to upgrade a read lock into the stronger write lock during the growing phase.

The order in which transactions are granted locks forces an execution ordering on the transactions in connection with their conflicting operations because transactions are forced to wait on conflicting locks.

By preventing transactions from acquiring any lock after the release of a lock, the 2PL protocol ensures serializability.

This can be illustrated by showing how the *non-repeatable read* problem is solved using 2PL.

Recall our example below, of inconsistent retrieval involving two transactions T and S and two columns A and B. We depict below their non-2PL and 2PL executions with time diagrams. We denote local variables with lower case letters and with * values produced by committed transactions. In the 2PL execution, we specify the required lock requests.

A Non-2PL Execution

Time	T	S	A	B	Sum
t1	v=read(A)		3*	5*	8*
t2		x=read(A)	3*	5*	8*
t3	v=v+3		3*	5*	8*
t4	write(A,v)		6	5*	11
t5	v= read(B)		6	5*	11
t6	v = v-3		6	5*	11
t7	write(B,v)		6	2	8
t8	commit		6*	2*	8*
t9		y=read(b)	6*	2*	8*
t10		output(x+y) →	5	◇	8*
t11		commit			

In the non-2PL execution, there is a cyclic ordering captured by the conflicting of read(A) by S with the write(A) of T, and the conflicting of write(B) by T with the read(B) of S.

In the 2PL execution, by requesting and acquiring a write lock on A at the beginning, T blocks S's read lock request, forcing S to wait until T commits and reads the consistent values of A and B that are produced by T. In the non-2PL, a non-serializable execution, S reads inconsistent values for A and B. It reads B from T, and A from another previously committed transaction.

NOTES

A 2PL Execution

Time	T	S	A	B	Sum
t1	v=read(A)		3*	5*	8*
NOTES	t2	x=read(A)	3*	5*	8*
	t1	writelock(A)			
	t2	v=read(A)	3*	5*	8*
	t3		readlock(A)	3*	8*
	t4	v=v+3	WAIT	3*	8*
	t5	write(A,v)	WAIT	6	11
	t6	writelock(B)	WAIT	6	11
	t7	v= read(B)	WAIT	6	11
	t8	v = v-3	WAIT	6	11
	t9	write(B,v)	WAIT	6	8
	t10	release-locks	WAIT	6	8
	t11	commit	WAIT	6*	8*
	t12		x=read(A)	6*	8*
	t13		readlock(B)	6*	8*
	t14		y=read(b)	6*	8*
	t15		output(x+y) →	8	8*
	t16		release-locks		
	t17		commit		

Advantages of 2PL

- It is easy to enforce. DBMS has to keep a track of only what phase a transaction is in.
- 2PL has been a major factor in the success of databases specially while handling concurrent transactions.
- It does not involve the real-time testing for serializability and the generation of precedence graphs.

Variations of 2PL

- Basic 2PL: This involves the growth and the release phase.
- Conservative 2PL: This involves acquiring of all locks at the beginning and releasing when done. It is inefficient, because you will request more than and longer than what is really need.
- Strict 2PL: There is a distinct growth phase, but the shrinkage phase is abrupt. Used by Oracle.

There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**. A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and *write-set*. As

mentioned earlier, the **write-set** is the set of all items that a transaction it writes and the **read-set** of a transaction is the set of all items that are read by the transaction. If any of the predeclared items required cannot be locked, the transaction does not lock any item. Instead, it waits until all the items are available for locking. Conservative 2PL is a deadlock-free protocol. However, it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in most situations.

In practice, the most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules. Here, none of the exclusive (write) locks are released by a transaction T till it has committed or aborted. Therefore, no item written by T can be read or written in by any other item until T commits. This leads to a strict recoverability schedule. Strict 2PL is not deadlock-free. A more restrictive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules. In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL. Notice the difference between conservative and rigorous 2PL; the former must lock all its items *before it starts* so once the transaction starts, it is in the shrinking phase, whereas the latter does not unlock any of its items until *after it terminates* (by committing or aborting) so the transaction is in its expanding phase until it ends.

In many cases, the **concurrency control subsystem** itself is responsible for generating the readlock and writelock requests. For example, if the system is to enforce the strict 2PL protocol, then, whenever transaction T issues a read(X), the system calls the readlock (X) operation on behalf of T. If the state of LOCK (X) is write-locked by some other transaction T', the system places T on the waiting queue for item X; otherwise, it grants the readlock(X) request and permits the read(X) operation of T to execute. On the other hand, if transaction T issues a writeitem (X), the system calls the writelock (X) operation on behalf of T. If the state of LOCK (X) is writelocked or readlocked by some transaction T', the system places T on the waiting queue for item X; if the state of LOCK (X) is readlocked and T itself is the only transaction holding the read lock on X, the system upgrades the lock to write locked and permits the write_item (X) operation by T; Finally, if the state of LOCK (X) is unlocked, the system grants the writelock (X) request and permits the write(X) operation to execute. After each action, the system must update its lock table appropriately.

Although the two-phase locking protocol guarantees serializability (that is, every schedule that is permitted is serializable), it does not permit *all possible* serializable schedules (that is, some serializable schedules will be prohibited by the protocol). In addition, the use of locks can cause two additional problems: deadlock and starvation.

NOTES

NOTES**Hierarchical Locking**

We can read/write lock data at various granularities.

- Entire database
- Relation
- tuple
- Disk page/block
- B-tree node
- Attribute value

Locking the entire database would reduce concurrency considerably. It may seem that maximal concurrency would be attained by just locking each attribute value, however, disk read/writes are usually done in pages or blocks, so there is no value to locking just a single attribute value, an entire block must be locked. But locking at this level requires more overhead on figuring out what to lock and where it is located on the disk. In some cases, we want to lock an entire relation, such as when we are reading an item in a B-tree. In such cases, we can't change the B-tree in the middle of looking for the item!

So let us modify our locking to look at a *tree* of items rather than just a single item. In this case, we can view the tree as either a tree where each node is independent (for example, a B-tree) or a tree where the children are subsets of the parent (e.g., a tree of the database down to tuple level).

A simple protocol for independent node trees is the following:

- Can only lock if the lock is currently held by parent (except for first item locked).
- A single transaction cannot lock the same item twice.

Schedules that obey this simple protocol are serializable. (General idea of proof given in reading.)

This won't work for database-relation-tuple trees since locking a parent implies that you have exclusive use of all the children. We need to also warn others that they can't lock children. We have three types of locks:

- (i) LOCK - Lock item and all descendants.
- (ii) WARN - No other transaction may lock (but may warn).
- (iii) UNLOCK - Removes lock/warning from item.

The warning protocol is as follows:

1. First locks or warns root.
2. Only locks/warns if it holds a warning on the parent (by the transaction).
3. Only removes locks/warns if no child is locked/warned (by the transaction).

4. Obeys two-phase protocol - All locks/warns first, then all unlocks.
5. Scheduler only allows a lock if no other transaction has a lock or warning on A, and allows a warning only if no other transaction has a lock on A.

Protocols

11.2.4 Concurrency Control Protocols based on Timestamp

NOTES

Timestamp Ordering

Another protocol that we can use to generate serial schedules is **timestamp ordering**. The general idea is to order the transactions by time, and to use this ordering to give priority to read and writes in earlier transactions. This will result in a serializable schedule that is equivalent to the serial schedule of executing each transaction in the order in which it arrived.

Step 1: Assign a **timestamp** to each transaction. This is either:

- A unique integer (increased for each transaction), or
- A system clock time (only one transaction permitted per tick).

Step 2: The next step is to associate.

- Associate a read-time RT with each item which is the latest transaction time at which an item has been read.
- Associate a write-time WT with each item which is the latest transaction time at which an item has been written.

We want a serial schedule which only reads/writes in the following conditions:

- We can only read an item if it was written by some transaction earlier than us.
- We can only write an item if it has yet to be read by some transaction later than us.

So, when we read an item, we have to check that the WT is less than our transaction timestamp. When we write an item, we have to check that the RT is less than our transaction timestamp. In both cases, we update the WT and RT if the operation is permitted. If we have violated one of the conditions then the transaction must *abort*.

TS(T): Timestamp of a transaction T.

Read_TS(X): Read timestamp of an item X. This is the largest timestamp among all the timestamps of transactions which have successfully read X.

Write_TS(X): Write timestamp of an item X. The largest timestamp among all the timestamps of transactions which have successfully written X.

Protocol

- Case 1: T requests READ(X).
- Case 1.1: If $TS(T) \geq Write_TS(X)$, execute READ(X) and update Read_TS(X).

NOTES

Case 1.2: If $TS(T) < Write_TS(X)$, abort T.

Case 2: T requests WRITE(X).

Case 2.1: If $TS(T) \geq Write_TS(X)$ and $TS(T) \geq Read_TS(X)$,
execute WRITE(X) and update Write_TS(X).

Case 2.2: If $Write_TS(X) > TS(T) \geq Read_TS(X)$, ignore WRITE(X)
and continue T.

Case 2.3: If $TS(T) < Read_TS(X)$, abort T.

Abortion of a transaction may causes abortion of another transaction which has
read a data item written by the aborted transaction called Cascading Rollback

Strict Timestamp Ordering

A variation of basic timestamp ordering called **strict** timestamp ordering ensures
that the schedules are both **strict** (for easy recoverability) and (conflict) serializable..
In this variation, a transaction T that issues a read_item(x) or write_item(x) such
that $TS(T) > write_TS(X)$ has its read or write operation *delayed* until the
transaction T that *wrote* the value of X (hence $TS(T) = write_TS(X)$) has committed
or aborted. To implement this algorithm, it is necessary to simulate the locking of
an item X that has been written by transaction T until T either commits or aborts.
This algorithm does not cause deadlock, since T waits for T' only if $TS(T) > TS(T')$.

Thomas's Write Rule. A modification of the basic TO algorithm, known
as **Thomas's write rule**, does not enforce conflict serializability; but it rejects
fewer write operations, by modifying the checks for the **write(X)** operation as
follows:

1. If $read_TS(X) > TS(T)$, then abort and roll back T and reject the operation.
2. If $write_TS(X) > TS(T)$, then do not execute the write operation but continue
processing. This is because some transaction with timestamp greater than
 $TS(T)$ and hence after T in the timestamp ordering-has already written the
value of X. Hence, we must ignore the **write(X)** operation of T because it
is already outdated and obsolete. Notice that any conflict arising from this
situation would be detected by case (1).
3. If neither the condition in part (1) nor the condition in part (2) occurs, then
execute the **write(X)** operation T and $write_TS(X)$ to $TS(T)$.

11.2.5 Concurrency Control using Multi-Versioning

We can observe that concurrency control is required when two transactions are
reading and writing to the same location. Problems arise because the correct
version of the data at that location may have been corrupted by the other transaction,
e.g., a transaction could read an *obsolete* value, that is, one that will be written in
future by an earlier transaction in the serialization order.

An obvious solution then is to keep around *all* versions of a data item, either indefinitely (a transaction-time database) or just during the lifetime of a set of concurrent transactions.

Keep not only the latest value X_1 of each data item X but also one or more old values X_2, X_3, \dots of the data item.

Reduce the probability of Cases 1.2 and 2.3 by letting transactions to access previous values of data items so that they preserve the order of their timestamps.

$\text{Read_TS}(X_i)$: The largest timestamp among all the timestamps of transactions which have successfully read the version X_i of item X .

$\text{Write_TS}(X_i)$: The largest timestamp among all the timestamps of transactions which have successfully written the version X_i of item X .

A new version of item X is created whenever $\text{WRITE}(X)$ is executed.

Its Write_TS and Read_TS are initialized to the timestamp of the transaction which has created it.

A version X_k is removed when $\text{Write_TS}(X_k)$ becomes smaller than $\max \{ \text{Write_TS}(X_j) \mid \text{Write_TS}(X_j) \leq \min\{\text{TS}(T)\} \}$.

Protocol

Case 1: T requests $\text{READ}(X)$.

Find the version X_i such that $\text{Write_TS}(X_i) = \max \{ \text{Write_TS}(X_j) \mid \text{Write_TS}(X_j) \leq \text{TS}(T) \}$,

Execute $\text{READ}(X_i)$, and update $\text{Read_TS}(X_i)$.

Case 2: T requests $\text{WRITE}(X)$.

Case 2.1: If there exists a version X_i with $\text{Write_TS}(X_i) = \max \{ \text{Write_TS}(X_j) \mid \text{Write_TS}(X_j) \leq \text{TS}(T) \}$ and $\text{TS}(T) < \text{Read_TS}(X_i)$, abort T .

Case 2.2: Otherwise, create a new version of X .

NOTES

Advantages

- Has access to previous versions of data.
- Aborts less frequently in timestamp ordering protocol.

Disadvantages

- Space blow-out (can mitigate with write-once storage media).
- Overhead on retrieving desired version.
- Overhead on maintaining versions.
- Doesn't solve concurrency control.

NOTES

Multiversion Two-Phase Locking using Certify Locks

In this multiple-mode locking scheme, there are three locking modes for an item: read, write, and certify, instead of just the two modes (read, write) discussed previously. Hence, the state of LOCK (X) for an item X can be one of read-locked, certify-locked, or unlocked. In the standard locking scheme, with only read and write locks, a write lock is an exclusive lock. We can describe the relationship between read and write locks in the standard scheme by means of the lock compatibility table shown in Figure 11.3 (a). An entry of ‘yes’ means that, if a transaction T holds the type of lock specified in the column header on item X and if transaction T’ can obtain the lock because the locking modes are compatible. On the other hand; an entry of ‘no’ in the table indicates that the locks are not compatible, so T’ must wait until T releases the lock.

(a)		Read	Write
	Read	Yes	No
	Write	No	No

(b)		Read	Write	Certify
	Read	Yes	Yes	No
	Write	Yes	No	No
	Certify	No	No	No

Fig. 11.3 Lock compatibility tables: (a) A compatibility table for read/write locking scheme. (b) A compatibility table for read/write/certify locking scheme

In the standard locking scheme, once a transaction obtains a write lock on an item, no other transactions can access that item. The idea behind multiversion 2PL is to allow other transactions T’ to read an item X while a single transaction T holds a write lock on X. This is accomplished by allowing *two versions* for each item X; one version must always have been written by some committed transaction. The second version X’ is created when a transaction T acquires a write lock on the item. Other transactions can continue to read the *committed version* of X while T holds the write lock. Transaction T can write the value of X’ as needed, without affecting the values of the committed version X. However, once T is ready to commit, it must obtain a **certify lock** on all items that it currently holds write locks on before it can commit. The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write-locked items are released by any reading transactions in order to obtain the certify locks. Once the certify locks – which are exclusive locks – are acquired, the committed version X of the data item is set to the value of version X’. Version X’ is discarded, and the certify locks are then released. The lock compatibility table for this scheme is shown in Figure 11.3 (b).

NOTES

In this multiversion 2PL scheme, reads can proceed concurrently with a single write operation – an arrangement not permitted under the standard 2PL schemes. The cost is that a transaction may have to delay its commit until it obtains exclusive certify locks on *all the items* it has updated. It can be shown that this scheme avoids cascading aborts, since transactions are only allowed to read the version X that was written by a committed transaction. However, deadlocks may occur if upgrading of a read lock to a write lock is allowed, and these must be handled by variations of the techniques discussed later on.

11.2.6 Validation (Optimistic) Concurrency Control Techniques

Optimistic concurrency control techniques are also known as validation or certification techniques. In such techniques no checking is done while the transaction is being executed. In many of the concurrency control methods, validation-based technique is used. Until the transaction reaches its end, no updates in the transaction are directly applied to the items in the database. The updates are applied to the local copies of transactions while the transaction is being executed.

After completion of the transaction a **validation phase** checks whether or not the serializability order is maintained. The system must keep certain information that the validation phase requires. The transaction is allowed to commit if it does not violate serializability. The local copies are used to update the database. However, if serializability is violated, the transaction aborts and restarts later.

There are three phases for this concurrency control protocol:

1. **Read phase:** In this phase, a transaction reads the values of committed data items from the database and applies updates on the local copies of data items. These local copies are kept in a workspace reserved for the transaction.
2. **Validation phase:** This phase checks whether the updates made to the database violate the serializability order.
3. **Write phase:** If the validation phase finds that application of updates on the database items does not disturb the serializability order, then the updates are made permanent on the database. Otherwise, the transaction is aborted discarding the updates. It is restarted later.

The aim of a validation-based protocol is to do all the checks at once. This also reduces the overhead of transaction until the validation phase. In systems with little interference between transactions, most of the schedules will be found to be serializable. However, systems including high interference between transactions will fail in the validation phase. As a consequence, many transactions will have to abort after their completion. The updates committed by them will have to be discarded. They will have to start later. Under these circumstances, this approach does not perform well. This technique assumes that there will not be much interference in the system. Hence, the schedule will be validated in the validation phase. Therefore, this approach is named as '**optimistic approach**'.

NOTES

The optimistic protocol uses transaction timestamps and maintains write_sets and read_sets of the transactions in the system. In addition, for each transaction, the start and end times for some of the three phases need to be kept.

The set of items that a transaction writes is referred to as the **write_set** of that transaction. Similarly, the set of items that a transaction reads is called the **read_set**.

For transaction T_i , the protocol checks to ensure that T_i does not interfere with any committed transactions or with any other transactions currently in their validation stage. This checking takes place in transaction T_i 's validation phase. This phase also checks that for each similar transaction T_j which is committed or which is in the validation phase, any one of the following conditions is met:

1. Write phase of transaction T_j gets completed before the read phase of transaction T_i begins.
2. T_i begins the write phase after the write phase of T_j is completed; and the read_set of T_i does not have any items in common with the write_set of T_j .
3. The read_set and write_set of T_i do not have any times in common with the write_set of T_j and T_j completes its read phase before the read phase of T_i .

While validating transaction T_i , the condition that is checked first is the first condition for each transaction T_j .

This is because of the following reasons:

- (i) It is the most simple condition to check.
- (ii) The second condition will be checked only if the first condition is false.
- (iii) The third condition which is the most complicated, is checked only if the second one is false.

There will be no interference and T_i is validated successfully if any one of these three conditions is met. However, if not even one of these three conditions holds, the validation of transaction T_i fails. AS interference may have taken place, it is aborted and restarted later.

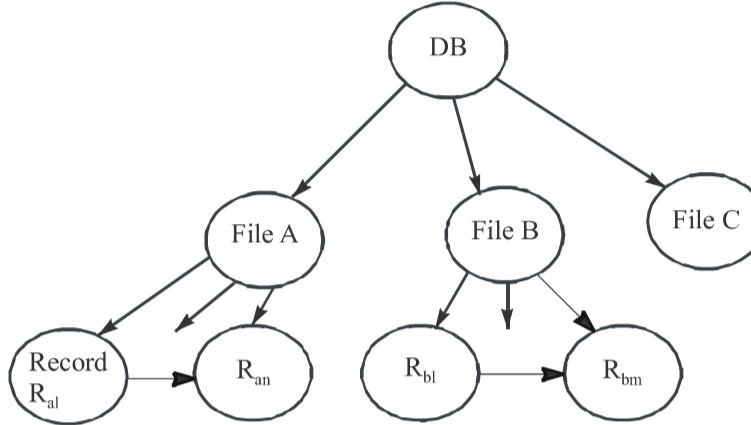
11.3 MULTIPLE GRANULARITY LOCKING PROTOCOL

In the concurrency control schemes described so far, we have used each individual data item as the unit on which synchronization is performed.

There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit. As an example , consider the tree shown in Figure 11.4, which consists of three levels of nodes. The highest level represent the entire database. Below it are nodes of type files ; the database consists of exactly these files. Finally , each file has nodes of

type record. As before , the file consist of exactly those records that are its child nodes and no record can be present in more than one file.

Protocols



NOTES

Fig. 11.4 Tree with Three Levels of Nodes

Each node in the tree can be locked individually either in shared or exclusive lock modes. When a transaction locks a node, in either shared or exclusive mode, then automatically the transaction gets hold of all the descendants of that node in the same lock mode. It uses two phase locking and makes use of the following types of locks:

- (a) **Exclusive (X) Lock**—When a Transaction is granted an X lock on a node ‘K’, it is implicitly granted X lock on all the descendants of node ‘K’.
- (b) **Shared (S) Lock**—When a Transaction is granted an S lock on a node ‘K’, it is implicitly granted S lock on all the descendants of node “K”.
- (c) **Intentional X (IX) Lock**—When a transaction needs X lock on a node “K”, the transaction would need to apply IX lock on all the precedent nodes of “K” starting from the root node. So, when a node is found locked in IX lock mode, it indicates that some of its descendant nodes must be locked in X mode.
- (d) **Intentional S (IS) Lock**—When a transaction needs an S lock on a node “K”, the transaction would need to apply IS lock on all the precedent nodes of “K” starting from the root node. So, when a node is found locked in IS lock mode, it indicates that some of its descendent nodes must be locked in S mode.
- (e) **SIX Lock**—When a node is locked in SIX mode, it indicates that the node is explicitly locked in S Mode and IX Mode. So, the entire tree rooted by that node is locked in S mode and some nodes in that are locked in X mode. This mode is compatible only with IS mode.

The Compatibility Matrix is as indicated in Table 11.1:

Table 11.1 Compatibility Matrix

NOTES

	<u>IS</u>	<u>IX</u>	<u>SIX</u>	<u>S</u>	<u>X</u>
<u>IS</u>	True	True	True	True	False
<u>IX</u>	True	True	False	False	False
<u>SIX</u>	True	False	False	False	False
<u>S</u>	True	False	False	True	False
<u>X</u>	False	False	False	False	False

The multiple granularity locking protocol, which ensures serializability, as follows:

Each transaction T_i can lock a node P by following these rules:

1. It must observe the lock compatibility as mentioned above.
2. It must lock the root of the tree first, and can lock it in any mode.
3. It can lock a node P in S or IS mode only if it currently has the parent of P locked in either IX or IS mode.
4. It can lock a node P in X, SIX, or IX mode only if it currently has the parent of P locked in IX or SIX mode.
5. It can lock a node P only if it has not previously unlocked any node.
6. It can unlock a node P only if it currently has none of the children of P locked.

Check Your Progress

1. State the ways in which serializability can be achieved?
2. List the levels of locking.
3. What are the types of locks?
4. What are the rules that the system must enforce to ensure smooth operation of the shared locking system?
5. Differentiate between optimistic locking and pessimistic locking.
6. Why is the strict two-phase locking protocol adopted?
7. What is timestamp ordering?
8. What is the read phase?

11.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Serializability can be achieved by several different means. Locking and time-stamping protocols are two very common techniques.

2. Typical levels of locking are:

- (i) Database locking
- (ii) Table locking
- (iii) Page locking
- (iv) Row locking
- (v) Column locking

3. Locks can be binary, shared/exclusive locks and certify locks.

4. To ensure smooth operation of the shared/exclusive locking system, the system must enforce the following rules:

- (i) Read lock(X) or write lock(X) commands must be issued before any read or write operations are performed.
- (ii) Write lock(X) command must be issued before performing any writetr(X) operation on the same data item.
- (iii) An unlock (X) command must be issued after completion of all readtr(X).
- (iv) A read lock(X) command must not be issued while holding readlock or write lock on X.
- (v) A write lock(X) command must not be issued while holding readlock or write lock on X.

5. In optimistic locking, the data is locked only when it is being saved. The lock is released after the data is saved. In pessimistic locking, the data is locked when editing begins. The lock is released after the data is saved or discarded.

6. The strict two-phase locking protocol is adopted by commercial DBMS because of its simplicity and ease of implementation compared to other alternatives.

7. Strict two-Phase Locking is a protocol that can be used to generate serial schedules. The general idea is to order the transactions by time, and to use this ordering to give priority to read and writes in earlier transactions. This will result in a serializable schedule that is equivalent to the serial schedule of executing each transaction in the order in which it arrived.

8. In read phase, a transaction reads the values of committed data items from the database and applies updates on the local copies of data items. These local copies are kept in a workspace reserved for the transaction.

Protocols

NOTES

11.5 SUMMARY

- Serializability can be achieved by several different means. *Locking* and *time-stamping* protocols are two very common techniques. Locking is a more

NOTES

pessimistic approach to concurrency control while time-stamping is a more optimistic approach.

- Lock is a variable associated with every data item to indicate whether the data item is available for an operation to be performed on it. Lock variable synchronizes the operations performed on these data items by the concurrent transactions.
- Locking ensures conflict prevention by holding a lock on various parts of a database. Sometimes, these locks are held by the DBMS; sometimes the DBA or a user himself holds the lock explicitly.
- Binary locking scheme has two states. The states are designated by ‘0’ and ‘1’. These two digits are used to indicate whether a data item is locked or unlocked.
- A binary lock is easy to implement, it also looks satisfactory enough but suffers from serious difficulties. It strictly prohibits more than one transaction to access a data item simultaneously. This scheme does not even permit more than one transaction to perform a read operation on the same data item simultaneously.
- Conservative/Pessimistic protocols are based on the assumption that conflicts are frequent, so a transaction has to get permission before performing an operation on a data item.
- Aggressive/Optimistic protocols are based on the assumption that conflicts are rare, therefore, it is possible for transactions to perform conflicting operations without having to wait.
- In 2PL, a transaction requests an appropriate lock just before performing an operation. If a transaction requests a conflicting lock, it is *blocked*, awaiting the release of the conflicting lock. Otherwise, the appropriate lock is granted to the transaction.
- Another protocol that we can use to generate serial schedules is **timestamp ordering**. The general idea is to order the transactions by time, and to use this ordering to give priority to read and writes in earlier transactions.
- In this multiple-mode locking scheme, there are three locking modes for an item: read, write, and certify, instead of just the two modes (read, write).
- Optimistic concurrency control techniques are also known as validation or certification techniques. In such techniques no checking is done while the transaction is being executed. In many of the concurrency control methods, validation-based technique is used. Until the transaction reaches its end, no updates in the transaction are directly applied to the items in the database.

11.6 KEY WORDS

- **Lock:** A variable associated with every data item to indicate whether the data item is available for an operation to be performed on it.
- **Explicit Lock Requests:** Requests issued if a different kind of lock is required during an operation.
- **Optimistic Locking:** Locking wherein data is locked only when it is being saved.
- **Strict Timestamp Ordering:** A variation of basic timestamp ordering which ensures that the schedules are both strict and serializable.

NOTES

11.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What is a timestamp? How does the system generate a timestamp?
2. Give an example to show that ‘two-phase locking protocol’ is not deadlock free.
3. Two-phase locking protocol ensures conflict serializable, but not converse. Give a schedule which is conflict serializable, but does not satisfy the two-phase locking protocol.
4. What are the ‘undo’ and ‘redo’ type log entries? Describe the write ahead logging protocol.
5. What are checkpoints? Why are they important?

Long Answer Questions

1. Describe the different types of locks—binary, shared, exclusive.
2. Describe timestamp-based concurrency control protocol.
3. Write short notes on
 - (a) Concurrency control
 - (b) 2PL protocol
 - (c) Recovery in DBMS
4. Discuss the problem of deadlock and starvation. Illustrate with example that the two phase locking protocol is not deadlock free.
5. Describe the two-phase locking protocol. How is it different from strict and conservative two-phase locking protocol?

NOTES

6. When a schedule satisfies the two-phase locking protocol, is there any possibility of dead lock? If not, why? If yes, explain with an example.
7. Prove that the basic two-phase locking protocol guarantees conflict serializability. What benefit is provided by strict two-phase locking? What are the resultant disadvantages? Discuss.

11.8 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

UNIT 12 RECOVERY AND ATOMICITY

NOTES

Structure

- 12.0 Introduction
- 12.1 Objectives
- 12.2 Types of Failure and Storage Structure
- 12.3 Database Recovery Techniques
- 12.4 Log Based Recovery
- 12.5 Recovery with Concurrent Transactions
- 12.6 Buffer Management
- 12.7 Failure with Loss of Non Volatile Storage
- 12.8 Advanced Recovery Techniques in a Database
- 12.9 Remote Backup Systems
- 12.10 Answers to Check Your Progress Questions
- 12.11 Summary
- 12.12 Key Words
- 12.13 Self Assessment Questions and Exercises
- 12.14 Further Readings

12.0 INTRODUCTION

A computer system, like any other device, there can be any case in database system when database failure happens. So, the data which is stored in database should be available all the time. There may be several reasons that data gets deleted, damaged or hacked by coincidence. Database system must take steps in advance to maintain the atomicity and durability of transactions and also to recover the data in case of failures.

In the event of a malfunction, the method of restoring the database to a correct state is known as the database restoration. It is a function that DBMS can provide to ensure the reliability of the database. Here, reliability implies both, the DBMS' resistance to different forms of failure and its ability to recover from them. The database needs to be restored again. From its inconsistent state, which was caused by loss, back to a stable state.

In this unit, you will study about the recovery and atomicity techniques of database recovery, log based recovery, recovery with concurrent transactions, buffer management, failure with loss of non-volatile storage and remote backup system.

NOTES

12.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the various types of database recovery techniques
- Understand recovery of database using log
- Describe buffer management
- Discuss the advance recovery systems and remote back-up systems

12.2 TYPES OF FAILURE AND STORAGE STRUCTURE

Different types of failure may occur in a database system and each of them requires to be handled in a different manner. Following are the different types of failure.

1. **Transaction failure:** we know that transaction is a logical work or program that may results in modification of data in a database. There are two types of errors that results in transaction failure.
 - (i) **System errors:** wherever the information system itself terminates an active transaction which is to prevent due to some system condition (may be deadlock) or not be able to execute it. The transaction can be re-executed later.
 - (ii) **Logical errors:** in this case the transaction fails due to some internal condition such data not found, overflow or bad input.
2. **System crash:** There are some problems which are external to the system. There is a hardware defect, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and transfer transaction processing to a halt.
3. **Disk failure:** A disk block drops its content as a result of either a head crash or failure during a data transfer operation. If we talk about the early days of technology evolution, then it had been a classic drawback where storage drives or hard-disk drives familiarized to failing oftentimes. Disk failures consists of the formation of unreachability to the disk , disk crash or the other failure , dangerous sectors which abolishes all or a section of disk storage.

Storage Structure

There are different types of storage structure given as follows:

1. **Volatile storage:** Volatile storage devices are placed near to the CPU and are surrounded on the chipset itself. A volatile storage cannot survive after system crashes. The main memory and cache memory are examples of the memory board which will store a merely little quantity of knowledge.

- 2. Non-volatile storage:** Non-volatile storage memories are created to persist system crashes. These devices slower in the accessibility but enormous in information storage capability. Some of the examples are flash memory, hard-disks, non-volatile RAM, magnetic tapes.

NOTES**12.3 DATABASE RECOVERY TECHNIQUES**

Recovery control techniques help recover the database from catastrophic as well as non-catastrophic failures. In case of recovering the database from catastrophic failures, the recovery method restores the previous copy of transactions from the logs and reconstructs a more stable database. The log contains a class of records called checkpoints, which are periodically written to the log at the time when the system updates the database and all buffers are modified. When checkpoints are read from the log, the following actions are performed:

- All the transactions are suspended temporarily.
- All buffers, which are modified, are written to the disk.
- A record known as checkpoint is written to the log which, in turn, is written to disk.
- All the transactions, which were suspended, are resumed.

Recovery method performs the undo and redo operations on the transaction to make the database consistent when it is affected by the non-catastrophic failures.

There are two types of recovery control techniques:

- (i) Deferred Update
- (ii) Immediate Update

(i) Deferred Update

The deferred update technique does not update the database physically until the transaction is committed. Before committing, recording of all the transaction updates is done in local transaction workspace known as buffers. During the commit phase, updates are first stored in the log and are subsequently applied to the database. In the event of failure of transaction before committing, the database is not affected and no UNDO operation is required. The REDO operation is performed to apply the changes from log to the database if the system fails. This technique is also called the NO-UNDO/REDO algorithm.

Recovery using Deferred Update in Single-User Environment: This process of recovery in a single-user environment uses an algorithm, which is called RDU_S algorithm. This algorithm uses two lists of transactions, the committed transaction and the active transaction. It also uses the REDO procedure, which redoing the WRITE_ITEM operation, WRITE-OP and consists of examining the following log entry [write_item, T, X, new_value], setting the value of data item, say X, in the database replaced by a new value.

NOTES

Recovery using Deferred Update in Multi-User Environment: In the recovery that uses deferred update in a multi-user environment, one can consider a system in which concurrency control uses the strict two-phase locking. It uses an algorithm called the RDU_M algorithm. This algorithm redoing all WRITE operations of the committed transactions in the log, in the same order in which they were written. Transactions, active but not committed, are cancelled and must be restarted. Figure 12.1 shows the recovery in a multi-user environment.

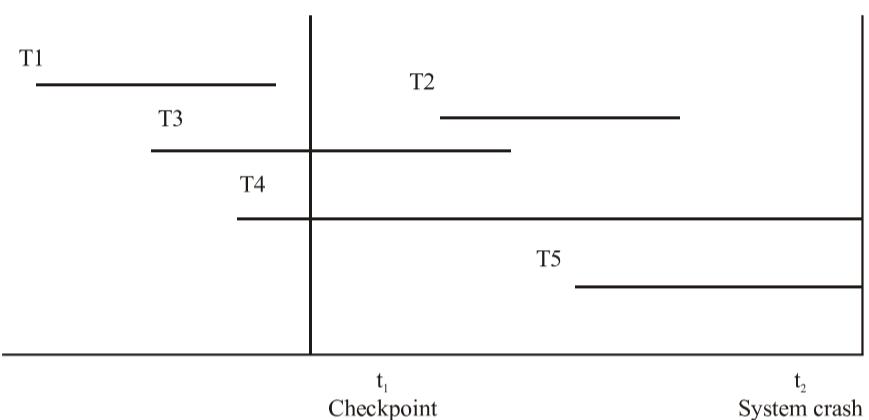


Fig. 12.1 Recovery in Multi-User Environment

(ii) Immediate Update

In this kind of update, the database is updated using some operations of a database before the transaction is committed. These operations are stored in a log before applying to the database. In case it is aborted after storing changes in the database, the system needs to perform the undo operation. Immediate update is also called the UNDO/REDO algorithm.

Recovery using Immediate Update in Single-User Environment: In this technique, the RIU_S algorithm, which uses the REDO and UNDO procedures, is used. The RIU_S algorithm performs the following tasks:

- It uses two lists of transactions, a committed transaction and an active transaction.
- It undoes all the WRITE_ITEM operations of the active transactions from the log using the UNDO procedure.
- It redo each of the WRITE_ITEM operations of the committed transactions noting from the log using the REDO procedure.

Recovery using immediate update in multi-user environment: In this method, the RIU_M algorithm, which uses the REDO and UNDO procedures, is used. The RIU_M algorithm performs the following tasks:

- It uses two lists of transactions, a committed transaction and an active transaction.

- It undoes each of the WRITE_ITEM operations of the active transactions in the log using the UNDO procedure. The operations are undone in the reverse order.

It redoing all WRITE_ITEM operations on the committed transactions using the log applying the REDO procedure, in the same order in which they are entered in the log.

Recovery and Atomicity

NOTES

Check Your Progress

1. What are the two types of transaction failure?
2. What are the two types of database recovery control techniques?

12.4 LOG BASED RECOVERY

As we know that, Atomicity property of DBMS states that either all the operations of transactions must be performed or none. The modifications must be made permanent in the database by committed transaction but not by an aborted transaction.

Log and Log Records

The log is a structure of log records and recording all the update activities in the database. Logs for each transaction are maintained in a stable storage. Any operation which is made on the database is documented is on the log. An update log record is created to reflect that modification, before performing any modification to database.

An update log record can be in the form of: $\langle T_i, X_j, V_1, V_2 \rangle$ has these fields:

- **Transaction identifier:** It is a unique identifier of the transaction performed.
- **Data item:** Data item is a unique identifier of the data item written.
- **Old value:** Old value i.e. value of data item earlier to write.
- **New value:** New value i.e. value of data item afterward write operation.

Other types of log records are:

1. **$\langle T_i \text{ start} \rangle$:** It comprises the information about when a transaction T_i starts.
2. **$\langle T_i \text{ commit} \rangle$:** It comprises information about when a transaction T_i commits.
3. **$\langle T_i \text{ abort} \rangle$:** It comprises information about when a transaction T_i aborts.

Undo and Redo Operations

The system has available both the old value earlier to changes of data item and new value that is to be written for data item. It is because of all database modifications must be preceded by creation of log record. This permits the system to accomplish redo and undo operations as suitable:

NOTES

- 1. Undo:** sets the data item specified in log record to old value by using a log record.

- 2. Redo:** sets the data item specified in log record to new value by using a log record.

The database can be modified using two approaches:

- 1. Immediate Modification Technique:** If database alteration done while transaction is still active then it is said to use immediate modification technique.

- 2. Deferred Modification Technique:** If the transaction does not change the database until it has partially committed then it is said to use deferred modification technique.

Whenever a system crash occurs, the system checks the log to find which transactions need to be undone and which need to be redone.

Use of Checkpoints

We must consult the log, when a system crash occurs. Using the checkpoints require to search the entire log to find out this information. But there are two major problems with this method:

1. Most of the transactions might requires to be redone which have previously written their updates into the database. Even though redoing them will have no reason to harm.
2. The search process/method is time-consuming.

So here, user familiarize with checkpoints to reduce these types of overhead. A log record in the form of <checkpoint L> which is used to represent a checkpoint in log. Where L is a list of transactions active at the time of the checkpoint. When a checkpoint log record is added to log all the transactions that have committed earlier. So this checkpoint have < T_i commit> log record before the checkpoint record. Any database alterations prepared by T_i is written to the database either earlier to the checkpoint or as part of the checkpoint itself. So, there is no need to perform a redo operation on T_i , at recovery time.

The system examines the log to find the last <checkpoint L> record, after a system crash has occurred. The undo or redo operations essential to be applied only to transactions in L. These operations needs to be applied on all transactions that started execution subsequently the record was written to the log. Assume that we denote this set of transactions as T. As mentioned in recovery using Log records, same rules of undo and redo are applicable on T.

The user required to only inspect the part of the log starting with the last checkpoint log record to find out whether a commit or abort record occurs in the log for each transaction in T or to find the set of transactions T. Consider for example, there is the set of transactions { T_0, T_1, \dots, T_{100} }. Assume that the latest checkpoint took place through the execution of transaction T_{57} and T_{59} ,

NOTES

12.5 RECOVERY WITH CONCURRENT TRANSACTIONS

The system has a single disk buffer and a single log, nonetheless of the number of concurrent transactions. We permit a buffer block to have data items which are updated by one or more transactions and also allow immediate modification. All transactions share the buffer blocks.

Interaction with Concurrency Control

The recovery scheme can be influenced by the concurrency-control scheme that is used. We must undo the updates performed by the transaction, that is, to roll back a failed transaction. Suppose that a transaction T_0 has to be rolled back, and a data item Q that was updated by T_0 has to be brought back to its old value. We restore the value by using the undo information in a log record, using the log-based schemes for recovery. Consider that a second transaction T_1 has achieved yet another update on Q before T_0 is rolled back. So the result will be an updation performed by T_1 will be lost if T_0 is rolled back.

Therefore, we require that, if a transaction T has updated a data item Q then no other transaction may update the same data item until T has committed or been rolled back. We can guarantee this requirement easily by using strict two-phase locking. It means, two-phase locking with exclusive locks is to be detained until the end of the transaction.

Transaction Rollback

Consider we roll back a failed transaction, using the log, say, T_i . The system scans the log for every log record of the form $\langle T_i, X_j, V_1, V_2 \rangle$ backward and found in the log, the system restores the data item X_j to its old value V_1 . Scanning of the log terminates when the log record $\langle T_i, \text{start} \rangle$ is found.

A transaction may have updated a data item more than once we know that scanning the log backward is important, since. Consider the pair of log records

$\langle T_i, A, 10, 20 \rangle$

$\langle T_i, A, 20, 30 \rangle$

The log records denote a modification of data item A by T_i which is monitored by another modification of A by T_i . Scanning the log backward sets A correctly to 10. If the log were scanned in the forward direction, then A would be set to 20, which is incorrect.

NOTES

Locks held by a transaction T may be released only after the transaction has been rolled back, if strict two-phase locking is used. Once transaction T updated a data item, no other transaction could have updated the same data item. So, restoring the old value of the data item will not erase the effects of any other transaction.

Checkpoints

Checkpoints are used to reduce the number of log records and when it recovers from a crash then the system must scan. It was essential to believe only the following transactions during recovery, because here we assumed no concurrency:

- If any, that is one transaction that was active at the time of the most recent check-point.
- Those transactions that started after the most recent checkpoint.

Some transactions may have been dynamic at the time of the most recent checkpoint, so when transactions can execute concurrently, then this situation is quite complicated.

In a concurrent transaction-processing system, we imagine that transactions do not act upon updates either on the log or the buffer blocks although the checkpoint is in progress. During transaction processing system, we need that the checkpoint logs record which is to be of the form \langle checkpoint L \rangle . Where L is a list of transactions active at the time of the checkpoint.

Because the transaction processing will have to close down while a checkpoint is in progress so the requirement that transactions must not act upon any updates to the log or to buffer blocks during check pointing can be bothersome. A check point where transactions are allowed to make updates even while buffer blocks are being written out it is known as a fuzzy checkpoint.

Restart Recovery

After the system recovers from a crash then it constructs two lists that is the undo-list having a list of transactions to be undone, and the redo-list having a list of transactions to be redone.

So the system initially having two lists which are both empty. The system scans the log backward and investigative each record until it obtains the first one \langle checkpoint \rangle record like:

- It adds T_i to redo-list, in case, for each record found of the form $\langle T_i \text{ commit} \rangle$.
- It adds T_i to undo-list, if T_i is not in redo-list, in case, for each record found of the form $\langle T_i \text{ start} \rangle$.

The system checks the list L in the checkpoint record after investigating all the appropriate log records. If T_i is not in redo-list then it adds T_i to the undo-list, this is the case for each transaction T_i in L .

After the redo-list and undo-list have been constructed, so the next step is to do recovery which has to be done as following:

1. In this phase, log records of transactions on the redo-list are ignored. For every transaction T_i in the undo-list, the scan stops when the $\langle T_i \text{ start} \rangle$ records have been found. The system rescans the log from the most recent record backward and execute an undo for each log record that belongs transaction T_i on the undo-list.
2. If the checkpoint record was passed in step 1, then this step may involve scanning the log forward. The system place the most recent $\langle \text{checkpoint } L \rangle$ record on the log.
3. The system scans the log forward from the most latest $\langle \text{checkpoint } L \rangle$ record. It also achieves redo for each log record that belongs to a transaction T_i that is on the redo-list. In this phase, the systems do not take into account the log records of transactions on the undo-list.

It is essential part in step 1 which is used to process the log backward, which results the state of the database is correct.

On the undo-list, after the system has undone all transactions then it redoes those trans- actions on the redo-list. It is necessary part to process the log forward. So after the recovery process has finished then the transaction processing resumes.

Before redoing transactions in the redo-list, it is essential to undo the transaction in the undo-list. By using the algorithm which is in steps 1 to 3 a problem may occur. Consider an example: Suppose that data item A which has initially has the value 30. Assume that a transaction T_i which is updated data item A to 50 and then aborted. Rollback of transaction would restore A to the value 30. Suppose that another transaction T_j then updated data item A to 60 and then committed and after that the system not working. At the time of the crash, the state of the log is

$\langle T_i, A, 30, 50 \rangle$
 $\langle T_j, A, 30, 60 \rangle$
 $\langle T_j \text{ commit} \rangle$

If we apply the operation of redo pass first then, A will be set to 60. So in the undo pass, A will be set to 30, but this is not the correct one method. The final value of Q should be 60, which we can be assured by performing undo before performing redo.

NOTES

12.6 BUFFER MANAGEMENT

Storage disk devices used for permanent storage is very slow as compared to temporary storage device such as memory. Thus, to achieve practical and sound performance, a database management system tries to improve the performance of the memory. Any work that is done in the memory will be erased as soon as

NOTES

power is lost, so to ensure the durability one of the ACID property, it ensures that all the committed work gets saved on to the disk. In order to manage the disk-memory interfaces the buffer manager divides most of the database memory into buffers.

There are two types of buffering done by a buffer manager:

- 1. Log-record buffering:** In this buffering method, instead of directly storing the records on to the stable storage, log records are buffered in main memory. We transfer the log records to the stable storage only when the buffer is full, or a log force is executed. Log force is an operation performed to commit a particular transaction by forcing all its log records including the commit record to update on the stable storage. Several such log records can thus be sent to a stable storage using a single output operation, thus reducing the I/O cost.

The rules below must be followed if log records are buffered:

- i. Log records are output to stable storage in the order in which they are created.
- ii. Transaction T_i enters the commit state only when the log record $\langle T_i \text{ commit} \rangle$ has been output to stable storage.
- iii. Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage. This rule is called the write-ahead logging or WAL rule. It requires only undo information to be output.

- 2. Database buffering:** this type of buffering can be implemented either in an area of real main-memory reserved for the database, or in virtual memory. Implementing buffer in reserved main memory has several drawbacks as memory is partitioned before-hand between database buffer and applications, limiting flexibility. Whereas if buffer is implemented within the virtual memory then the OS should not write out the dbase buffer pages itself, but should request the dbase system to force-output the buffer block. The database system in turn would force-output the buffer blocks to dbase after writing relevant log records to stable storage.

Database maintains an in-memory buffer of data blocks. This is achieved as follows:

- When a new block is needed; if buffer is full an existing block needs to be removed from buffer.
- If the block chosen for removal has been updated, it must be output to disk.
- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first (Write ahead logging)

NOTES

- No updates should be in progress on a block when it is output to disk. This can be ensured by implementing latches. Latches are locks held for short duration. They ensure that before writing a data item, a transaction acquires exclusive lock on block containing the data item; Lock can be released once the write is completed.

Check Your Progress

3. Define log.
4. What are the two types of buffering done by a buffer manager?

12.7 FAILURE WITH LOSS OF NON VOLATILE STORAGE

We have observed the case where a failure results in the loss of information belongs to volatile storage whereas the content of the non-volatile storage remains unaffected. Failures in which the content of non-volatile storage is lost are unusual. But there is need to be ready to handle this type of failure.

The objective is to dump the whole content of the database on a stable storage from time to time like, once per day. Suppose, if a failure take place which results in the loss of physical database blocks. The system will use the latest dump to bring back the database to an earlier consistent state. After this re-establishment, the system uses the log which bring the database system to the consistent state.

Surely, there is no transaction which may be active throughout the dump procedure. A technique related to check pointing must take place:

1. The records which are currently residing in main memory having output of all log records onto stable storage.
2. It is necessary to output all buffer blocks onto the disk.
3. Duplicate the contents of the database to steady storage.
4. Onto the steady storage, output a log record that is <dump>.

An archival dump, is a dump of the database contents. In the meantime, we can archive the dumps and use them later to inspect old states of the database. Dumps of a check pointing and database of buffers are similar. The simple dump method discussed here is very costly because of the following two reasons.

1. The whole database must be copied to steady storage and the result will be significant data transfer.
2. In the meantime, during the dump procedure, transaction processing is halted because of that CPU cycles are wasted. So some schemes have been developed which is called Fuzzy dump. It states that while the dump is in progress, while the transactions are active.

12.8 ADVANCED RECOVERY TECHNIQUES IN A DATABASE

NOTES

Recovery algorithms are methods which are used to guarantee durability and transaction atomicity despite failures. Using recovery algorithm, the recovery subsystem guarantees atomicity by undoing the actions of transactions which do not commit. Durability makes sure that even if failures occur, all actions of committed transactions survive.

The recovery methods involves that its logs stored data buffers into disk and flushing as:

- Different methods can be used when flushing occurs.
- **Shadowing:** Multiple versions of data items can be maintained by writing the updated buffer at a different disk location.
- **In-place updating:** Overwriting the old value on disk by writing the buffer back to the same original disk location.

There are two main methods in recovery process:

- a) Log-based recovery using WAL protocol.
- b) Shadow-paging

1. Write-Ahead Logging (WAL)

It is a technique which is used to provide atomicity and durability in the databases. The changes are first recorded in the log and then on stable storage before writing on the database. All the modification are written on the log before they are applied. Consider a situation that a system is performing some operation and it fails due to some reason. On restart, it will check the log to find the status of the job. On the basis of this comparison, it will undo or redo the things.

2. Steal/no-steal— Force/no-force Approaches

No-steal approach

In this approach, updates are made to data items in pages cannot be written to disk until the transaction has committed.

Steal approach

- 1) An updated buffer can be written before the transaction commits.
- 2) Advantage: To avoid the requirement for a very large buffer space.

Force/No-Force approaches

In a force approach, all pages which were updated at the commit point of a transaction by the transaction are written to permanent secondary storage immediately. While in case of no-force approach, after the transaction commits, pages need not be written to disk immediately.

NOTES**3. Shadow Paging**

Shadow paging is another technique to log-based recovery. This is useful if transactions execute one after the other. The main idea of shadow paging is to maintain 2 page tables throughout the transaction one is the current page table and other is the shadow page table. Often the shadow page table is stored in a non-volatile storage device, so that in case of failure we can at least recover the state of database prior to the start of the transaction. During the execution of the transaction shadow page table is never modified. At the start of the transaction both the page tables are the same. Only current page table is used for accessing the data item during the transaction execution. Whenever any page is used to write for the first time, always a copy of that page is made. The current page table then points to the copy and always updates are performed on the copy.

Example of Shadow Paging: In figure 12.2 Shadow page table is untouched throughout the life span of a transaction. A write operation performed on page 4 is updated on the current page tables.

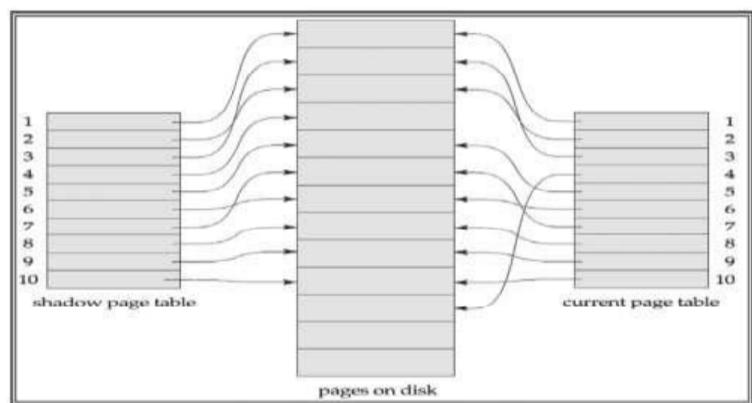


Fig. 12.2 Concept of Shadow Paging.

Whenever a transaction is complete and need to commit a transaction. We send all modified pages from main memory to disk. Now the current page table will become the new shadow page table. In order to achieve this we keep a pointer to the shadow page table at a fixed (known) location on disk and simply update the pointer to point to current page table on disk. Once pointer to shadow page table has been written, transaction is committed. So now even if a crash occurs no recovery is needed as the new transactions can start directly with the shadow page table. Pages that are on the disk and not pointed to from current/shadow page table should be removed.

Advantages and Disadvantage of Shadow-Paging

Advantages: As compared to log based, in shadow paging there is no overhead of writing log records

NOTES**Disadvantages:**

- Copying the entire page table to create a shadow page table is very expensive.
- As we need to push every updated page and page table on the disk the commit overhead is high.
- Individual pages may be updated which leads to Data fragmentation as related pages get separated on disk.
- After completion of every transaction, all the pages containing old versions of modified data needs to be removed.
- It is very hard to extend algorithm in order to allow concurrent transactions.

12.9 REMOTE BACKUP SYSTEMS

As we know that that the out-dated transaction-processing systems are centralized or client-server systems. These types of systems are susceptible to environmental disasters such as earthquakes, fire or flood. So, there is a requirement for transaction-processing systems which can function in case of environmental disasters or system failures. The time for which the system is unusable must be extremely small. These types of systems must be available all the time. Performing transaction processing at one site, called the primary site can achieve high availability, while a remote backup site where all the data from the primary site are simulated. The remote backup site is also known as secondary site. As updates are performed at the primary, the remote site must be kept synchronized with the primary site. We attain synchronization by sending all log records from primary site to the remote backup site. The remote backup site which should be physically separated from the primary like, for example, a disaster at the primary does not damage the remote backup site because we can locate it in a different site.

The remote backup site takes over processing only when the primary site fails. It performs recovery and this is done by using its copy of the data from the secondary site. With the impact of this, the remote backup site is doing recovery actions.

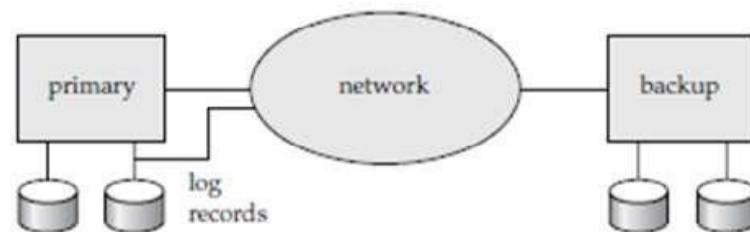


Fig.12.3 Architecture of Remote Backup System

There are different problems which must be addressed in designing a remote backup system:

- **Transfer of control:** The backup site takes over processing and becomes the new primary, only when the primary fails. And when the original primary site make progress, it can either take over the role of primary site again or play the role of remote backup. The easiest method of transferring control is for the old primary to receive redo logs from the old backup site. It also ensures to catch up with the updates by applying them locally. By this procedure, the old primary can act as a remote backup site.
- **Detection of failure:** It is important for the remote backup system to find when the primary has failed. Damage of communication lines which can fool the remote backup into believing that the primary has failed. So the solution of this problem is, maintain different communication links with independent modes of failure which is between the primary and the remote backup. For example, in addition to the network connection, with services provided by different telecommunication companies, there may be a separate modem connection over a telephone line. So these types of connections can be backed up with the help of manual intervention by operators and these operators can communicate over the telephone system.
- **Time to recover:** Consider if the log at the remote backup raises very large, in that case, the recovery can takes place a long time. As a result, the adjournment before the remote backup which takes over that can be significantly reduced. So the solution is a hot-spare configuration which can make overthrow by the backup site almost instantaneous.
- **Time to commit:** A transaction must not be declared committed until its log records have reached the backup site, to ensure that the updates of a committed transaction are durable. This delay can result some systems permit lower degrees of durability and longer wait to commit a transaction. The degrees of durability can be classified as follows.
 - a) **One-safe.** A transaction obliges as soon as its commit log record, which is written to stable storage at the primary site. But the main problem arise is that when the backup site takes over processing, then the updates of a committed transaction may not have made it to the backup site. So, the updates can be lost.
 - b) **Two-very-safe.** As soon as its commit log record is written to stable storage at the primary and the backup site then a transaction commits. The problem arises with this method is that if either the primary or the backup site is down, then transaction processing cannot proceed.
 - c) **Two-safe.** If both primary and backup sites are active, then this method is the same as two-very-safe. The transaction is allowed to commit as

Recovery and Atomicity

NOTES

NOTES

soon as its commit log record is written to stable storage at the primary site, only if only the primary is active. While avoiding the problem of lost transactions faced by the one-safe scheme, this method provides better availability than two-very-safe. So this results in a slower commit than the one-safe scheme.

Check Your Progress

5. What is Write-Ahead Logging (WAL)?
6. Write a note on shadow paging.

12.10 ANSWERS TO CHECK YOUR PPROGRESS QUESTIONS

1. A transaction can fail due to system errors and logical errors.
2. There are two types of recovery control techniques:
 - (i) Deferred update
 - (ii) Immediate update
3. The log is a structure of log records and recording all the update activities in the database.
4. There are two types of buffering done by a buffer manager:
 - (i) Log-record buffering
 - (ii) Database buffering
5. Write-Ahead Logging (WAL) is a technique which is used to provide atomicity and durability in the databases. The changes are first recorded in the log and then on stable storage before writing on the database.
6. Shadow paging is another technique to log-based recovery. This is useful if transactions execute one after the other. The main idea of shadow paging is to maintain 2 page tables throughout the transaction one is the current page table and other is the shadow page table.

12.11 SUMMARY

- Recovery control techniques help recover the database from catastrophic as well as non-catastrophic failures. In case of recovering the database from catastrophic failures, the recovery method restores the previous copy of transactions from the logs and reconstructs a more stable database.
- The deferred update technique does not update the database physically until the transaction is committed. Before committing, recording of all the transaction updates is done in local transaction workspace known as buffers.

NOTES

- In immediate update, the database is updated using some operations of a database before the transaction is committed.
- The log is a structure of log records and recording all the update activities in the database. Logs for each transaction are maintained in a stable storage. Any operation which is made on the database is documented is on the log.
- After the system recovers from a crash then it constructs two lists that is the undo-list having a list of transactions to be undone, and the redo-list having a list of transactions to be redone.
- Log-record buffering instead of directly storing the records on to the stable storage, log records are buffered in main memory. We transfer the log records to the stable storage only when the buffer is full, or a log force is executed.
- Database buffering can be implemented either in an area of real main-memory reserved for the database, or in virtual memory.
- Write-Ahead logging is a technique which is used to provide atomicity and durability in the databases. The changes are first recorded in the log and then on stable storage before writing on the database.
- The main idea of shadow paging is to maintain 2 page tables throughout the transaction one is the current page table and other is the shadow page table. Often the shadow page table is stored in a non-volatile storage device, so that in case of failure we can at least recover the state of database prior to the start of the transaction.

12.12 KEY WORDS

- **Log:** It is a structure of log records and recording all the update activities in the database.
- **Backup:** The process of copying of files to secondary devices.

12.13 SELF ASSESSMENT QUESTIONS AND EXERCISES**Short Answer Questions**

1. What are the different types of failure?
2. What do you understand by transaction rollback?
3. Discuss the concept of remote back-up system.

Long Answer Questions

1. Explain the database recovery techniques based on deferred and immediate update.

NOTES

2. What do you understand by log based recovery? Explain.
3. What is buffer management? Explain the two types of buffering done by a buffer manager.
4. Explain the advance recovery techniques.

12.14 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

BLOCK - V STORAGE

NOTES

UNIT 13 DATA ON EXTERNAL STORAGE

Structure

- 13.0 Introduction
- 13.1 Objectives
- 13.2 File Organization and Indexing
 - 13.2.1 Indexed Files
 - 13.2.2 Single Level Ordered Index
 - 13.2.3 Primary Index
 - 13.2.4 Clustering Index
 - 13.2.5 Secondary Index
 - 13.2.6 Non-Key Secondary Index
 - 13.2.7 Multi-Level Index
- 13.3 Index Data Structures
 - 13.3.1 Hash Based Indexing
 - 13.3.2 Tree Based Indexing
 - 13.3.3 Comparison of File Organization
- 13.4 Answers to Check Your Progress Questions
- 13.5 Summary
- 13.6 Key Words
- 13.7 Self Assessment Questions and Exercises
- 13.8 Further Readings

13.0 INTRODUCTION

A device that holds all the addressable data storage that is not within the main storage or memory of a computer is an external storage device, often referred to as auxiliary storage and secondary storage. Removable or non-removable, temporary or permanent, and available over a wired or wireless network may be an external storage unit.

External storage allows users to store data independently at a comparatively low cost from the main or primary storage and memory of a device. Without having to open up a device, it increases storage space.

An ultimate view of the stored data is provided by a database system. However, data is stored in various storage devices in the form of bits and bytes.

In this unit, you will study about the data on external storage, indexes, file organization and indexing, cluster indexes, primary and secondary indexes, index

NOTES

data structure, hash based indexing, tree base indexing and comparison of file organizations.

13.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the significance of indexing
- Explain the different types of indexing
- Understand the index data structures

13.2 FILE ORGANIZATION AND INDEXING

When files are classified and given an index so that retrieval is easy, it is called indexing.

13.2.1 Indexed Files

Hashing is a computational technique for organizing files. Records in hashed files can be stored and retrieved quickly. However, the difficulty in hashed files is that they are difficult to process in key order, which is important if you want to access all records with keys in a certain range.

Indexing is a data structure-based technique for accessing records in a file. Indexes are auxiliary access structures, which are used to speed up the retrieval of records in response to *certain* search conditions. A main file of records can be supplemented by one or more indexes.

Index structures provide secondary access paths, which provide alternative ways of accessing the records without affecting the physical placement of records on the disk. Indexes may be part of the main file or separate files, and may be created and destroyed as required without affecting the main file.

Indexes allow for efficient access to records based on the indexing fields that are used to construct the index. Any field of the file can be used to create an index, and multiple indexes on different fields can be constructed on the same file.

13.2.2 Single Level Ordered Index

Ordered access structures are similar to indexes in textbooks. The indexes or tables of contents list the important terms in alphabetical order along with page numbers where the information can be found.

You can search an index to find the address (page numbers in this case) and locate the term by searching the appropriate page. The alternative is to

complete a linear search of the textbook, looking through each page one at a time.

Data on External Storage

For a file consisting of several fields, and index access structure is defined on a single field which is referred to as **indexing field**. An index is usually composed of two components, the index field value as well as a list of pointers to all disk blocks containing records with that field value. The values in the index are ordered, so that a binary search on the index can be done. The index file is a lot smaller than the data file. Therefore, the binary search is much quicker.

There are different types of indexes; these include:

- Primary index
- Clustering index
- Secondary index

13.2.3 Primary Index

A primary index is an ordered file whose records are of fixed length with two fields. The first field is the primary key of the main data file, and the second is a pointer to a disk block. There is one index entry in the index file for each **block** in the data file. Each index entry has the value of the primary key field for the first record in the block and a pointer to the block.

NOTES

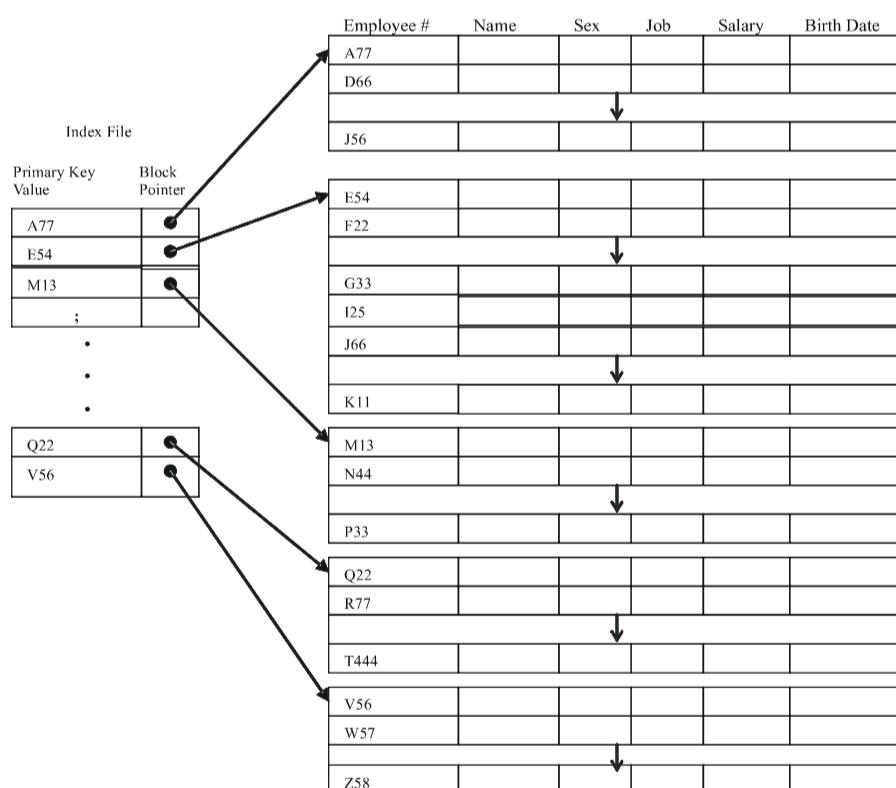


Fig. 13.1 Primary Index

NOTES

Each entry in the index has an employee id value and a pointer. The total number of entries in the index is the same as the number of disk blocks in the ordered data file.

$\langle K(1) = (A77), P(1) = \text{address of block 1} \rangle$

The first record in each block of the data file is called the anchor record or the block anchor.

Indexes can be characterized as dense or sparse.

Dense index: This has an index entry for each search key value, that is, each record in the data file.

Sparse (non-dense) index: It has index entries for only some of the search key values. A primary index is a sparse (non-dense) index because it includes an entry for each disk block of the data file rather than for every search value.

The index file for a primary index needs fewer blocks than the data file, because there are lesser number of entries as compared to records of data. Also, each index entry is smaller in size as compared to a data record because it only has two fields. Therefore, more index entries can fit into a single block than data records. This implies that a binary search on the index file requires lesser number of blocks accesses as compared to a binary search on the data file.

Remember a binary search for an ordered data file requires $\log_2 b$ block accesses. If the primary index file contains b_i blocks, then to locate a record with a search key value, it requires a binary search of the index (b_i blocks), plus an access to the block containing the record: $\log_2 b_i + 1$ block accesses.

A record whose primary key value is K_i is in the block with the address P_i , where $K(i) \leq K(i+1)$. To retrieve a record, we do a binary search of the index file, to find the appropriate entry, then retrieve the data block whose address is $P(i)$.

Insertion and deletion of records are a problem with primary indexes, because they are ordered files. The problem increases with primary indexes because when we insert a record into the correct position in the data file, not only does space have to be made in the blocks for the new record, but index entries may also need to be changed because the anchor records of some blocks will change.

Problems can be overcome as seen previously using an unordered overflow file or a linked list of overflow records for each block in the data file.

13.2.4 Clustering Index

A clustering index occurs when records are physically ordered on a non-key field, which does not have a distinct value for each record. The field is called clustering

field. A clustering index can be created to speed the retrieval of records that have the same value for the clustering field.

Data on External Storage

Clustering indexes are different from primary indexes. This is because it is the requirement of a primary index that the ordering field of the data file should have a distinct value for each record.

An ordered file containing two fields is a clustering index. The first field is of the same type as the clustering field of the data file whereas the second fields is a block pointer. For each distinct value of the clustering field which contains the value and a pointer to the first block in the data file (containing a record with that value), there is one entry in the clustering index. For its clustering field, indexing fields with different values are stored in the same block.

NOTES

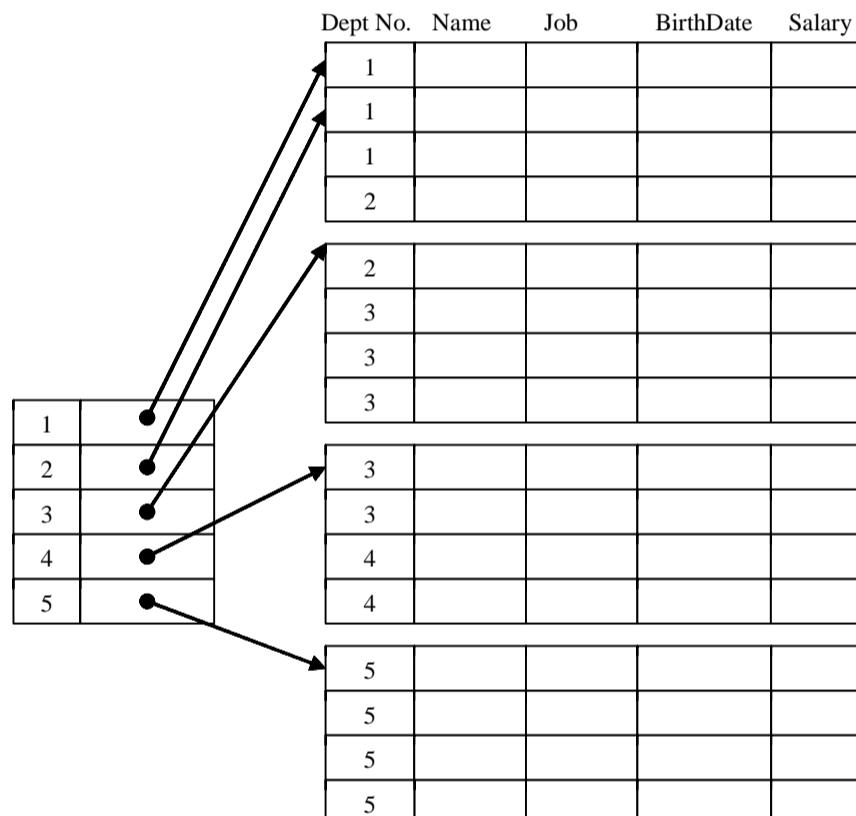
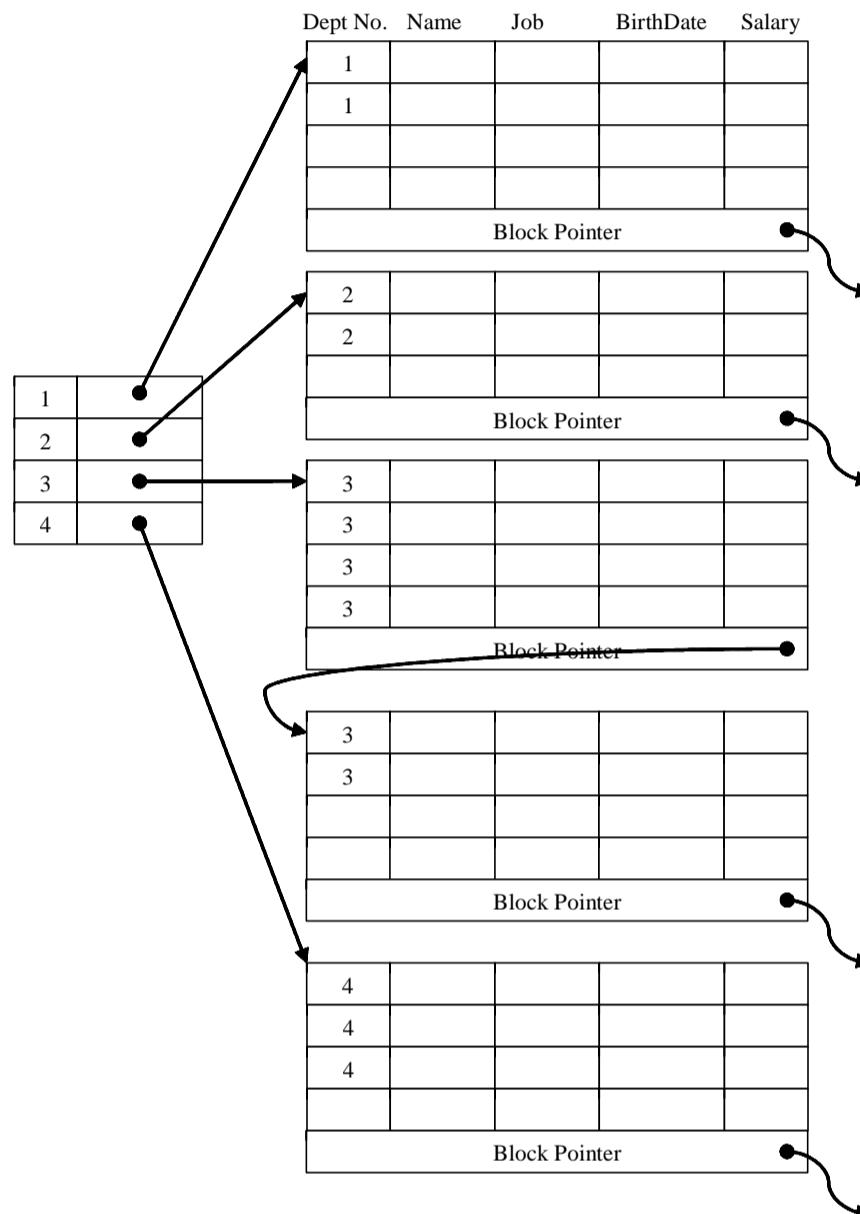


Fig. 13.2 Clustering Index

Insertion and deletion still cause problems because the records in the data file are still physically ordered. It is common to reserve a whole block or cluster of blocks for each value of the clustering field. All records with that value are placed in the block (or cluster). See Figure 13.3 for an example.

NOTES

**Fig. 13.3 Clustering with Each Distinct Key Value being Stored in a Separate Block**

A clustering index is a non-dense index, because there is an entry for every distinct value of the indexing field (which is non-key by definition) and has duplicate values, rather than for every record in the file.

An index is similar to the directory structures used for extendible hashing. Both are searched to find a pointer to the data block containing desired record. The main difference is that an index search uses the values of the search field itself, whereas a hash directory search uses the hash value that is calculated by applying the hash function to the search field.

13.2.5 Secondary Index

Data on External Storage

A secondary index offers a secondary means of accessing a file for which some primary access already exists. The secondary index may be on a field. This field is a candidate key with a unique value in each record, or a non-key with duplicate values.

An ordered file containing two fields is the index. The first field is of the same data type as the non-ordering field of the data file that is an indexing field. The second field is either a block pointer or a record pointer. Many secondary indexes can exist for the same file.

A secondary key is a secondary index access structure on a key field that has a distinct value for every record. There is one index entry for each record in the data file. This accounts for its density.

The index entries are ordered by value $K(i)$. Therefore, a binary search can be performed. Block anchors cannot be used as the records in the data file are not physically ordered by the values of the secondary key field. Instead of creating an index entry for each block, it is created for each record in the data file.

A block pointer is used to point to the block containing the record with the desired key value. The most suitable block is transferred to the main memory and a search can be performed for the desired record within the block.

A secondary index needs more storage space and longer search time than a primary index, because there are large numbers of entries. However, the improvement in the search time for a secondary index is greater than for a primary index since a linear search would have to be performed if the secondary index does not exist.

13.2.6 Non-Key Secondary Index

A secondary index can be created on a non-key field of a file. In this case, many records in the data file can have the same value for the indexing field. There are several ways an index like this can be created:

- Include several index entries with the same key value, one for each record. It would be a dense index.
- Have variable-length records for the index entries, with a repeating field for the pointer. A list of pointers can be kept in the index for the key, one to each block that contains a record whose indexing field is equal to the key value. Binary search can be used, but must be modified.
- Keep the length of the index entries fixed, and ensure the presence of a single entry for each index field value. However, a level of indirection to handle the numerous pointers must be created. This is a non-dense scheme.
 - o In the index entry, there is a pointer P which points to a block of record pointers.

NOTES

NOTES

- o Each record pointer in the block points to one of the data file records with the key value K for the indexing field.
- o If a value of K is seen in too many records and if the record pointers are unable fit in a single disk block, a linked list of blocks or a cluster is made use of.
- o Retrieval using the index requires one or more blocks accesses because of the extra level, but searching and inserting more records is straightforward.

A secondary index provides a logical ordering on the records by the indexing field.

13.2.7 Multi-Level Index

The indexing described so far involves ordered index files. A binary search is applied to the index to locate pointers to disk blocks or records in the file having a specific index field value.

Remember from before, a binary search requires $\log_2 b_i$ block accesses for an index with b_i blocks. Each step of the binary search reduces the part of the index file we search by a factor of 2 (each step divides the search space by 2, or halves the search space).

With multi-level indexes, the idea is to reduce the part of the index that we continue to search by a larger factor, the blocking factor of the index, where the bfr_i is greater than 2. The blocking factor of the index, bfr_i is called **fan-out**, or fo of the multi-level index.

Searching a multi-level index requires approximately $(\log_{fo} b_i)$ block accesses, which is a smaller number than for a binary search if the fan-out is larger than 2. If the fan out is equal to 2, there is no difference in the number of block accesses.

The index file is called the first level of a multi-level index. It is an ordered file with a distinct value for each key value $K(i)$. Therefore, we can create a **primary index** for the first level. This index to the first level is called second level of the multi-level index.

The second level is a primary index; therefore, we can use block anchors, and the second level has **one entry for each block** in the first level index. The blocking factor for all other levels is the same as for the first level, because the size of each index entry is the same. Each entry has one field value and one block address.

If the first level has r_1 entries and the blocking factor (which is also the fan out) for the index is $bfr_i = fo$, then the first level needs $\lceil r_1 / fo \rceil$ blocks, which is the number of entries r_2 needed at the second level of the index.

If necessary, this process can be repeated at the second level. The third level, which is an index for the second level, has an entry for each second level block, so the number of third level entries is $r_3 = \lceil r_2/f_0 \rceil$.

We only require a second level only if the first level needs more than one block of disk storage, and we require a third level only if the second level requires more than one block as well.

This process can be repeated until all the index entries at some level t fit in a single block.

The block at the t^{th} level is the top-level index. Each level reduces the number of entries from the previous level by a factor of f_0 (the index fan out or blocking factor of the index).

A multi-level index with r_1 first level entries will have approximately t levels, where:

$$t = \lceil \log_{f_0}(r_1) \rceil$$

The multi-level indexes can be used on any type of index—primary, clustering or secondary—as long as the first level index has distinct values for $K(i)$ and fixed-length entries.

NOTES

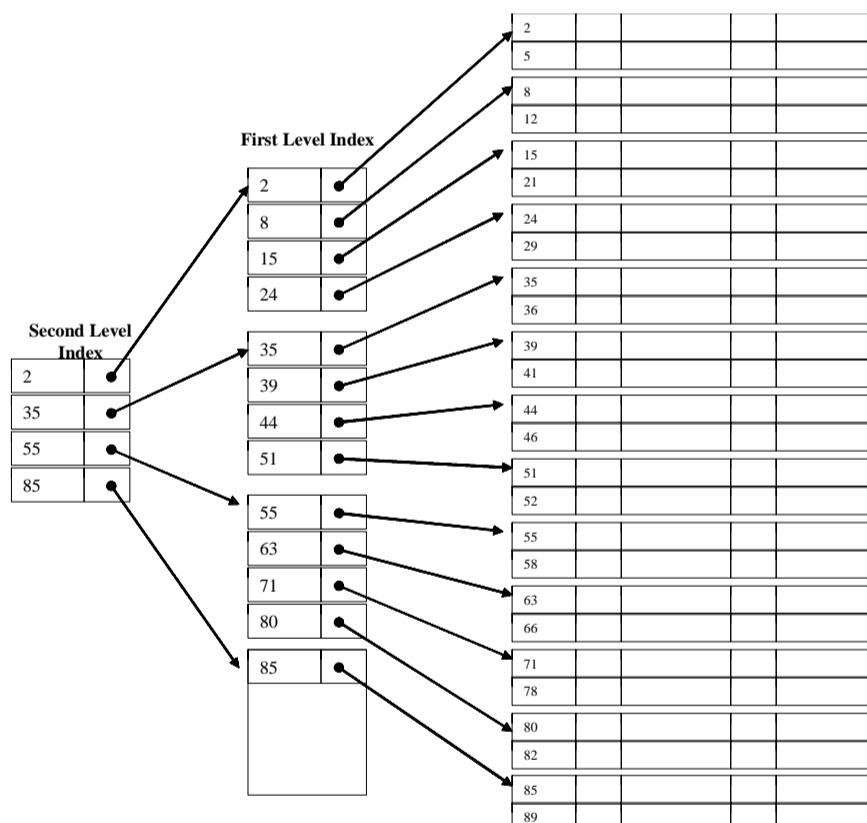


Fig. 13.4 A Two-Level Primary Index

NOTES**13.3 INDEX DATA STRUCTURES**

Indexing is used to enhance the performance of a database. This is done by minimizing the number of disk accesses which are required when a query is processed. The index is a type of data structure. It is used to access and locate the data in a database table quickly.

Indexes can be formed by using some database columns:

Search Key	Data Reference
-------------------	-----------------------

Fig. 13.5 Structure of Index

The first column of the database is the search key which consists a copy of the candidate key or primary key of the table. In primary key, the values are stored in sorted order so that the corresponding data can be retrieved easily.

The second column of the database is the data reference which consists of a set of pointers holding the address of the disk block, where the value of the particular key can be found.

13.3.1 Hash Based Indexing

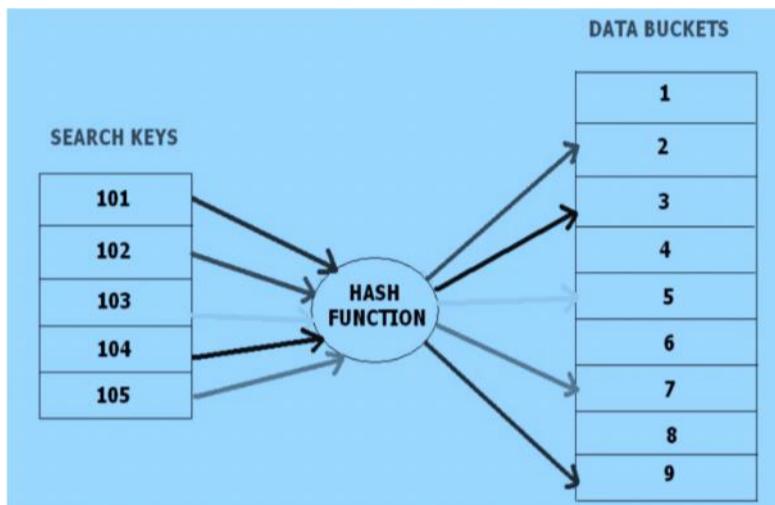
When we want to retrieve particular data in database system then it becomes very ineffective to search all the index values and reach the desired data. So the solution of this is Hashing. Hashing is a well-organized technique which is used to directly search the location of desired data on the disk and without using index structure. Data is stored at the data blocks whose address is produced by using hash function. The memory location where these records are stored is called as data bucket or data block.

Hash File Organization

- **Hash Function:** It is a mapping function which maps all the set of search keys to actual record address. Mostly, hash function uses primary key which is used to generate the hash index address of the data block. This function can be mathematical function to any complex mathematical function.
- **Data bucket:** These are the memory locations where the records are stored. These are also called *unit of storage*.
- **Hash Index:** Each hash index has a depth value which is used to indicate how many bits are used for calculating a hash function. The address of these bits is 2^n buckets.

The Figure below shows the functioning of hash function:

Data on External Storage



NOTES

Hashing is divided into two sub categories:

1. Static Hashing

In this hashing, the hash function always computes the same address when a search-key value is provided. Consider an example, if we want to create address for STUDENT_ID = 76 using mod (5) hash function, it always result in the same bucket address 4. The number of data buckets in the memory for this hashing remains constant all over because there will not be any changes to the bucket address.

Operations

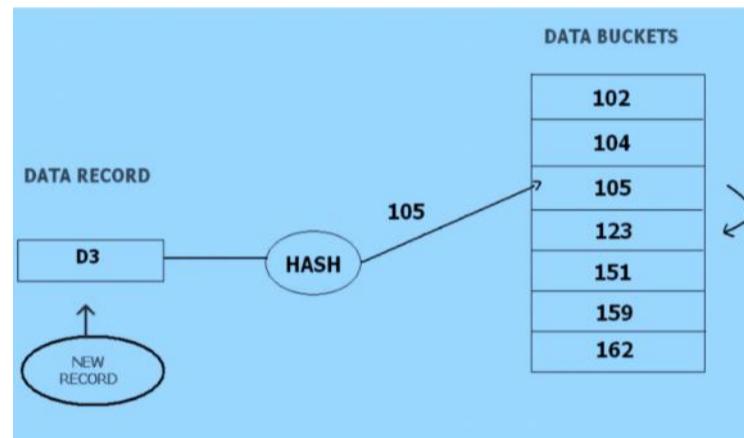
- **Insertion:** The hash function h generate a bucket address for the new record based on its hash key K , in case, When a new record is inserted into the table. The bucket address = $h(K)$.
- **Searching:** The hash function which is used to retrieve the bucket address for the record, in case, when a record needs to be searched. For Example, if we want to recover whole record for ID 76 and if on that ID, the hash function is mod (5), then the bucket address produced would be 4. Here ID acts as a hash key. This means they will directly got to address 4 and recover the whole record for ID 104.
- **Deletion:** Using the hash function, if we want to delete a record, we will first fetch the record which is assumed to be deleted. So we will remove the records for that address in memory.
- **Updation:** Byusing hash function, the data record that required to be updated is first searched and after that the data record is updated.

NOTES

Consider a situation, we want to insert some new records into the file and the data bucket address is generated by the hash function and is not empty or the data already exists in that address. This situation is called bucket overflow.

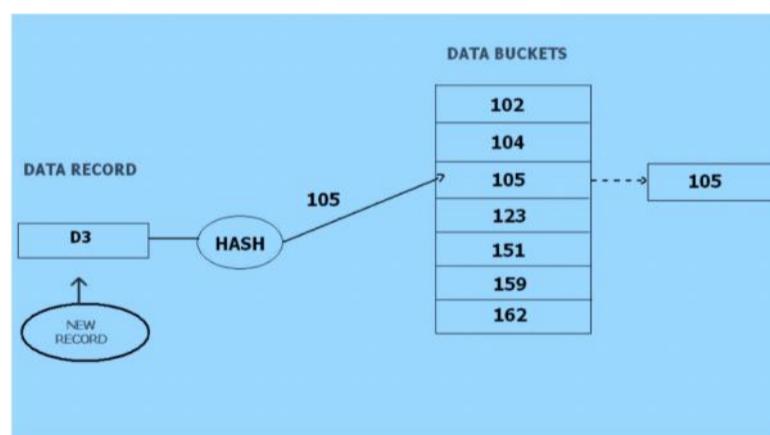
The solution of the above problem is given below:

- 1. Open Hashing:** In this method, instead of overwriting older one, the next available data block is used to enter the new record. This method is also called linear probing. For example, D3 is a new record which required to be inserted, the hash function produces address as 105. But it is full by this time. So the system finds the next available data bucket i.e. 123 and assigns D3 to it.



- 2. Closed Hashing:** In this hashing method, a new data bucket is assigned with same address and is linked it after the full data bucket. It is also called overflow chaining.

Consider an example to insert a new record D3 into the tables. The static hash function produces the data bucket address as 105. But this bucket is occupied to store the new data. So here, a new data bucket is added at the end of 105 data bucket and is linked to it. After that the new record D3 is inserted into the new bucket.



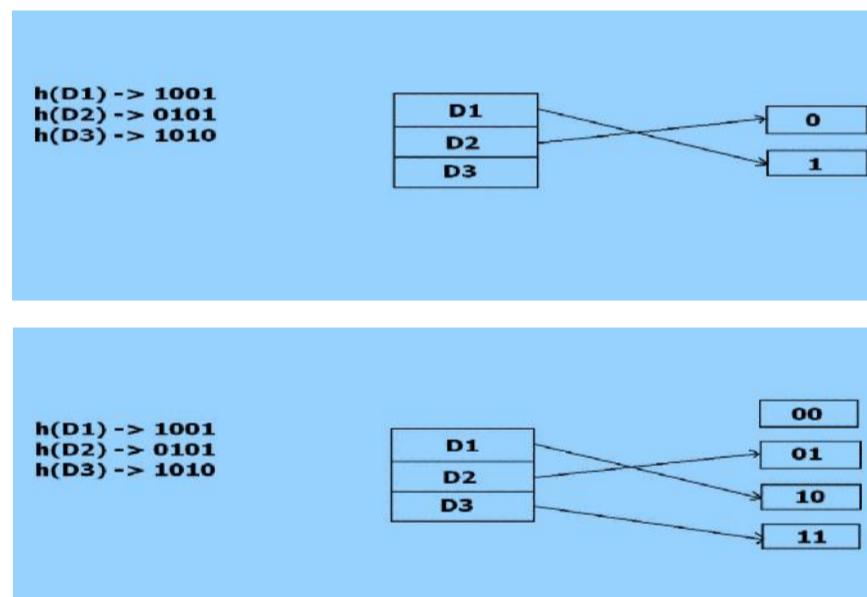
3. Quadratic Probing: This probing is very much alike to open hashing or linear probing. The difference between old and new bucket is linear. To find out the new bucket address, quadratic function is used.

4. Double Hashing: This type of hashing method is similar to linear probing. Here, the difference is fixed similar as in linear probing. But the calculations of this fixed difference is done by using another hash function.

2. Dynamic Hashing

The drawback of static hashing is that it does not shrink or expand dynamically, the reason is that the size of the database shrinks or grows. In dynamic hashing, as the records increases or decreases, the data buckets grows or shrinks. It is also known as extended hashing.

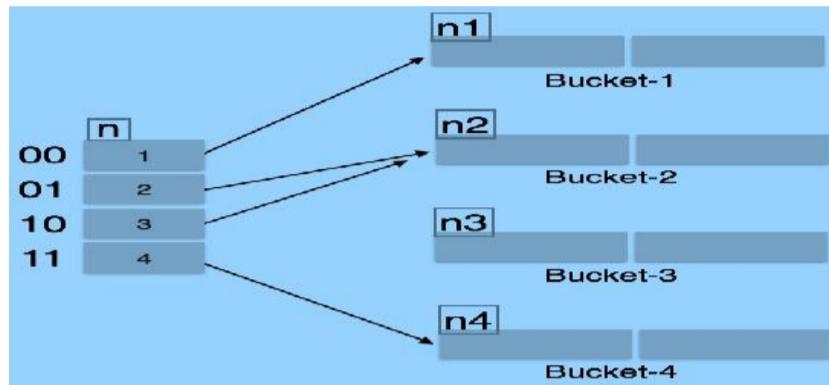
The hash function is prepared to produce a large number of values. For example, consider there are three data records D1, D2 and D3 and the hash function produces three addresses 1001, 0101 and 1010 respectively. This method of storing assumes only part of this address that is particularly only first one bit which is used to store the data. So this method attempts to load three of them at address 0 and 1 as shown below:



But here the problem is that, there is no bucket address is left for D3. The bucket has to grow vigorously to accommodate D3. So instead of 1 bit, it changes the address have 2 bits. Then it updates the existing data to have 2 bit address. Then it attempts to accommodate D3.

In dynamic hashing, hash function is made to yield a large number of values and are used limited initially.

NOTES

NOTES**Organization**

The prefix of the whole hash value is assumed as a hash index. For calculating bucket addresses, only a part of the hash value is used. To indicate how many bits are used for computing a hash function, every hash index has a depth value. The address of these bits can be 2^n buckets. When all the buckets are full, means when all these bits are consumed, then the depth value is increased linearly and twice the buckets are allocated.

Operation

- **Querying:** For querying, find out the depth value of the hash index and use the bits to calculate the bucket address.
- **Update:** update the data and implement a query.
- **Deletion:** To locate the desired data, implement a query and delete the same.
- **Insertion :** Calculate the address of the bucket
 - o If the bucket is already full then:
 - Add additional bits to the hash value.
 - Add more buckets.
 - Re-compute the hash function.
 - o Else
 - Add data to the bucket.
 - o Perform the remedies of static hashing, in case, if all the buckets are full.

When the data is organized in some ordering, then hashing is not constructive and then the queries require a range of data. Hash performs the best, when data is discrete and random.

13.3.2 Tree Based Indexing

We can use tree-like structures as index also. Consider an example of a binary search tree which can also be used as an index. To find out a specific record from

NOTES

a binary search tree, we used the additional advantage of binary search procedure which makes searching even faster. A binary tree has two pointers in each of its nodes, so it is also called as a **2-way Search Tree**. The number of value which are to be stored in each node is one less than the number of pointers.

B+ Tree is a balanced binary search trees. As we know that, as the database size grows, single level index records becomes large. It also degrades performance. B+ tree states that all leaf nodes persist at the same height and balanced. All leaf nodes are linked using link list to make B+ tree support random access and sequential access.

13.3.3 Comparison of File Organization

A database contains a huge amount of data. The data is grouped inside a table, and each table have linked records. A user can check that the data is stored in form of tables. But originally, there is a large amount of data is stored in physical memory in form of files.

File: A file is named collection of associated information which is recorded on secondary storage such as optical disks, magnetic disks and magnetic tapes.

File Organization: File Organization means the logical relationships among different records which consists of file that is the means of identification and can access to any specific record. We can say that, storing the files in certain order is called file Organization. **File Structure** means the format of data blocks and the label of any logical control record.

Types of File Organizations

There are different types of File Organizations i.e.

- Sequential File Organization
- Heap File Organization
- Hash File Organization
- B+ Tree File Organization
- Clustered File Organization

Where we have already discussed about hash file, B+ tree file. Some other file organization are:

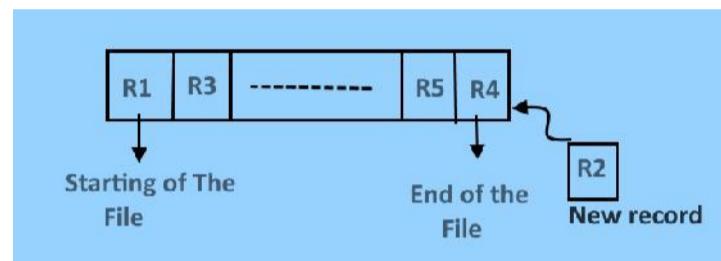
Sequential File Organization

This is the simplest one file organization system. In this method the file are arranged and stored one after another in a sequential manner. So to implement this method, there are two ways:

- (a) **Pile File Method:** In this method, we store the records in a sequence like one after other in the order and in which they are put in into the tables.

NOTES**Insertion of new record**

Consider there are some records like R1, R3 and so on upto R5 and R4 be fourth record in the sequence. The records are a row in any table. Assume that a new record R2 has to be inserted in the sequence, this is done by simply placing at the end of the file.



- (b) **Sorted File Method:** Whenever a new record has to be inserted, it is always be inserted in a sorted way. Based on any primary key or any other key, sorting of records can be done.

Insertion of new record: consider that there is a pre-existing sorted sequence of four records R1, R2, R3, and so on upto R7 and R8. Assume that a new record R2 has to be inserted in the sequence. So it will be inserted at the end of the file and after that it will sort the sequence.

**Advantages**

1. Efficient and fast method for huge amount of data.
2. Better design.
3. The files can be easily stored in magnetic tapes.

Disadvantages

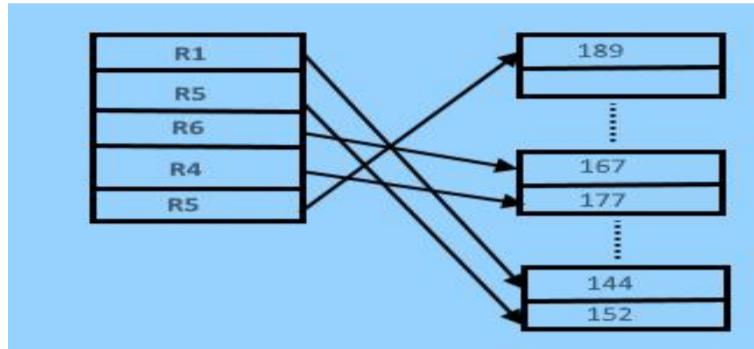
1. We cannot jump on a particular record that is required, instead we have to move in a sequential manner, this leads to time wastage.
2. Sorted file method takes time and space for sorting records, so it is inefficient method.

Heap File Organization

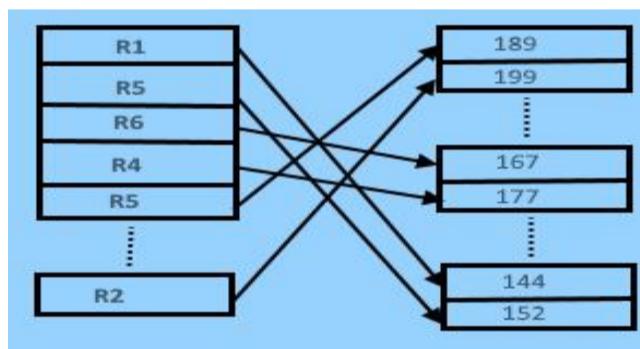
It is the responsibility of DBMS to store and manage the new records.

Heap File Organization deals with data blocks. In this method, the records are inserted at the end of the file, that is into the data blocks. The new record is

stored in some other block then no sorting or ordering is required. If a data block is full, then the other data block required not be the very next data block, instead it can be any block in the memory.

**NOTES**

Insertion of new record: Consider there are five records in the heap R1, R5, R6, R4 and R3. Suppose a new record R2 has to be inserted in the heap then, since the last data block, it means it is the data block 3 is full. It will be inserted in any of the database which is selected by the DBMS, assume that this data block 1.



To delete, search or update data in heap file organization, it means, it will traverse the data from the beginning of the file until we will get the requested record. Because of the database is very large, so searching, deleting or updating the record will take a lot of time.

Advantages

1. Retrieving and fetching records is faster than sequential record, this is the case of small databases.
2. In case of a huge number of data that is required to be loaded into the database at a time, so this method is best suitable.

Disadvantages

1. There is a problem of unused memory blocks.
2. Incompetent for larger databases.

NOTES

Check Your Progress

1. What is indexing?
2. What are the different types of indexes?
3. Define hashing.

13.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Indexing is a data structure-based technique for accessing records in a file. Indexes are auxiliary access structures, which are used to speed up the retrieval of records in response to certain search conditions.
2. There are different types of indexes; these include:
 - Primary index
 - Clustering index
 - Secondary index
3. Hashing is a well-organized technique which is used to directly search the location of desired data on the disk and without using index structure.

13.5 SUMMARY

- Hashing is a computational technique for organizing files. Records in hashed files can be stored and retrieved quickly. However, the difficulty in hashed files is that they are difficult to process in key order, which is important if you want to access all records with keys in a certain range.
- Indexing is a data structure-based technique for accessing records in a file. Indexes are auxiliary access structures, which are used to speed up the retrieval of records in response to certain search conditions.
- For a file consisting of several fields, and index access structure is defined on a single field which is referred to as indexing field. An index is usually composed of two components, the index field value as well as a list of pointers to all disk blocks containing records with that field value.
- A primary index is an ordered file whose records are of fixed length with two fields. The first field is the primary key of the main data file, and the second is a pointer to a disk block.
- A clustering index occurs when records are physically ordered on a non-key field, which does not have a distinct value for each record.

NOTES

- A secondary index offers a secondary means of accessing a file for which some Data on External Storage primary access already exists.
- A secondary index can be created on a non-key field of a file. In this case, many records in the data file can have the same value for the indexing field.
- The drawback of static hashing is that it does not shrink or expand dynamically, the reason is that the size of the database shrinks or grows. In dynamic hashing, as the records increases or decreases, the data buckets grows or shrinks.

13.6 KEY WORDS

- **Database Index:** It is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure.
- **Hashing:** It is a technique to calculate the direct location of a data record on the disk without using index structure.

13.7 SELF ASSESSMENT QUESTIONS AND EXERCISES**Short Answer Questions**

1. Discuss the significance of indexing.
2. Write a note on:
 - (a) Primary index
 - (b) Cluster index
 - (c) Secondary index
3. Compare the different types of file organizations.

Long Answer Questions

1. Explain the single level ordered index.
2. What is non-key secondary index? Explain.
3. Explain the hash-based and tree-based indexing in detail.

13.8 FURTHER READINGS

Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.

NOTES

- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.

UNIT 14 PERFORMANCE TUNING

Structure

- 14.0 Introduction
- 14.1 Objectives
- 14.2 Intuitions for Tree Indexes
- 14.3 B⁺ Tree
 - 14.3.1 Structure of B⁺ Tree
 - 14.3.2 B⁺ Trees: A Dynamic Index Structure
- 14.4 Indexed Sequential Access Method (ISAM)
- 14.5 Answers to Check Your Progress Questions
- 14.6 Summary
- 14.7 Key Words
- 14.8 Self Assessment Questions and Exercises
- 14.9 Further Readings

NOTES

14.0 INTRODUCTION

The objective of database tuning is to maximize use of system resources in order to perform work as rapidly and efficiently as possible. Database tuning defines a group of activities which is used to homogenize and optimize the performance of a database.

Performance tuning is the process of upgrading of system performance. The motivation for such activity, in computer system is called a performance problem which can be either real or unexpected. But in computer system, most of the systems answers to increased load having some degree of decreasing performance.

The steps involved to improve the database performance are:

- The use of non-correlated scalar sub query should be avoided.
- Multiple joins in a single query should be avoided.
- There is a requirement to improve SQL query performance.
- Eliminate cursors from the query.
- Some points should be kept in mind while creating and using indexes.
- Multi-statement table valued functions (TVFs) should be avoided.
- Points should be keep in mind while creating a highly selective index.

In this unit, you will study about the performance tuning, intuition for tree indexes, Indexed Sequential Access Method (ISAM), B⁺ trees and dynamic structure.

NOTES**14.1 OBJECTIVES**

After going through this unit, you will be able to:

- Define performance tuning
- Understand the benefits of B+ tree in indexing
- Explain the indexed sequential access method

14.2 INTUITIONS FOR TREE INDEXES

Data is stored in the form of records. Each record has a key field. This key field is used to require to help so that each data to be recognized uniquely.

Indexing is a data structure method which is used to completely retrieve records from the database files and these are based on some attributes and on these the indexing has been done.

Indexing is based on its indexing attributes. There are different types of indexing:

- **Primary Index:** It is defined on an ordered data file and that data file is ordered based on a key field. The key field is usually the primary key of the relation.
- **Secondary Index:** Secondary index has been created from a field and that field is a candidate key. It has a non-key with duplicate values or unique value in every record.
- **Clustering Index:** It is defined on an ordered data file and that data file is ordered on a non-key field.

Primary index is divided into two categories i.e. sparse and dense index.

Sparse Index

In sparse index, an index record consists of an actual pointer to the data on the disk and a search key. The index records are not created for every search key. In order to search a record, the first step is continue by index record. The second step is to reach at the particular location of the data. The system starts sequential search until the desired data is found, in case if the data we are looking for is not there, we directly reach by following the index.



Dense Index

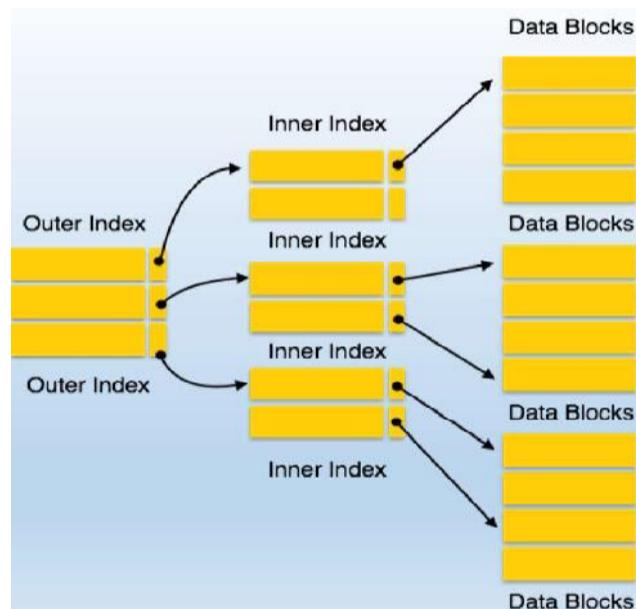
Performance Tuning

In dense index, there is an index record for every search key value in the database to make searching faster. But, it needs more space to accumulate index records itself. Index records consists of a pointer to the actual record on the disk and search key value.

China	→	China	Beijing	3,705,386
Canada	→	Canada	Ottawa	3,855,081
Russia	→	Russia	Moscow	6,592,735
USA	→	USA	Washington	3,718,691

Multilevel Index

Multilevel index is kept on the disk beside with the actual database files. As we know that, index records consists of data pointers and search-key values. The size of indices depends on the size of database. Size of the indices increases as the size of the database grows. To speed up the search operations, there is huge requirement to keep the index records in the main memory. If we are using a single-level index then a large size index cannot be kept in memory because it leads to multiple disk accesses.



To make the outermost level so small, the multi-level index helps in breaking down the index into a number of smaller indices in order. So that it can be saved in a single disk block. These smaller indices can be easily put up anywhere in the main memory.

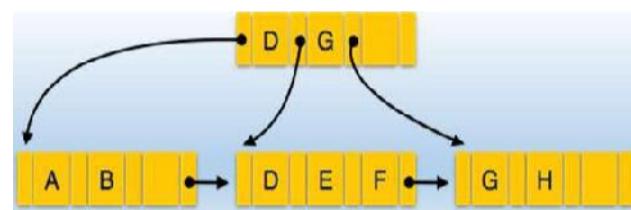
NOTES

NOTES**14.3 B⁺ TREE**

A B⁺ tree is a balanced binary search tree which tracks a multi-level index format. B⁺ tree guarantees that all leaf nodes persist at the same height and so it is balanced. The leaf nodes of a B⁺ tree represents the actual data pointers. A B⁺ tree can maintain random access as well as sequential access because the leaf nodes are connected using a link list.

14.3.1 Structure of B⁺ Tree

In B⁺ tree, the leaf node is at equal distance from the root node. So here, B⁺ tree is of the order N which is fixed for every B⁺ tree.

**Internal nodes**

- An internal node can contain at most N pointers.
- Non-leaf nodes or internal nodes contains at least $\lceil n/2 \rceil$ pointers excluding the root node.

Leaf nodes

- Each leaf node consists of one block pointer which points to next leaf node and makes a linked list.
- A leaf node can consists of at most N record pointers and N key values.
- Leaf nodes contain at least $\lceil n/2 \rceil$ key values and $\lceil n/2 \rceil$ record pointers.

B⁺ Tree Insertion

Insertion in a B⁺ Tree involves the following steps:

1. In B⁺ trees, and each entry is done at the leaf node and these are filled from bottom.
2. If a leaf node overflows then:
 - Divided node into two parts.
 - Divide at $i = \lceil (m + 1)/2 \rceil$.
 - Here, the first i entries are kept in one node.
 - Remaining of the entries that is $i+1$ onwards are encouraged to a new node.
 - i^{th} key is replicated at the parent of the leaf.

3. If a non-leaf node overflows then:

Performance Tuning

- Divide the node into two parts.
- Divide the node at $i = \lceil (m + 1)/2 \rceil$
- Entries up to i are retained in one node.
- Remaining of the entries are moved to a new node.

NOTES

B⁺ Tree Deletion

Deletion in B⁺ tree involves the following steps:

1. B⁺ tree entries which are deleted at the leaf nodes.
2. The marked entry is searched and then finally deleted.
 - o Delete and replace with the entry from the left position, if it is an internal node.
3. After the above steps of deletion, now underflow is tested.
 - o Allocate the entries from the nodes left to it, if underflow occurs.
 - o But if allocation is not possible from left, then allocate from the nodes right to it.
 - o Merge the node with left and right to it, if allocation is not possible from left or from right.

14.3.2 B⁺ Trees: A Dynamic Index Structure

B-tree and B⁺ tree are generally employed to implement multilevel indexing. As we know that the limitation of B-tree used for indexing. B-tree equivalent to a particular key value stores the data pointer along with key value in the node of a B-tree. But this method, significantly decreases the number of entries which can be packed into a node of a B-tree. By increasing the search time of a record, there is an increase in the number of levels in the B-tree.

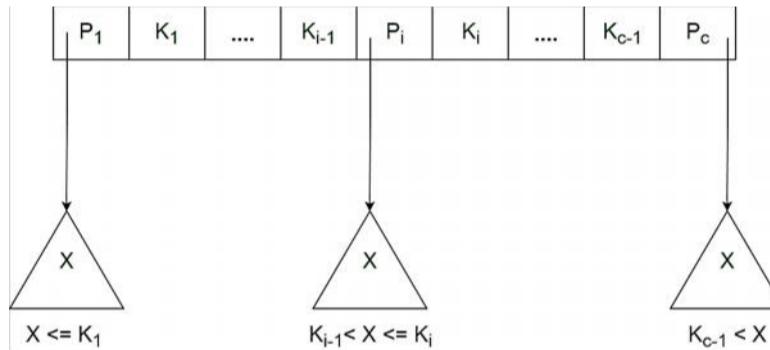
But as there are some drawbacks in B-trees, B⁺ tree removes all these above drawbacks. It is used to remove these limitations by storing data pointers only at the leaf nodes of the tree. So, the structure of internal nodes of the B⁺ tree is quite different from the structure of leaf nodes of a B⁺ tree. In order to access the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, so that data pointers are present only at the leaf nodes. But in this case, the leaf nodes are connected to give ordered access to the records. Therefore, the leaf nodes form the first level of index, and this can be done with the help of internal nodes which are forming the other levels of a multilevel index. To just act as a medium to control the searching of a record, some of the key values of the leaf nodes also seems in the internal nodes.

NOTES

So, it seems that a B⁺ tree, unlike a B-tree has two orders, ‘a’ and ‘b’. Here ‘a’ and ‘b’ represents, the first one for the internal nodes and the other one for the external (or leaf) nodes.

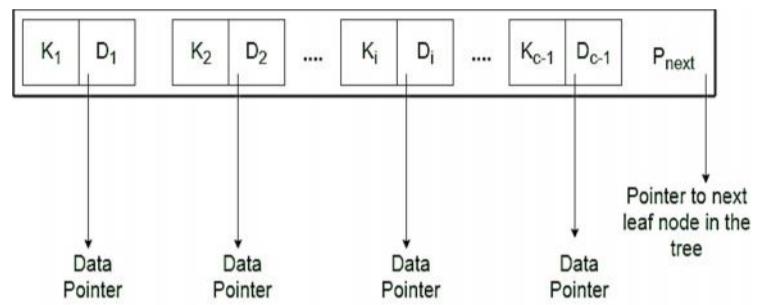
The structure of the internal nodes of a B⁺ tree of order ‘a’ is as follows:

- Each internal node which can be in the form :
 $\langle P_1, K_1, P_2, K_2, \dots, P_{c-1}, K_{c-1}, P_c \rangle$
 where each P_i is a tree pointer , each K_i is a key value and the condition is that : $c \leq a$.
- Each internal node can be represented as : $K_1 < K_2 < \dots < K_{c-1}$
- In the sub-tree pointed at by P_i for every search field values ‘X’, the n the following condition must holds :
 - $K_{i-1} < X \leq K_i$, for $1 < i < c$ and
 - $K_{i-1} < X$, for $i = c$
- Every internal nodes which can have at most ‘a’ tree pointers.
 - o If any internal node has ‘c’ pointers, then it has ‘c – 1’ key values that is, $c \leq a$.
 - o The other internal nodes have at least $\lceil a/2 \rceil$ tree pointers where as the root node has at least two tree pointers.

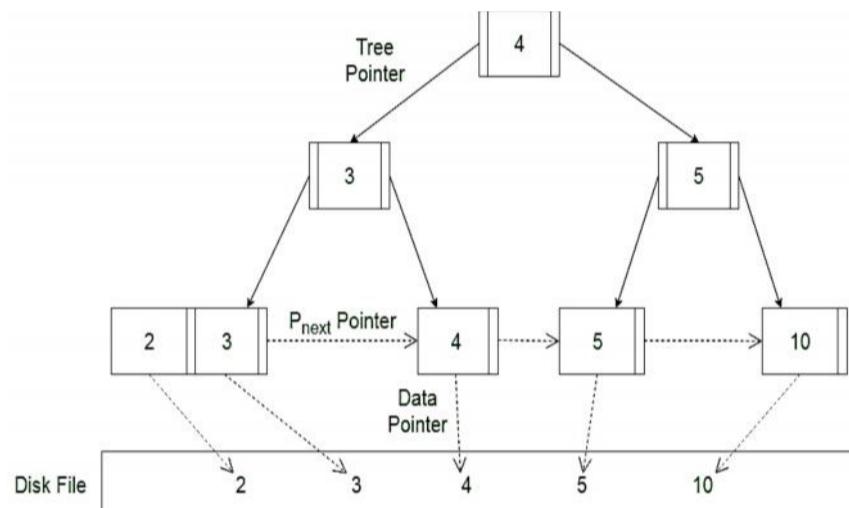


The arrangement of the leaf nodes of a B⁺ tree which are of the order ‘b’ can be represented as:

- Each leaf node can be represented in the form as shown below:
 $\langle \langle K_1, D_1 \rangle, \langle K_2, D_2 \rangle, \dots, \langle K_{c-1}, D_{c-1} \rangle, P_{\text{next}} \rangle$
 where each D_i is a data pointer , P_{next} points to next leaf node in the B⁺ tree , each K_i is a key value and the condition is that: $c \leq b$.
- Each leaf node can be represented as: $c \leq b$, $K_1 < K_2 < \dots < K_{c-1}$.
- Every leaf node can have the values that is at least $(\lceil b/2 \rceil)$.
- All leaf nodes are at identical level.



By using the P_{next} pointer, it is feasible to traverse all the leaf nodes.

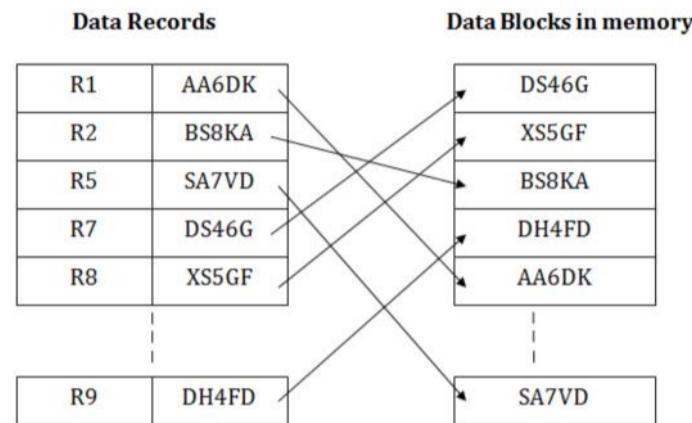


NOTES

A B⁺ tree with 'l' level can store more entries in its internal nodes than B-tree which emphasises the substantial improvement made to the search time for any given key. Presence of P_{next} pointers and taking lesser levels makes B⁺ tree very efficient and quick in accessing records from disks.

14.4 INDEXED SEQUENTIAL ACCESS METHOD (ISAM)

Indexed Sequential Access Method (ISAM) method is an innovative sequential file organization. In this method, an index value is created for each primary key and mapped with the record. This index consists of the address of the record in the file. The records are stored in the file using the primary key.

NOTES

If any record has to be retrieved based on its index value then the address of the data block is fetched and the record is retrieved from the memory.

Advantages of ISAM

- ISAM method supports partial retrieval and range retrieval of records. We can retrieve the data for the given range of value because the index is based on the primary key value. Similarly, the partial value can also be simply examined, that is, the student name starting with 'AM' can be easily searched.
- In ISAM method, searching a record in a large database is fast and easy where each record has the address of its data block.

Disadvantages of ISAM

- The ISAM files have to be reconstructed to maintain the sequence when the new records are inserted.
- In ISAM files, the space used by it needs to be released when the record is deleted. Otherwise the performance of the database can be slow down.
- This method needs extra space in the disk to stored the index value.

Check Your Progress

1. What are the two categories of primary index?
2. Define B⁺ tree.

14.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Primary index is divided into two categories i.e. sparse and dense index.
2. A B⁺ tree is a balanced binary search tree which tracks a multi-level index format.

14.6 SUMMARY

- Performance tuning is the process of upgrading of system performance.
- Indexing is a data structure method which is used to completely retrieve records from the database files and these are based on some attributes and on these the indexing has been done.
- Primary index is divided into two categories i.e. sparse and dense index.
- In sparse index, an index record consists of an actual pointer to the data on the disk and a search key.
- In dense index, there is an index record for every search key value in the database to make searching faster.
- A B⁺ tree is a balanced binary search tree which tracks a multi-level index format. B⁺ tree guarantees that all leaf nodes persist at the same height and so it is balanced.
- Indexed Sequential Access Method (ISAM) method is an innovative sequential file organization. In this method, an index value is created for each primary key and mapped with the record.

NOTES

14.7 KEY WORDS

- **B⁺ Tree:** It is an N-ary tree with a variable but often large number of children per node.
- **ISAM:** It is a method for creating, maintaining, and manipulating indexes of key-fields extracted from random data file records to achieve fast retrieval of required file records.

14.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What do you understand by performance tuning?
2. Discuss the different type of indexing.

Long Answer Questions

1. Explain the following:
 - (i) Structure of B⁺ tree
 - (ii) Insertion and deletion in B⁺ tree
2. What is indexed sequential access method (ISAM)? What are the advantages and disadvantages of ISAM?

NOTES

14.9 FURTHER READINGS

- Ramakrishnan, Raghu and Johannes Gehrke. 2003. *Database Management Systems*. New Delhi: McGraw-Hill Education.
- Silberschatz, Abraham, Henry Korth and S. Sudarshan. 2010. *Database System Concepts*, 6th Edition. New York: McGraw-Hill.
- Elmasri, Ramez and Shamkant B. Navathe. 2006. *Fundamentals of Database Systems*, 5th Edition. Boston: Addison-Wesley.
- Ritchie, Colin. 2004. *Relational Database Principles*, 2nd Edition. New Delhi: Cengage Learning India Pvt. Ltd.
- Maheshwari, Sharad and Ruchin Jain. 2006. *Database Management Systems Complete Practical Approach*. New Delhi: Firewall Media (Imprint of Laxmi Publications (P) Ltd.
- Coronel, Carlos M and Peter Rob. 2006. *Database Systems: Design, Implementation, and Management*, 7th Edition. US: Cengage Learning.
- Date, C. J. 2003. *An Introduction to Database Systems*, 8th Edition. Boston: Addison-Wesley.
- Leon, Alexis and Mathews Leon. 2008. *Database Management Systems*, 1st Edition. New Delhi: Vikas Publishing House Pvt. Ltd..
- Vaswani, Vikram. 2003. *MySQL: The Complete Reference*, 1st Edition. New York: McGraw Hill Professional.
- Murach, Joel. 2012. *Murach's MySQL*. California: Mike Murach & Associates.
- DuBois, Paul. 2007. *MySQL Cookbook*, 2nd Edition. California: O'Reilly Media.