ALAGAPPA UNIVERSITY

(Accredited with 'A+' Grade by NAAC (with CGPA: 3.64) in the Third Cycle and Graded as category - I University by MHRD-UGC)
(A State University Established by the Government of Tamilnadu)

KARAIKUDI – 630 003

DIRECTORATE OF DISTANCE EDUCATION

Bachelor of Computer Applications

Second Year – Third Semester

10133

RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS)

Copy Right Reserved For Private Use only

Author:
Dr. C.Balakrishnan
Assistant Professor
Alagappa Institute of Skill Development
Alagappa University,
Karaikudi. 630 003.

"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.

Reviewer
Dr. P. Prabhu,
Assistant Professor of Computer Application,
Directorate of Distance Education,
Alagappa University,
Karaikudi. 630 003.

RELATIONAL DATABASE MANAGEMENT SYSTEM
(RDBMS)
SYLLABI-BOOK MAPPING TABLE

	Syllabi	Mapping in Book
	BLOCK 1 : INTRODUCTION	
UNIT I	**Data base System Applications,** data base System VS file System – View of Data – Data Abstraction –Instances and Schemas – data Models – the ER Model	**1-11**
UNIT II	**Model :** Relational Model – Other Models – Database Languages – DDL – DML – database Access for applications Programs – data base Users and Administrator – Transaction Management – data base System Structure – Storage Manager – the Query Processor	**12-22**
UNIT III	**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.	**23-33**
	BLOCK 2 : RELATIONAL MODEL	
UNIT IV	**Introduction–** Integrity Constraint Over relations – Enforcing Integrity constraints – Querying relational data – Logical data base Design – Introduction to Views – Destroying / altering Tables and Views.	**34-48**
UNIT V	**Relational Algebra** – Selection and projection set operations – renaming – Joins – Division – Examples of Algebra overviews	**49-59**
UNIT VI	**Relational calculus** – Tuple relational Calculus – Domain relational calculus – Expressive Power of Algebra and calculus.	**60-65**
	BLOCK 3 : SQL QUERY	
UNIT VII	**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 Data bases. Schema refinement	**66-88**
UNIT VIII	**Normal forms :** Problems Caused by redundancy – Decompositions – Problem related to decomposition – reasoning about FDS – FIRST, SECOND, THIRD Normal forms – BCNF	**89-108**
UNIT IX	**Join:** Lossless join Decomposition – Dependency preserving Decomposition – Schema refinement in Data base Design – Multi valued Dependencies – FORTH Normal Form.	**109-116**

BLOCK 4: TRANSACTION

UNIT X	**Introduction :**Transaction Concept- Transaction State- Implementation of Atomicity and Durability – Concurrent – Executions – Serializability- Recoverability – Implementation of Isolation – Testing for serializability	**117-128**
UNIT XI	**Protocols :** Lock Based Protocols – Timestamp Based Protocols- Validation- Based Protocols – Multiple Granularity.	**129-138**
UNIT XII	**Recovery and Atomicity** – Log – Based Recovery – Recovery with Concurrent Transactions – Buffer Management – Failure with loss of nonvolatile storage- Advance Recovery systems- Remote Backup systems	**139-154**
	BLOCK 5 : STORAGE	
UNIT XII	**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	**155-165**
UNIT IV	**Performance Tuning**- Intuitions for tree Indexes – Indexed Sequential Access Methods (ISAM) – B+ Trees: A Dynamic Index Structure.	**166-174**

CONTENTS

BLOCK I INTRODUCTION

UNIT I DATABASE SYSTEM APPLICATIONS 1-11

1.1 Introduction
1.2 Objectives
1.3 Introduction to Database Systems
 1.3.1 Database Management System (DBMS)
 1.3.2 Applications of Database Management System (DBMS)
1.4 Database System Vs file System
 1.4.1 Problems with File System
 1.4.2 Advantages of Database System
1.5 View of Data
1.6 Data Abstraction
1.7 Instances and Schemas
1.8 Data Models
1.9 The ER Model
1.10 Answers to Check Your Progress Questions
1.11 Summary
1.12 Key Words
1.13 Self-Assessment Questions and Exercises
1.14 Further Readings

UNIT – II MODEL 12-22

2.1 Introduction
2.2 Objectives
2.3 Relational Model
2.4 Database Languages
 2.4.1 DDL
 2.4.2 DML
2.5 Database Access for applications Programs
2.6 Data base Users and Administrator
2.7 Transaction Management
2.8 Data base System Structure
2.9 Storage Manager
2.10 The Query Processor
2.11 Answers to Check Your Progress Questions
2.12 Summary
2.13 Key Words
2.14 Self-Assessment Questions and Exercises
2.15 Further Readings

UNIT – III HISTORY OF DATABASE SYSTEMS 23-33

3.1 Introduction
3.2 Objectives
3.3 Database design and ER diagrams
3.4 Beyond ER Design Entities
3.5 Attributes and Entity sets
3.6 Relationships and Relationship sets
3.7 Additional features of ER Model
3.8 Concept Design with the ER Model
3.9 Conceptual Design for Large enterprises
3.10 Answers to Check Your Progress Questions
3.11 Summary
3.12 Key Words
3.13 Self-Assessment Questions and Exercises
3.14 Further Readings

UNIT IV INTRODUCTINO TO RELATIONAL MODEL 34-48

4.1 Introduction
4.2 Objectives
4.3 Structure of Relational Model
4.4 Integrity Constraint over Relations
4.5 Enforcing Integrity constraints
4.6 Querying relational data
4.7 Logical Database Design
4.8 Introduction to Views
4.9 Destroying / altering Tables and Views
4.10 Answers to Check Your Progress Questions
4.11 Summary

UNIT V RELATONAL ALGEBRA 49-59

5.1 Introduction
5.2 Objectives
5.3 Introduction to Relational Algebra
5.4 Selection and projection set operations
5.5 Renaming
5.6 Joins
5.7 Division
5.8 Examples of Algebra overviews
5.9 Answers to Check Your Progress Questions
5.10 Summary
5.11Key Words
5.12 Self-Assessment Questions and Exercises
5.13 Further Readings

UNIT VI RELATIONAL CALCULUS 60-65
6.1 Introduction
6.2 Objectives
6.3 Relational Calculus

6.4 Tuple Relational Calculus
6.5 Domain Relational Calculus
6.6 Expressive Power of Algebra and Calculus
6.7 Answers to Check Your Progress Questions
6.8 Summary
6.9 Key Words
6.10 Self-Assessment Questions and Exercises
6.11 Further Readings

UNIT VII FORMS OF BASIC SQL QUERY 66-88

7.1 Introduction
7.2 Objectives
7.3 Introduction to SQL Queries
7.4 Examples of Basic SQL Queries
7.5 Introduction to Nested Queries
7.6 Correlated Nested Queries Set
7.7 Comparison Operators
7.8 Aggregative Operators
7.9 NULL Values
7.10Comparision using NULL Values
7.11 Logical connectivity's AND, OR and NOT
7.12 Outer Join
7.13 Disallowing NULL Values
7.14 PL/SQL
7.15 Complex Integrity Constraints in SQL Triggers and Active Databases
7.16 Answers to Check Your Progress Questions
7.17 Summary
7.18 Key Words
7.19 Self-Assessment Questions and Exercises
7.20 Further Readings

UNIT VIII NORMAL FORMS 89-108

8.1 Introduction
8.2 Objectives
8.3 Problems caused by redundancy
8.4 Decompositions
8.5 Problem related to decomposition
8.6 Reasoning about FDS
8.7 FIRST, SECOND, THIRD Normal Forms
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 – IX JOINS **109-116**

9.1 Introduction
9.2 Objectives
9.3 Lossless Join Decomposition
9.4 Dependency preserving Decomposition
9.5 Schema refinement in Database Design
9.6 Multi valued Dependencies
9.7 FORTH Normal Form
9.8 Answers to Check Your Progress Questions
9.9 Summary
9.10 Key Words
9.11 Self-Assessment Questions and Exercises
9.12 Further Readings

UNIT X INTRODUCTION **117-128**

10.1 Introduction
10.2 Objectives
10.3Transaction Concept
10.4 Transaction State
10.5 Implementation of Atomicity and Durability
10.6 Concurrent
10.7 Executions
10.8 Serializability
10.9 Recoverability
10.10 Implementation of Isolation
10.11 Testing for serializability
10.12 Answers to Check Your Progress Questions
10.13 Summary
10.14 Key Words
10.15 Self-Assessment Questions and Exercises
10.16 Further Readings

UNIT XI PROTOCOLS **129-138**

11.1 Introduction
11.2 Objectives
11.3 Lock Based Protocols
11.4 Timestamp Based Protocols
11.5Validation
11.6 Multiple Granularity
11.7 Answers to Check Your Progress Questions
11.8 Summary
11.9 Key Words
11.10 Self-Assessment Questions and Exercises
11.11 Further Readings

UNIT XII RECOVERY AND ATOMICITY **139-154**

12.1 Introduction
12.2 Objectives
12.3 Log
12.4 Based Recovery
12.5 Recovery with Concurrent Transactions
12.6 Buffer Management
12.7 Failure with loss of non-volatile storage
12.8Advance Recovery systems
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

UNIT XIII DATA ON EXTERNAL STORAGE **155-165**

13.1 Introduction
13.2 Objectives
13.3 File Organization and Indexing
13.4 Cluster Indexes, Primary and Secondary Indexes
13.5 Index data Structures
13.6 Hash Based Indexing
13.7 Tree base Indexing
13.8 Comparison of File Organizations
13.9 Indexes
13.10 Answers to Check Your Progress Questions
13.11 Summary
13.12 Key Words
13.13 Self-Assessment Questions and Exercises
13.14 Further Readings

UNIT XIV PERFORMANCE TUNING **166-174**

14.1 Introduction
14.2 Objectives
14.3 Intuitions for tree Indexes
14.4 Indexed Sequential Access Methods (ISAM)
14.5 A Dynamic Index Structure
14.6 Answers to Check Your Progress Questions
14.7 Summary
14.8 Key Words
14.9 Self-Assessment Questions and Exercises
14.10 Further Readings

Model Question Paper **175**

BLOCK – I

INTRODUCTION

UNIT I DATABASE SYSTEM APPLICATIONS

Structure

1.1 Introduction
1.2 Objectives
1.3 Introduction to Database Systems
  1.3.1 Database Management System (DBMS)
  1.3.2 Applications of Database Management System (DBMS)
1.4 Database System Vs file System
  1.4.1 Problems with File System
  1.4.2 Advantages of Database System
1.5 View of Data
1.6 Data Abstraction
1.7 Instances and Schemas
1.8 Data Models
1.9 The ER Model
1.10 Answers to Check Your Progress Questions
1.11 Summary
1.12 Key Words
1.13 Self-Assessment Questions and Exercises
1.14 Further Readings

1.1 Introduction

Database is a logical collection of inter-related data relevant to an enterprise. The Database Management System (DBMS) is a collection of interrelated data and a set of programs to access those data. The DBMS is designed to manage and maintain large amount of data in database. DBMS are widely used in all application areas, where data are involved.

1.2 Objectives

This section gives an introduction about the:

- Basics of Database systems
- Fundamental concepts of DBMS
- Applications of DBMS
- Differences between File system and Database system
- Various Data models

1.3 Introduction to Database Systems

This section gives definitions of some preliminary concepts of RDBMS.

- **Data -** raw facts that can be recorded and that have implicit meaning. For example, consider the names, register numbers, courses (subjects) and marks secured for each course
- **Datum** - singular form of Data or unit of Data
- **Information –** processed data is called information. Actually data are processes / interpreted to get some semantics
- **Database** – logical collection of inter-related data relevant to an enterprise
- **Database Management System** (DBMS) is a collection of interrelated data and a set of programs to access those data

1.3.1 Database Management System (DBMS)

The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient. The DBMS is important because without the existence of some kind of rules and regulations it is not possible to maintain the database.

It may be required to select the particular attributes of a particular table; the common attributes to create relationship between two tables; if a new record has to be inserted or deleted then which tables should have to be handled etc. These issues must be addressed by having some kind of rules to follow in order to maintain the integrity of the database.

DBMS is designed to manage large collection of information. Managing the data involves both defining structures for storage of information and providing methodologies for the manipulation of information.

In addition, the database system need to ensure the safety of the information stored, despite of system crashes or attempts at unauthorized access. The system should have mechanisms to avoid possible anomalous results, when data are shared.

1.3.2 Applications of Database Management System (DBMS):

A Database management system is a repository or a container for collection of computerized data files. The overall purpose of DBMS is to allow the users to define, store, retrieve and update the information contained in the database on demand.

Some of the major areas of application are as follows:

1. Banking
2. Airlines
3. Universities
4. Manufacturing and selling
5. Human resources
6. Databases touch all aspects of our lives.

Enterprise Information

- *Sales*: For product, customer, purchase and information.
- *Accounting*: For payments, receipts, account balances, assets and other accounting information.
- *Human resources*: For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.
- *Manufacturing*: For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.
- *Online retailers*: For sales data noted above plus online order tracking, generation of recommendation lists, and maintenance of online product evaluations.
- *Universities*: For student information, course registrations, and marks (in addition to standard enterprise information such as human resources and accounting).
- *Airlines*: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
- *Telecommunication*: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks

Banking and Finance

- *Banking*: For customer information, accounts, loans, and banking transactions.
- *Credit card transactions*: For purchases on credit cards and generation of monthly statements.
- *Finance*: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.

1.4 Database System Vs file System

1.4.1 Problems with File System

Prior to the usage of Database Management Systems (DBMSs), information are stored and maintained in **File systems**. A **file system** is a way of organizing information on a storage device like a computer hard drive. **Common file systems** include NTFS, FAT, etc. This typical file-processing system is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files.

Keeping organizational information in a file processing system has a number of major problems:

- **Data redundancy and inconsistency**: Same information may be duplicated in several places (files).

NOTES

For example, if a student enrols for double major (say, fine arts and mathematics) the basic details such as, address and telephone number of that student may appear in two different files viz. records of students of Fine Arts department and in the Mathematics department. This redundancy leads to higher storage and access cost.

- **Data inconsistency**: Copies of the same data in different files don't have inter-connection to maintain consistency.

  For example, an update of student address in Fine Arts Department will not automatically change in the records of Mathematics Department.

- **Difficulty in accessing data**: Conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. Data-retrieval systems are not efficient in file systems.

  For example, if we wish to find out the names of all students of the University who live within a particular city. It is very difficult to retrieve that information, because the details of students are maintained in different files. The files don't have inter-relationships to access the data across the files.

- **Data isolation**: In file system, the data are scattered across files, and files may have different formats. Hence, the data in file system are isolated. Developing application programs to retrieve those isolated data is difficult.

- **Integrity problems**. The data values stored in the database must satisfy certain types of **consistency constraints**.

  For example, suppose the university maintains separate accounts for each department about the grants given them. In case, the university needs to ensure that the account balance of a department may never goes below Rs. 1000. Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

- **Atomicity problems**: All the operations on data must be atomic – that is, it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

- **Concurrent-access anomalies**: To improve the overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. Indeed, today, the largest Internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data.

- **Security problems**. Not every user of the database system should be able to access all the data.

For example, in a university, staff of finance section should be permitted to see only that part of the database that has financial information. They do not need access to information about marks of the students. Enforcing such security constraints is difficult in file system.

These difficulties, among others, prompted the development of database systems.

1.4.2 Advantages of Database Systems

- **Controlling of Redundancy**: Data redundancy refers to the duplication of data (i.e storing same data multiple times). In a database system, by having a centralized database and centralized control of data by the DBA the unnecessary duplication of data is avoided. It also eliminates the extra time for processing the large volume of data. It results in saving the storage space.

- **Improved Data Sharing**: DBMS allows a user to share the data in any number of application programs.
- **Data Integrity**: Integrity means that the data in the database is accurate. Centralized control of the data helps in permitting the administrator to define integrity constraints to the data in the database. For example: in student database, we can enforce integrity that it must accept the students only from Tamil Nadu state for the admission under Distance Education mode.

- **Security:** Having complete authority over the operational data, enables the DBA in ensuring that the only mean of access to the database is through proper channels. The DBA can define authorization checks to be carried out whenever access to sensitive data is attempted.

- **Data Consistency:** By eliminating data redundancy, we greatly reduce the opportunities for inconsistency. For example: is a customer address is stored only once, we cannot have disagreement on the stored values. Also updating data values is greatly simplified when each value is stored in one place only. Finally, we avoid the wasted storage that results from redundant data storage.

- **Efficient Data Access :** In a database system, the data is managed by the DBMS and all access to the data is through the DBMS providing a key to effective data processing

- **Enforcements of Standards**: With the centralized of data, DBA can establish and enforce the data standards which may include the naming conventions, data quality standards etc.

- **Data Independence**: In a database system, the database management system provides the interface between the application programs and the data. When changes are made to the data representation, the meta-data obtained by the DBMS is changed but the DBMS is continues to provide the data to application program in the

previously used way. The DBMs handles the task of transformation of data wherever necessary.

- **Reduced Application Development and Maintenance Time :** DBMS supports many important functions that are common to many applications, accessing data stored in the DBMS, which facilitates the quick development of application

The following table 1.1 illustrates the comparison between Database Systems and File Systems:

Table 1.1 Database Systems Vs File Systems

Database Systems	File Systems
Multi-user access	It does not support multi-user access
Design to fulfill the need for small and large businesses	It is only limited to smaller DBMS system.
Remove redundancy and Integrity	Redundancy and Integrity issues
Expensive. But in the long term Total Cost of Ownership is cheap	It's cheaper
Easy to implement complicated transactions	No support for complicated transactions

1.5 View of Data

As it is already said, the DBMS is a collection of interrelated data and a set of programs that allow users to access and modify these data. A major purpose of a database system is to provide users with an *abstract* view of the data. That is, the system hides certain details of how the data are stored and maintained.

1.6 Data Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers to use complex data structures to represent data in the database. Since many database-system users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system. The following figure 1.1 depicts the three levels of abstraction of data in DBMS:

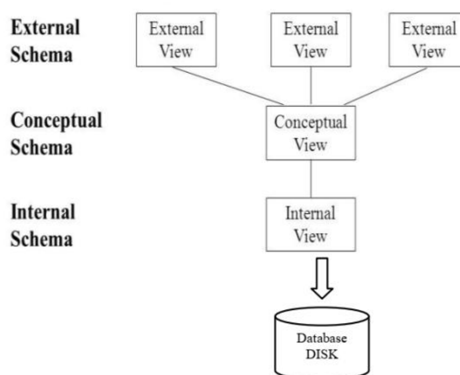


Figure 1.1 Levels of Abstraction in a DBMS

- **Physical level (or Internal View / Schema)**: The lowest level of abstraction describes *how* the data are actually stored. The physical level describes complex low-level data structures in detail.

- **Logical level (or Conceptual View / Schema)**: The next-higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data independence**. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

- **View level (or External View / Schema):** The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

1.7 Instances and Schemas

- Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database.
- The overall design of the database is called the database **schema**.
- Schemas are changed infrequently, if at all. The concept of database schemas and instances can be understood by analogy to a program written in a programming language.
- A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema. Database systems have several schemas, partitioned according to the levels of abstraction.
- The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level.
- A database may also have several schemas at the view level, sometimes called **subschemas**, which describe different views of the database. Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes

NOTES

186 Data Models

Data model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a database at the physical, logical, and view levels.

Data Models are fundamental entities to introduce abstraction in a DBMS. Data models define how data is connected to each other and how they are processed and stored inside the system.

The Data models are as follows:

- Earlier, the **Network** data model and **Hierarchical** data model are used. Due to its complexity, both are not used in the present scenario.
- Then comes the **Entity-Relationship Model**. The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects. An entity is a “thing” or “object” in the real world that is distinguishable from other objects. The entity relationship model is widely used in database design.
- **Relational Model**. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**.
- **Object-Based Data Model**. Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. The **object-relational data model** combines features of the object-oriented data model and relational data model.
- **Semi-structured Data Model**. The semi-structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The **Extensible Markup Language (XML)** is widely used to represent semi-structured data.

Check Your Progress

1. What is meant by Data?
2. Which one is useful, whether File Systems or Database Systems to main the data effectively?
3. What are the data models are not very much practiced in current DBMS tools?

1.9 The ER Model

Entity-Relationship (ER) Model is a notion based model. It denotes the real-world entities and relationships among them. To formulate real-world scenario into the database model, the ER Model is used to create entity set, relationship set, general attributes and constraints.

ER Model is best used for the conceptual design of a database.

ER Model is based on −

- **Entities** and their *attributes.*
- **Relationships** among entities.

These concepts are explained in the following figure 1.2:

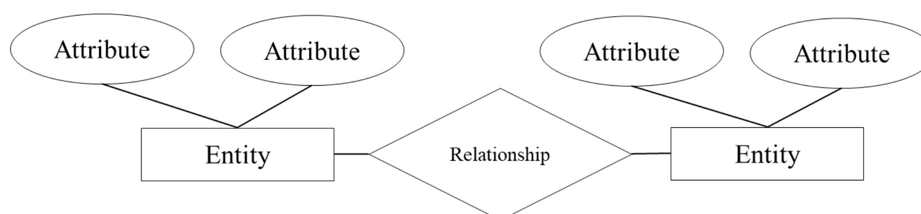


Figure 1.2 E-R Model

- **Entity** − an entity in an ER Model is a real-world entity having properties called **attributes**. Every **attribute** is defined by its set of values called **domain**. For example, in a school database, a student is considered as an entity. Student has various attributes like name, age, class, etc.

- **Relationship** − the logical association among entities is called ***relationship***. Relationships are mapped with entities in various ways. Mapping cardinalities define the number of association between two entities.

Mapping cardinalities −

- one to one
- one to many
- many to one
- many to many

1.10 Answers to Check Your Progress Questions

1. Raw fact is said to be Data
2. Because of its features such as, controlling redundancy, maintaining consistency, data independence, security, etc. the Database Systems are very much useful over File systems for maintaining data.
3. Due to its complexity in maintaining data, the **Network** data model and **Hierarchical** data model are not very much practiced.

NOTES

1.11 Summary

- Raw facts that can be recorded and that have implicit meaning are called Data.
- Processed data is called information.
- Database is a logical collection of inter-related data relevant to an enterprise
- Database Management System (DBMS) is a collection of interrelated data and a set of programs to access those data
- A Database management system is a repository or a container for collection of computerized data files.
- To simplify users' interactions with the system, the DBMS has three levels of abstraction, physical level, logical level and view level
- The collection of information stored in the database at a particular moment is called an instance of the database.
- The overall design of the database is called the database schema.
- Data model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.
- A data model provides a way to describe the design of a database at the physical, logical, and view levels.

1.12 Keywords

- **Data** is a raw fact. Processed data are called **information**.
- Collection of inter-related data is said to be **Database**.
- Set of procedures to access and manage the data in Database are called **Database Management Systems (DBMS)**.
- **Instance** is the collection of information stored in the database at a particular moment.
- The **relational model** uses a collection of tables to represent both data and the relationships among those data.
- **Data model** is a collection of conceptual tools.

1.13 Self-Assessment Questions and Exercises

Short Answer Questions:

1. Define: Data
2. What is meant by Database?
3. What are the data models?
4. What is a schema?
5. What is an entity?

Long Answer Questions:

1. Compare the features of File Systems and Database Systems.
2. Describe the characteristics of various Data models.
3. Write short-notes on the following:
    a. Instances
    b. View of Data
    c. Data Abstraction
    d. E-R model

1.14 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011.

UNIT II DATABASE MODELS

Structure

2.1 Introduction
2.2 Objectives
2.3 Relational Model
2.4 Database Languages
    2.4.1 DDL
    2.4.2 DML
2.5 Database Access for applications Programs
2.6 Database Users and Administrator
2.7 Transaction Management
2.8 Data base System Structure
2.9 Storage Manager
2.10 The Query Processor
2.11 Answers to Check Your Progress Questions
2.12 Summary
2.13 Key Words
2.14 Self-Assessment Questions and Exercises
2.15 Further Readings

2.1 Introduction

Data models define the logical structure of a **database**. **Data Models** are fundamental entities to introduce abstraction in a **DBMS**. **Data models** define how **data** is connected to each other and how they are processed and stored inside the system. A Database model defines the logical design and structure of a database and defines how data will be stored, accessed and updated in a database management system. While the **Relational Model** is the most widely used database model, there are other models too:

- Hierarchical Model
- Network Model
- Entity-relationship Model
- Relational Model

2.2 Objectives

This chapter delivers the fundamentals about:

- Data models
- Principles of Relational model
- Database Languages
- Components of Database Architecture

2.3 Relational Model

The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name.

The data is arranged in a relation which is visually represented in a two dimensional table. The data is inserted into the table in the form of tuples (which are nothing but rows). A tuple is formed by one or more than one attributes, which are used as basic building blocks in the formation of various expressions that are used to derive meaningful information. There can be any number of tuples in the table, but all the tuple contain fixed and same attributes with varying values.

The relational model is implemented in database where a relation is represented by a table, a tuple is represented by a row, an attribute is represented by a column of the table, attribute name is the name of the column such as 'identifier', 'name', 'city' etc., attribute value contains the value for column in the row. Constraints are applied to the table and form the logical schema.

In order to facilitate the selection of a particular row/tuple from the table, the attributes i.e. column names are used, and to expedite the selection of the rows some fields are defined uniquely to use them as indexes, this helps in searching the required data as fast as possible.

All the relational algebra operations, such as Select, Intersection, Product, Union, Difference, Project, Join, Division, Merge etc. can also be performed on the Relational Database Model. Operations on the Relational Database Model are facilitated with the help of different conditional expressions, various key attributes, pre-defined constraints etc.

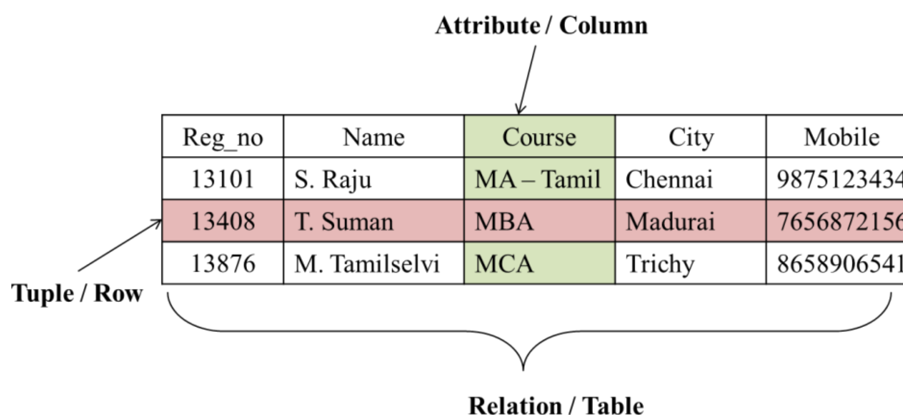


Figure 2.1 Relational Model

The main highlights of this model are −

- Data is stored in tables called **relations**.
- Relations can be normalized.
- In normalized relations, values saved are atomic values.

NOTES

- Each row in a relation contains a unique value.
- Each column in a relation contains values from a same domain.

2.4 Database Languages

A database system provides a **Data-Definition Language (DDL)** to specify the database schema and a **data-manipulation language** to express database queries and updates. In practice, the data definition and data-manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

2.4.1 Data-Definition Language (DDL)

A database schema is specified by a set of definitions expressed by a special language called a **Data-Definition Language** (**DDL**). The DDL is also used to specify additional properties of the data. We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a **data storage and definition** language. These statements define the implementation details of the database schemas, which are usually hidden from the users.

The data values stored in the database must satisfy certain **consistency constraints**. For example, suppose the university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, database systems implement integrity constraints that can be tested with minimal overhead.

- **Domain Constraints**. A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.
- **Referential Integrity**. There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. More precisely, the *dept name* value in a *course* record must appear in the *dept name* attribute of some record of the *department* relation. Database modifications can cause violations of referential integrity. When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- **Assertions**. An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms.

For example, "Every department must have at least five courses offered every semester" must be expressed as an assertion. When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.

- **Authorization**. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being: **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; **update authorization**, which allows modification, but not deletion, of data; and **delete authorization**, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization. The DDL, just like any other programming language, gets as input some instructions (statements) and generates some output. The output of the DDL is placed in the **data dictionary**, which contains **metadata**—that is, data about data. The data dictionary is considered to be a special type of table that can only be accessed and updated by the database system itself (not a regular user). The database system consults the data dictionary before reading or modifying actual data

2.4.2 Data- Manipulation Language (DML)

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model. The types of access are:

- Retrieval of information stored in the database
- Insertion of new information into the database
- Deletion of information from the database
- Modification of information stored in the database

There are basically two types:

- **Procedural DMLs** require a user to specify *what* data are needed and *how* to get those data.
- **Declarative DMLs** (also referred to as **nonprocedural DMLs**) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data. A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data-manipulation language* synonymously.

There are a number of database query languages in use, either commercially or experimentally.

The levels of abstraction apply not only to defining or structuring data, but also to manipulating data. At the physical level, we must define

NOTES

algorithms that allow efficient access to data. At higher levels of abstraction, we emphasize ease of use. The goal is to allow humans to interact efficiently with the system. The query processor component of the database system translates DML queries into sequences of actions at the physical level of the database system.

2.5 Database Access for applications Programs

A **database application** is a computer **program** whose primary purpose is entering and retrieving information from a computerized **database**.

When a user wants to retrieve data from the Database, the database management system is acts a bridge between the application program, (that determines what data are needed and how they are processed), and the operating system of the computer, which is responsible for placing data on the magnetic storage devices.

To retrieve data from the database, the following operations are performed internally:

- A user issues an access request, using some application program or data manipulation language.
- The application program determines what data are needed and communicates the need to the database management system.
- The DBMS intercepts the request and interprets it
- The DBMS inspects, in turn, the external schema, the external/conceptual mapping, the conceptual schema, the conceptually internal mapping, and storage structure definition.
- The data base management system instructs the operating system to locate and retrieve the data from the specific location on the magnetic disk (or whatever device it is stored on).
- A copy of the data is given to the application program for processing.

There are also special types of programming languages that combine imperative control structures (for example, for loops, while loops and if-then-else statements) with statements of the data manipulation language. These languages, sometimes called *fourth-generation languages*, often include special features to facilitate the generation of forms and the display of data on the screen. Most major commercial database systems include a fourth generation language.

2.6 Database Users and Administrator

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

- **Naive users** are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously.
- **Application programmers** are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program.
- **Sophisticated users** interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category. **Online analytical processing (OLAP)** tools simplify analysts' tasks by letting them view summaries of data in different ways. For instance, an analyst can see total sales by region (for example, North, South, East, and West), or by product, or by a combination of region and product (that is, total sales of each product in each region). The tools also permit the analyst to select specific regions, look at data in more detail (for example, sales by city within a region) or look at the data in less detail (for example, aggregate products together by category). Another class of tools for analysts is **data mining** tools, which help them find certain kinds of patterns in data.
- **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modelling systems.

2.7 Transaction Management

A **transaction** is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database-consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates.

Transaction management ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

2.8 Database System Structure

The following Figure 2.2 depicts the architecture of a database system. The various components shown in the figure 2.2 gives an overview of the components of a database system and the connections among them.

NOTES

The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs. Database systems can be centralized, or client-server, where one server machine executes work on behalf of multiple client machines. Database systems can also be designed to exploit parallel computer architectures. Distributed databases span multiple geographically separated machines.

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the **storage manager** and the **query processor** components. The storage manager is important because databases typically require a large amount of storage space. The query processor is important because it helps the database system simplify and facilitate access to data.

It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

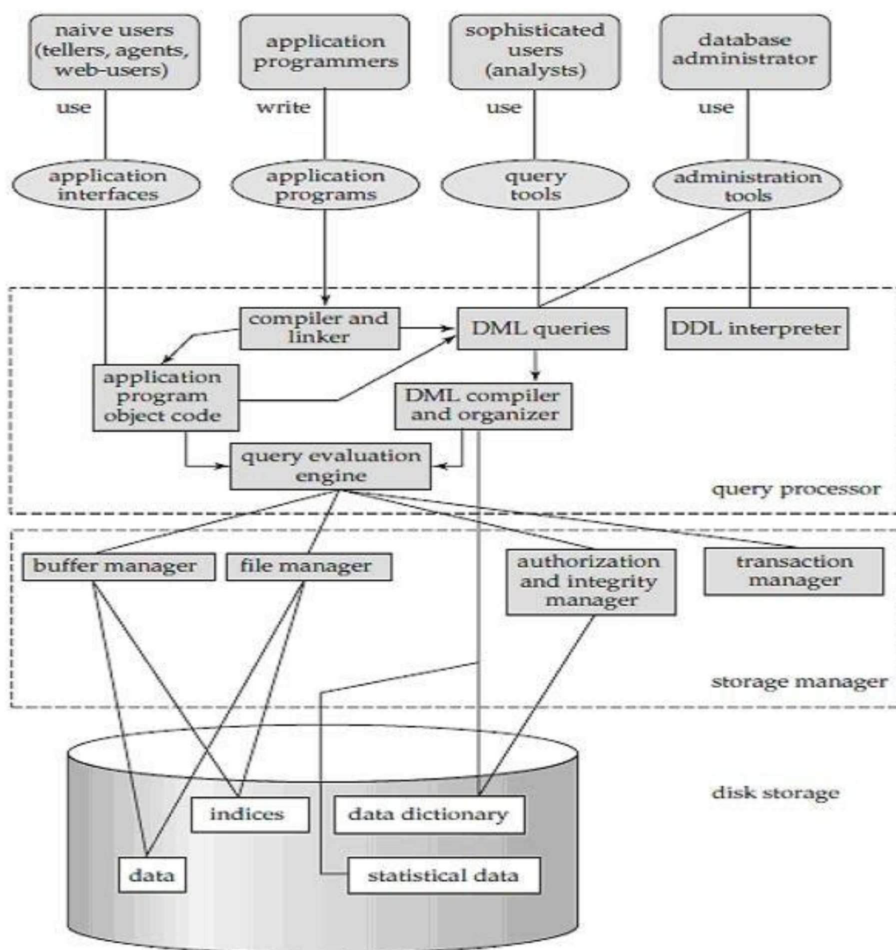


Figure 2.2 Database System Architecture

Database applications are usually partitioned into two or three parts, as in Figure 2.3.

In two-tier architecture, the application resides at the client machine, where it invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

In contrast, in three-tier architecture, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface. The application server in turn communicates with a database system to access data. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients.

Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web (WWW).

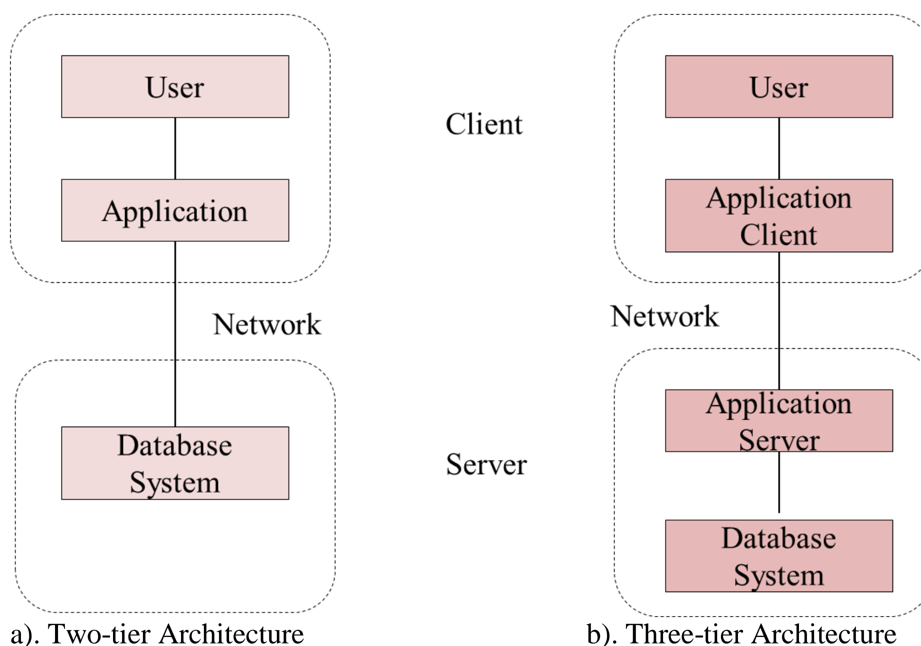


Figure 2.3 Two-tier and Three-tier Architectures

2.9 Storage Manager

A *storage manager* is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The raw data are stored on the disk using the file system, which is usually provided by a conventional operating system. The storage manager translates the various DML statements into low-level file-system commands.

NOTES

Thus, the storage manager is responsible for storing, retrieving, and updating data in the database. The storage manager components include:

- **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
- **Transaction manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.
- **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
- **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

Check Your Progress

1. How data are arranged in Relational model?
2. Define: DML.
3. What is meant by Transaction?

2.10 The Query Processor

The query processor components include

- DDL interpreter, which interprets DDL statements and records the definitions in the data dictionary.
- DML compiler, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs query optimization, that is, it picks the lowest cost evaluation plan from among the alternatives. Query evaluation engine, which executes low-level instructions generated by the DML compiler.

2.11 Answers to Check Your Progress Questions

1. The data is arranged in a relation which is visually represented in a two dimensional table.
2. A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model.

3. A **transaction** is a collection of operations that performs a single logical function in a database application.

2.12 Summary

- The relational model uses a collection of tables to represent both data and the relationships among those data.
- The data is arranged in a relation which is visually represented in a two dimensional table. The data is inserted into the table in the form of tuples (which are nothing but rows)
- A **Data-Definition Language (DDL)** to specify the database schema.
- A **Data-Manipulation Language (DML)** enables users to access or manipulate data.
- Procedural DMLs and Declarative DMLs are the two types of DML
- A **query** is a statement requesting the retrieval of information.
- The Naive users, Application programmers, Sophisticated users and the Specialized users are the types of users of Database Systems
- A transaction is a collection of operations that performs a single logical function in a database application
- In two-tier architecture, the application resides at the client machine invokes database system functionality at the server machine through query language statements.
- In three-tier architecture, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface.
- A *storage manager* is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- A query processor consists of DML interpreter and DML compiler

2.13 Keywords

- **Relational model** is said to be a collection of tables
- **Tables** contains tuples (rows) and attributes (columns)
- **DBMS** consists of languages called DDL and DML
- A **database application** is a computer **program** whose primary purpose is entering and retrieving information from a computerized **database**.

NOTES

- A **storage manager** is an interface between the low-level data and the application programs and queries.

- A **query** can usually be translated into any of a number of alternative evaluation plans that all give the same result.

2.14 Self-Assessment Questions and Exercises

Short Answer Questions:

1. Define: Relational model
2. What are the two types of DML?
3. Who is Naïve user?
4. What are the functional components of Database System?
5. What is a query?

Long Answer Questions:

1. List and explain the constraints applied to the Tables.
2. Describe the components of a Database Architecture.
3. Write short-notes on the following:
    a. Storage manager
    b. Transaction manager
    c. Query processor
    d. Three-tier architecture

2.15 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011.

UNIT III HISTORY OF DATABASE SYSTEMS

Structure

3.1 Introduction
3.2 Objectives
3.3 Database design and ER diagrams
3.4 Beyond ER Design Entities, Attributes and Entity sets
3.5 Relationships and Relationship sets
3.6 Additional features of ER Model
3.7 Concept Design with the ER Model
3.8 Conceptual Design for Large enterprises
3.9 Answers to Check Your Progress Questions
3.10 Summary
3.11 Key Words
3.12 Self-Assessment Questions and Exercises
3.13 Further Readings

3.1 Introduction

The ER or (**Entity Relational Model**) is a high-level conceptual data model diagram. Entity-Relation model is based on the notion of real-world entities and the relationship between them. ER modeling helps you to analyze data requirements systematically to produce a well-designed database. An **Entity–relationship model (ER model)** describes the structure of a database with the help of a diagram, which is known as **Entity Relationship Diagram (ER Diagram)**. An ER model is a design or blueprint of a database that can later be implemented as a database. The main components of E-R model are: entity set and relationship set.

3.2 Objectives

This chapter gives introduction to:

- ER Model of Database
- Elements of ER diagrams
- Database design and ER diagrams

3.3 Database design and ER diagrams

Database Design Techniques

The database design process aims to create database structures that will efficiently store and manage data. Database design has four phases: requirements analysis, conceptual design, logical design, and physical design.

NOTES

There are two techniques involved in designing a Database:
1. ER Modeling (Top down Approach)
2. Normalization (Bottom Up approach)

Top-down approach Vs Bottom-up Approach

Top-down approaches stress an initial focus on knowledge of higher-level constructs, such as identification of populations and collections of things and entity types, membership rules, and relationships between such populations. Adoption of a top-down approach will generally start with a set of high-level requirements, such as a narrative. These requirements start a process of identifying the types of things needed to represent data with as well as the attributes of those things, which may become attributes in tables. In the top-down database design tradition, the database analyst initially attempts to develop a conceptual data model by identifying highly abstracted data objects (things/entity types) that may exist within the domain—i.e., the analyst attempts to construct a domain ontology. Techniques applied by the analyst typically include making observations, conducting interviews, and other data collection strategies.

Usually, inspiration for the data model also comes from a close analysis of the domain business rules. In addition, structural properties, such as relationships between entity types and relationship cardinality are identified. In many cases, an initial conceptual data model is drafted that does not include all data attributes. Once a satisfactory conceptual data model has been developed, the database analyst may turn his/her attention to the technological platform on which the final data repository will be deployed (i.e., development of the logical data schema). Development of the logical schema requires the database analyst to consider any mapping issues between the structures on the **ER (Entity-Relationship) model** and chosen persistent mechanism.

Bottom-up approaches view database design as proceeding from an initial analysis of lower-level conceptual units, such as attributes and functional dependencies and then moving towards an acceptable logical data model through logical groupings of associated attributes. In other words, bottom-up approaches tend to view the task of population identification as a process of generalizing object identity from examples of structural dependencies (e.g., bundling/categorizing attributes that appear to co-occur). Input into a bottom-up approach, for example, could be views of data, such as screen shots or reports (printouts), or patterns of co-occurring attribute values identified within large datasets. A well-known approach to database design that can be used as a bottom-up approach is **Normalization.**

3.4 Beyond ER Design Entities, Attributes and Entity sets

The **entity-relationship** (**E-R**) data model was developed to facilitate database design by allowing specification of an *enterprise schema* that represents the overall logical structure of a database. The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the E-R model. The E-R

data model employs three basic concepts: entity sets, relationship sets, and attributes, which we study first.

Entity Sets

An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in a university is an entity. An entity has a set of properties, and the values for some set of properties may uniquely identify an entity. For instance, a person may have a *person id* property whose

value uniquely identifies that person. Thus, the value 677-89-9011 for *person id* would uniquely identify one particular person in the university. Similarly, courses can be thought of as entities, and *course id* uniquely identifies a course entity in the university. An entity may be concrete, such as a person or a book, or it may be abstract, such as a course, a course offering, or a flight reservation.

An entity set is a set of entities of the same type that share the same properties, or attributes. The set of all people who are instructors at a given university, for example, can be defined as the entity set *instructor*. Similarly, the entity set *student* might represent the set of all students in the university. In the process of modeling, we often use the term *entity set* in the abstract, without referring to a particular set of individual entities. We use the term extension of the entity set to refer to the actual collection of entities belonging to the entity set. Thus, the set of actual instructors in the university forms the extension of the entity set *instructor*.

Entity sets do not need to be disjoint. For example, it is possible to define the entity set of all people in a university (*person*). A *person* entity may be an *instructor* entity, a *student* entity, both, or neither.

An entity is represented by a set of attributes. Attributes are descriptive properties possessed by each member of an entity set. The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute. Possible attributes of the *instructor* entity set are *ID*, *name*, *dept name*, and *salary*. In real life, there would be further attributes, such as street number, apartment number, state, postal code, and country, but we omit them to keep our examples simple. Possible attributes of the *course* entity set are *course id*, *title*, *dept name*, and *credits*. Each entity has a value for each of its attributes. For instance, a particular *instructor* entity may have the value 12121 for *ID*, the value Wu for *name*, the value Finance for *dept name*, and the value 90000 for *salary*.

The *ID* attribute is used to identify instructors uniquely, since there may be more than one instructor with the same name. In the United States, many enterprises find it convenient to use the *social-security* number of a person2 as an attribute whose value uniquely identifies the person. In general the enterprise would have to create and assign a unique identifier for each instructor.

A database thus includes a collection of entity sets, each of which contains any number of entities of the same type. simple, only some of the attributes of the two entity sets are shown. A database for a university may include a number of other entity sets. For example, in addition to keeping track of instructors and students, the university also has information about courses, which are represented by the entity set *course*

NOTES

3.5 Relationships and Relationship sets

A **relationship** is an association among several entities. For example, we can define a relationship *advisor* that associates instructor Katz with student Shankar. This relationship specifies that Katz is an advisor to student Shankar.

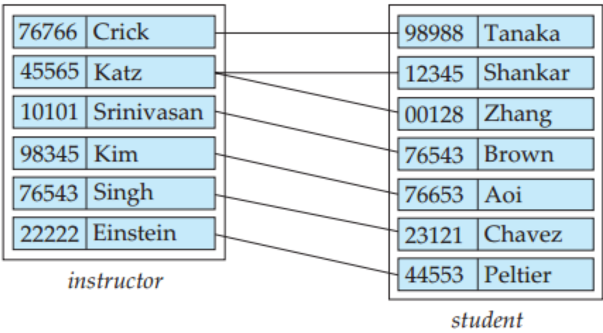
A **relationship set** is a set of relationships of the same type. Formally, it is a mathematical relation on $n \geq 2$ (possibly nondistinct) entity sets. If $E1, E2, \ldots, En$ are entity sets, then a relationship set R is a subset of $\{(e1, e2, \ldots, en) \mid e1 \in E1, e2 \in E2, \ldots, en \in En\}$ where $(e1, e2, \ldots, en)$ is a relationship. Consider the two entity sets *instructor* and *student*. We define the relationship set *advisor* to denote the association between instructors and students.

As another example, consider the two entity sets *student* and *section*. We can define the relationship set *takes* to denote the association between a student and the course sections in which that student is enrolled.

The association between entity sets is referred to as participation; that is, the entity sets $E1, E2, \ldots, En$ **participate** in relationship set R. A **relationship instance** in an E-R schema represents an association between the named entities in the real-world enterprise that is being modeled. As an illustration, the individual *instructor* entity Katz, who has instructor *ID* 45565, and the *student* entity Shankar, who has student *ID* 12345, participate in a relationship instance of *advisor*. This relationship instance represents that in the university, the instructor Katz is advising student Shankar.

The function that an entity plays in a relationship is called that entity's **role**. Since entity sets participating in a relationship set are generally distinct, roles

Relationship Set Advisor



are implicit and are not usually specified. However, they are useful when the meaning of a relationship needs clarification. Such is the case when the entity sets of a relationship set are not distinct; that is, the same entity set participates in a relationship set more than once, in different roles. In this type of relationship set, sometimes called a **recursive** relationship set, explicit role names are necessary to specify how an entity participates in a relationship instance. For example, consider the entity set *course* that records

information about all the courses offered in the university. To depict the situation where one course (C2) is a prerequisite for another course (C1) we have relationship set *prereq* that is modeled by ordered pairs of *course* entities. The first course of a pair takes the role of course C1, whereas the second takes the role of prerequisite course C2. In this way, all relationships of *prereq* are characterized by (C1, C2) pairs; (C2, C1) pairs are excluded.
A relationship may also have attributes called **descriptive attributes**. Consider a relationship set *advisor* with entity sets *instructor* and *student*.

3.6 Additioal features of ER Model

- **Specialization** – The process of designating to sub grouping within an entity set is called specialization. In above figure, the "person" is distinguish in to whether they are "employee" or "customer".
  Formally in above figure specialization is depicted by a triangle component labelled (is a), means the *customer* is a *person*.
  Sometime this ISA (is a) referred as a superclass-subclass relationship. This is also used to emphasize on to creating the distinct lower level entity sets.

- **Generalization** – generalization is relationship that exist between higher level entity set and one or more lower level entity sets. Generalization synthesizes these entity sets into single entity set.

- **Higher level and lower level entity sets** – This property is created by specialization and generalization. The attributes of higher level entity sets are inherited by lower level entity sets.
  For example: In above figure "customers" and "employee" inherits the attributes of "person".

- **Attribute inheritance**: When given entity set is involved as a lower entity set in only one "ISA" (is a) relationship, it is referred as a *single attribute inheritance*. If lower entity set is involved in more than one ISA (is a) relationship, it is referred as a *multi attribute inheritance.*

- **Aggregation**: there is a one limitation with E-R model that it cannot express relationships among relationships. So aggregation is an abstraction through which relationship is treated as *higher level entities*.

3.7 Concept Design with the ER Model

Entity-Relationship model is used in the conceptual design of a database (☞ conceptual level, conceptual schema)
Design is independent of all physical considerations (DBMS, OS, . . .).
A database schema in the ER model can be represented pictorially (Entity-Relationship diagram)
A graphical technique for understanding and organizing the data independent of the actual database implementation we need to be familiar with the following terms to go further.

NOTES

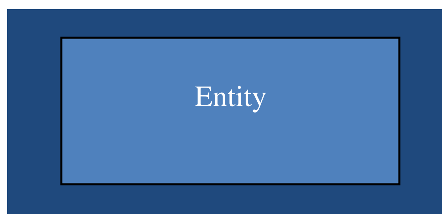
Entity

Any thing that has an independent existence and about which we collect data. It is also known as entity type. In ER modeling, notation for entity is given below.

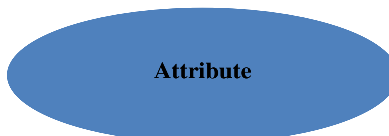
Entity instance
Entity instance is a particular member of the entity type. Example for entity instance : A particular employee

Regular Entity
An entity which has its own key attribute is a regular entity. Example for regular entity: Employee.

Weak entity
An entity which depends on other entity for its existence and doesn't have any key attribute of its own is a weak entity. Example for a weak entity. In a parent/child relationship, a parent is considered as a strong entity and the child is a weak entity. In ER modeling, notation for weak entity is given below



Attributes
Properties/characteristics which describe entities are called attributes. In ER modeling, notation for attribute is given below.



Domain of Attributes
The set of possible values that an attribute can take is called the domain of the attribute. For example, the attribute day may take any value from the set {Monday, Tuesday ... Friday}. Hence this set can be termed as the domain of the attribute day.

Key attribute
The attribute (or combination of attributes) which is unique for every entity instance is called key attribute. E.g the employee_id of an employee, pan_card_number of a person and etc. If the key attribute consists of two or more attributes in combination, it is called a composite key. In ER modeling, notation for key attribute is given below.

Simple attribute
If an attribute cannot be divided into simpler components, it is a simple attribute. Example for simple attribute : employee_id of an employee.

Composite attribute
If an attribute can be split into components, it is called a composite attribute. Example for composite attribute : Name of the employee which can be split into First_name, Middle_name, and Last_name.

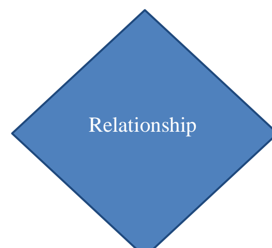
Single valued Attributes
If an attribute can take only a single value for each entity instance, it is a single valued attribute. Example for single valued attribute : age of a student. It can take only one value for a particular student.

Multi-valued Attributes
If an attribute can take more than one value for each entity instance, it is a multi-valued attribute. Example for multi valued attribute : telephone number of an employee, a particular employee may have multiple telephone numbers. In ER modeling, notation for multi-valued attribute is given below

Stored Attribute
An attribute which need to be stored permanently is a stored attribute Example for stored attribute : name of a student

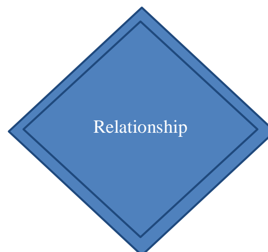
Derived Attribute
An attribute which can be calculated or derived based on other attributes is a derived attribute. Example for derived attribute : age of employee which can be calculated from date of birth and current date. In ER modeling, notation for derived attribute is given below

Relationships
Associations between entities are called relationships. Example: An employee works for an organization. Here "works for" is a relation between the entities employee and organization. In ER modeling, notation for relationship is given below.



NOTES

However in ER Modeling, To connect a weak Entity with others, you should use a weak relationship notation as given below



Degree of a Relationship

Degree of a relationship is the number of entity types involved. The n-ary relationship is the general form for degree n. Special cases are unary, binary, and ternary ,where the degree is 1, 2, and 3, respectively. Example for unary relationship: An employee is a manager of another employee Example for binary relationship : An employee works-for department. Example for ternary relationship : customer purchase item from a shop keeper

Cardinality of a Relationship

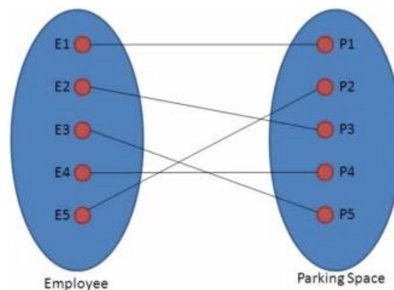
Relationship cardinalities specify how many of each entity type is allowed. Relationships can have four possible connectivity as given below.

1. One to one (1:1) relationship
2. One to many (1:N) relationship
3. Many to one (M:1) relationship
4. Many to many (M:N) relationship

The minimum and maximum values of this connectivity is called the cardinality of the relationship

Example for Cardinality – One-to-One (1:1)

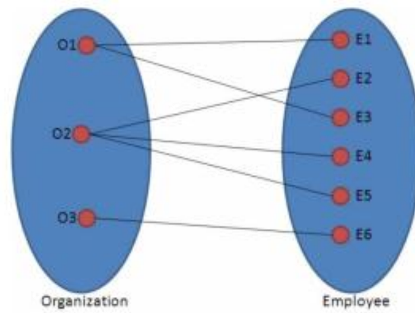
Employee is assigned with a parking space



One employee is assigned with only one parking space and one parking space is assigned to only one employee. Hence it is a 1:1 relationship and cardinality is One-To-One (1:1) In ER modeling, this can be mentioned using notations as given below

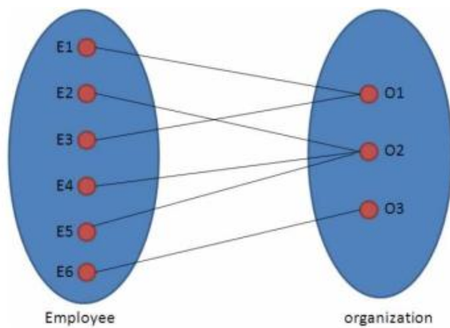
Example for Cardinality – One-to-Many (1:N)

Organization has employees



One organization can have many employees , but one employee works in only one organization. Hence it is a 1:N relationship and cardinality is One-To-Many (1:N) In ER modeling, this can be mentioned using notations as given below

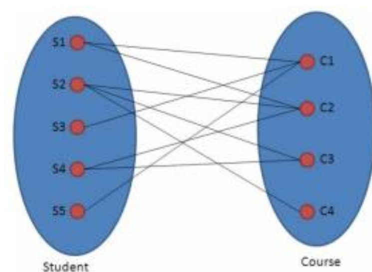
Example for Cardinality – Many-to-One (M :1)
It is the reverse of the One to Many relationship. employee works in organization



One employee works in only one organization But one organization can have many employees. Hence it is a M:1 relationship and cardinality is Many-to-One (M :1)

In ER modeling, this can be mentioned using notations as given below.

Cardinality – Many-to-Many (M:N)
Students enrols for courses



One student can enrol for many courses and one course can be enrolled by many students. Hence it is a M:N relationship and cardinality is Many-to-Many (M:N)

Check Your Progress

1. What are the phases in database design?
2. Define: ER model
3. What is meant by an entity?

3.8 Conceptual Design for Large enterprises

The process of conceptual design consists of more than just describing small fragments of the application in terms of ER diagrams.
For a large enterprise, the design may require the efforts of more than one designer and span data and application code used by a number of user groups.
An important aspect of the design process is the methodology used to structure the development of the overall design and to ensure that the design takes into account all user requirements and is consistent.
The usual approach is that the requirements of various user groups are considered, any conflicting requirements are somehow resolved, and a single set of global requirements is generated at the end of the requirements analysis phase.
An alternative approach is to develop separate conceptual schemas for different user groups and to then *integrate* these conceptual schemas.
To integrate multiple conceptual schemas, we must establish correspondences between entities, relationships, and attributes, and we must resolve numerous kinds of conflicts.

3.9 Answers to Check Your Progress Questions

1. Database design has four phases: requirements analysis, conceptual design, logical design, and physical design.
2. The entity-relationship **(E-R)** data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database.
3. An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects.

3.10 Summary

- The ER model is a high-level data model diagram
- ER diagrams are a visual tool which is helpful to represent the ER model Entity relationship diagram displays the relationships of entity set stored in a database
- ER diagrams help you to define terms related to entity relationship modeling
- ER model is based on three basic concepts: Entities, Attributes & Relationships

- An entity can be place, person, object, event or a concept, which stores data in the database
- Relationship is nothing but an association among two or more entities
- A weak entity is a type of entity which doesn't have its key attribute
- It is a single-valued property of either an entity-type or a relationship-type
- It helps you to defines the numerical attributes of the relationship between two entities or entity sets

3.11 Keywords

- A thing or object in the real world with an independent existence that can be differentiated from other objects is an **entity**
- A collection of entities of an entity type at a point of time is an **entity set**.
- A collection of similar entities are the **entity type**.
- An attribute in a table that references the primary key in another table OR it can be null is known as **foreign key** .
- **Composite attributes**, attributes that consist of a hierarchy of attributes
- **Composite key**: composed of two or more attributes, but it must be minimal
- **Attributes** that contain values calculated from other attributes are derived attributes

3.12 Self-Assessment Questions and Exercises

Short Answer Questions:

1. Define: Entity
2. What is meant by Entity set?
3. What are the features of ER model?
4. What is an attribute?

Long Answer Questions:

1. Describe the data base design and ER diagrams
2. Describe the features of the ER model
3. Write short notes on
    a. Conceptual Design with ER model
    b. Conceptual Design for Large enterprises

3.13 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
  Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011.

BLOCK – II
RELATIONAL MODEL

UNIT IV INTRODUCTION

Structure

4.1 Introduction
4.2 Objectives
4.3 Structure of Relational Model
4.4 Integrity Constraint over Relations
4.5 Enforcing Integrity constraints
4.6 Querying relational data
4.7 Logical Database Design
4.8 Introduction to Views
4.9 Destroying / altering Tables and Views
4.10 Answers to Check Your Progress Questions
4.11 Summary
4.12 Key Words
4.13 Self-Assessment Questions and Exercises
4.14 Further Readings

4.1 Introduction

The relational model represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship. In relational model, the data and relationships are represented by collection of inter-related tables. Each table is a group of column and rows, where column represents attribute of an entity and rows represents records.

4.2 Objectives

This chapter will impart the concepts of:

- Relational model
- Structure of relational model
- Integrity constraints
- Querying relational data
- views

4.3 Structure of Relational Model

The relational model is today the primary data model for commercial data processing applications. It attained its primary position because of its simplicity, which eases the job of the programmer, compared to earlier data models such as the network model or the hierarchical model. In this, we first study the fundamentals of the relational model. A substantial theory exists for relational databases.

Structure of Relational Databases:

A relational database consists of a collection of **tables**, each of which is assigned a unique name. For example, consider the *Faculty* table of Table 4.1, which stores information about Faculty members. The table has four column headers: *Fac_id, Fac_name, dept,* and *Position*. Each row of this table records information about a Faculty member, consisting of the Faculty member's *ID*, *name*, *dept name*, and *position*. Similarly, the *course* table of Table 4.2, stores information about courses, consisting of a *course id*, *title*, *dept name*, and *credits*, for each course. Note that each Faculty member is identified by the value of the column Fac*_id*, while each course is identified by the value of the column *course id*. Table 4.3 shows a third table, *prerequisites*, which stores the prerequisite courses for each course. The table has two columns, *course id* and *prereq id*. Each row consists of a pair of course identifiers such that the second course is a prerequisite for the first course. Thus, a row in the *prereq* table indicates that two courses are *related* in the sense that one course is a prerequisite for the other.

Table 4.1 Faculty Relation

Fac_id	Fac_name	Dept	Posisiton
F101	Dr. S. Raman	Comp. Sci.	Professor
F103	Dr. M. Balaji	Comp. Sci.	Asst. Professor
F204	Dr. M. John	Tamil	Asst. Professor
F206	Dr. K. Kumar	Tamil	Asst. Professor
F401	Dr. S. Sunil	English	Asst. Professor
F305	Dr. F. Mohamed	Maths	Professor
F306	Dr. K. Mala	Maths	Assoc. Professor

In general, a row in a table represents a *relationship* among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of *table* and the mathematical concept of *relation*, from which the relational data model takes its name. In mathematical terminology, a *tuple* is simply a sequence (or list) of values. A relationship between *n* values is represented mathematically by an *n-tuple* of values, i.e., a tuple with *n* values, which corresponds to a row in a table.

Table 4.2 Course Relation

Course_id	Title	Dept	Credits
CS-101	Fundamentals of Programming	Comp. Sci.	3
MA-101	Principles of Calculus	Maths	3
TA-102	Tamil Illakiya Varalaru	Tamil	3
TA-204	Tholkappium	Tamil	5
EN-101	Basic Communicative English	English	3
MA-205	Advanced Calculus	Maths	5
CS-204	Java Programming	Comp. Sci.	5

Table 4.3 Prerequisites Relation

Course_id	Prereq_id
TA-204	TA-102
MA-205	MA-101
CS-204	CS-101

Thus, in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table. Examining Table 4.1, we can see that the relation *Faculty* has four attributes: ***Fac_id, fac_name, dept, and position.*** We use the term **relation instance** to refer to a specific instance of a relation, i.e., containing a specific set of rows. The instance of *Faculty* shown in Table 4.1 has 7 tuples, corresponding to 7 Faculty members.

In this topic, we shall be using a number of different relations to illustrate the various concepts underlying the relational data model. These relations represent part of a university. They do not include all the data an actual university database would contain, in order to simplify our presentation. The order in which tuples appear in a relation is irrelevant, since a relation is a *set* of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Table 4.1, or are unsorted, as in Table 4.4, does not matter; the relations in the two figures are the same, since both contain the same set of tuples. For ease of exposition, we will mostly show the relations sorted by their first attribute. For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute. Thus, the domain of the *salary* attribute of the *Faculty* relation is the set of all possible salary values, while the domain of the *name* attribute is the set of all possible Faculty names.

We require that, for all relations *r*, the domains of all attributes of *r* be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units.

Table 4.4 Unsorted Faculty Relation

Fac_id	Fac_name	Dept	Posisiton
F101	Dr. S. Raman	Comp. Sci.	Professor
F306	Dr. K. Mala	Maths	Assoc. Professor
F204	Dr. M. John	Tamil	Asst. Professor
F601	Dr. M. Shahana	Physics	Assoc. Professor
F401	Dr. S. Sunil	English	Asst. Professor
F305	Dr. F. Mohamed	Maths	Professor

F206	Dr. K. Kumar	Tamil	Asst. Professor
F103	Dr. M. Balaji	Comp. Sci.	Asst. Professor

For example, suppose the table *Faculty* had an attribute *phone number*, which can store a set of phone numbers corresponding to the Faculty. Then the domain of *phone number* would not be atomic, since an element of the domain is a set of phone numbers, and it has subparts, namely the individual phone numbers in the set.

The important issue is not what the domain itself is, but rather how we use domain elements in our database. Suppose now that the *phone number* attribute stores a single phone number. Even then, if we split the value from the phone number attribute into a country code, an area code and a local number, we would be treating it as a non-atomic value. If we treat each phone number as a single indivisible unit, then the attribute *phone number* would have an atomic domain.

The **null** value is a special value that signifies that the value is unknown or does not exist. For example, suppose as before that we include the attribute *phone number* in the *Faculty* relation. It may be that a Faculty does not have a phone number at all, or that the telephone number is unlisted. We would then have to use the null value to signify that the value is unknown or does not exist. We shall see later that null values cause a number of difficulties when we access or update the database, and thus should be eliminated if at all possible. We shall assume null values are absent initially.

Creating a basic table involves naming the table and defining its columns and each column's data type.

The SQL **CREATE TABLE** statement is used to create a new table.

Syntax

The basic syntax of the CREATE TABLE statement is as follows −

```
CREATE TABLE table_name(
   column1 datatype,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
   PRIMARY KEY( one or more columns )
);
```

4.4 Integrity Constraints over Relations

Database integrity refers to the validity and consistency of stored data. Integrity is usually expressed in terms of constraints, which are consistency rules that the database is not permitted to violate. Constraints may apply to each attribute or they may apply to relationships between tables.

NOTES

Integrity constraints ensure that changes (update deletion, insertion) made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database.

4.4.1 Types of Integrity Constraints

Various types of integrity constraints are-

a. Domain Integrity
b. Entity Integrity Constraint
c. Referential Integrity Constraint
d. Key Constraints

a. **Domain Integrity**- Domain integrity means the definition of a valid set of values for an attribute. You define data type, length or size, is null value allowed , is the value unique or not for an attribute ,the default value, the range (values in between) and/or specific values for the attribute.

b. **Entity Integrity Constraint**- This rule states that in any database relation value of attribute of a primary key can't be null.

c. **Referential Integrity Constraint-**It states that if a foreign key exists in a relation then either the foreign key value must match a primary key value of some tuple in its home relation or the foreign key value must be null.

d. **Key Constraints-** A Key Constraint is a statement that a certain minimal subset of the fields of a relation is a unique identifier for a tuple.

There are 4 types of key constraints-

- Candidate key.
- Super key
- Primary key
- Foreign key

4.4.2 Key Constraints

We must have a way to specify how tuples within a given relation are distinguished. This is expressed in terms of their attributes. That is, the values of the attribute values of a tuple must be such that they can *uniquely identify* the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes.

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example, the *ID* attribute of the relation *Faculty* is sufficient to distinguish one Faculty tuple from another. Thus, *ID* is a superkey. The *name* attribute of *Faculty*, on the other hand, is not a superkey, because several Faculty members might have the same name. Formally, let *R* denote the set of attributes in the schema of relation *r*. If we say that a subset *K* of *R* is a *superkey* for *r* , we are restricting consideration to instances of relations *r* in which no two distinct tuples have the same values

on all attributes in *K*. That is, if $t1$ and $t2$ are in r and $t1 = t2$, then $t1.K = t2.K$.

A superkey may contain extraneous attributes. For example, the combination of *ID* and *name* is a superkey for the relation *Faculty*. If *K* is a superkey, then so is any superset of *K*. We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**.

It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of *name* and *dept name* is sufficient to distinguish among members of the *Faculty* relation. Then, both *{ID}* and *{name*, *dept name}* are candidate keys. Although the attributes *ID* and *name* together can distinguish *Faculty* tuples, their combination, *{ID*, *name}*, does not form a candidate key, since the attribute *ID* alone is a candidate key.

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation. A key (whether primary, candidate, or super) is a property of the entire relation, rather than of the individual tuples. Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time. The designation of a key represents a constraint in the real-world enterprise being modelled.

Primary keys must be chosen with care. As we noted, the name of a person is obviously not sufficient, because there may be many people with the same name. In the United States, the social-security number attribute of a person would be a candidate key. Since non-U.S. residents usually do not have social-security numbers, international enterprises must generate their own unique identifiers.

An alternative is to use some unique combination of other attributes as a key. The primary key should be chosen such that its attribute values are never, or very rarely, changed. For instance, the address field of a person should not be part of the primary key, since it is likely to change. Social-security numbers, on the other hand, are guaranteed never to change. Unique identifiers generated by enterprises generally do not change, except if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique.

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined. A relation, say $r1$, may include among its attributes the primary key of another relation, say $r2$. This attribute is called a **foreign key** from $r1$, referencing $r2$.

The relation $r1$ is also called the **referencing relation** of the foreign key dependency, and $r2$ is called the **referenced relation** of the foreign key. For

NOTES

example, the attribute *dept name* in *Faculty* is a foreign key from *Faculty*, referencing *department*, since *dept name* is the primary key of *department*. In any database instance, given any tuple, say *ta*, from the *Faculty* relation, there must be some tuple, say *tb*, in the *department* relation such that the value of the *dept name* attribute of *ta* is the same as the value of the primary key, *dept name*, of *tb*.

Now consider the *section* and *teaches* relations. It would be reasonable to require that if a section exists for a course, it must be taught by at least one Faculty; however, it could possibly be taught by more than one Faculty. To enforce this constraint, we would require that if a particular (*course id*, *sec id*, *semester*, *year*) combination appears in *section*, then the same combination must appear in *teaches*. However, this set of values does not form a primary key for *teaches*, since more than one Faculty may teach one such section. As a result, we cannot declare a foreign key constraint from *section* to *teaches* (although we can define a foreign key constraint in the other direction, from *teaches* to *section*).

The constraint from *section* to *teaches* is an example of a **referential integrity constraint**; a referential integrity constraint requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.

4.5 Enforcing Integrity constraints

Data integrity refers to the correctness and completeness of data within a database. To enforce data integrity, you can constrain or restrict the data values that users can insert, delete, or update in the database. These mechanisms allow you to maintain these types of data integrity:

- Requirement – requires that a table column must contain a valid value in every row; it cannot allow null values. The **create table** statement allows you to restrict null values for a column.
- Check or validity – limits or restricts the data values inserted into a table column. You can use triggers or rules to enforce this type of integrity.
- Uniqueness – no two table rows can have the same non-null values for one or more table columns. You can use indexes to enforce this integrity.
- Referential – data inserted into a table column must already have matching data in another table column or another column in the same table. A single table can have up to 192 references.

4.6 Querying Relational Data

A query is a request for data or information from a database table or combination of tables. This data may be generated as results returned by

Structured Query Language (SQL) or as pictorials, graphs or complex results, e.g., trend analyses from data-mining tools.

One of several different query languages may be used to perform a range of simple to complex database queries. SQL, the most well-known and widely-used query language, is familiar to most database administrators (DBAs).

4.7 Logical Database Design

4.7.1 Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and the **database instance**, which is a snapshot of the data in the database at a given instant in time. The concept of a relation corresponds to the programming-language notion of a variable, while the concept of a **relation schema** corresponds to the programming-language notion of type definition.

In general, a relation schema consists of a list of attributes and their corresponding domains. The concept of a relation instance corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time;

Table 4.5 Department Relation

Dept_name	Campus	No. of Rooms
Comp. Sci.	Science	7
Maths	Science	5
Tamil	Arts	4
Physics	Science	9
English	Arts	5
Bio-technology	Science	6
Physical Education	Education	8
Social Works	Arts	4

Similarly, the contents of a relation instance may change with time as the relation is updated. In contrast, the schema of a relation does not generally change. Although it is important to know the difference between a relation schema and a relation instance, we often use the same name, such as *Faculty*, to refer to both the schema and the instance. Where required, we explicitly refer to the schema or to the instance, for example “the *Faculty* schema,” or “an instance of the *Faculty* relation.” However, where it is clear whether we mean the schema or the instance, we simply use the relation name.

Consider the *department* relation of Table 4.5. The schema for that relation is

***department* (*dept*, *campus*, *No. of Rooms*)**

NOTES

Note that the attribute *dept* appears in both the *Faculty* schema and the *department* schema. This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations.

For example, suppose we wish to find the information about all the Faculty members who work in the Science Campus. We look first at the *department* relation to find the *dept name* of all the departments housed in Science campus. Then, for each such department, we look in the *Faculty* relation to find the information about the Faculty associated with the corresponding *dept name*.

Let us continue with our university database example. Each course in a university may be offered multiple times, across different semesters, or even within a semester. We need a relation to describe each individual offering, or section, of the class. The schema is

***section* (*course id*, *semester*, *year*, *dept*)**

Table 4.6 Section Relation

Course_id	Semster	Year	Dept
CS-101	1	I	Comp. Sci.
MA-101	2	I	Maths
TA-102	1	I	Tamil
TA-204	3	II	Tamil
EN-101	1	I	English
MA-205	4	II	Maths
CS-204	4	II	Comp. Sci.

Table 4.6 shows a sample instance of the *section* relation. We need a relation to describe the association between Faculty and the class sections that they teach. The relation schema to describe this association is

***teaches* (*fac_id*, *course id*, *semester*, *dept*)**

Table 4.7 Teaches Relation

Fac_id	Course_id	Semster	Dept
F101	CS-101	1	Comp. Sci.
F306	MA-101	2	Maths
F204	TA-102	1	Tamil
F206	TA-204	3	Tamil
F401	EN-101	1	English
F305	MA-205	4	Maths
F103	CS-204	4	Comp. Sci.

Table 4.7 shows a sample instance of the *teaches* relation. As you can imagine, there are many more relations maintained in a real university database. In addition to those relations we have listed already, *Faculty*,

department, *course*, *section*, *prereq*, and *teaches*, we use the following relations in this text:

- *student* (*ID*, *name*, *dept name*, *tot cred*)
- *advisor* (*s id*, *i id*)
- *takes* (*ID*, *course id*, *sec id*, *semester*, *year*, *grade*)
- *classroom* (*building*, *room number*, *capacity*)
- *time slot* (*time slot id*, *day*, *start time*, *end time*)

4.7.2 Schema Diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams**. Figure 4.1 shows the schema diagram for our university organization. Each relation appears as a box, with the relation name at the top in blue, and the attributes listed inside the box. Primary key attributes are shown underlined. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

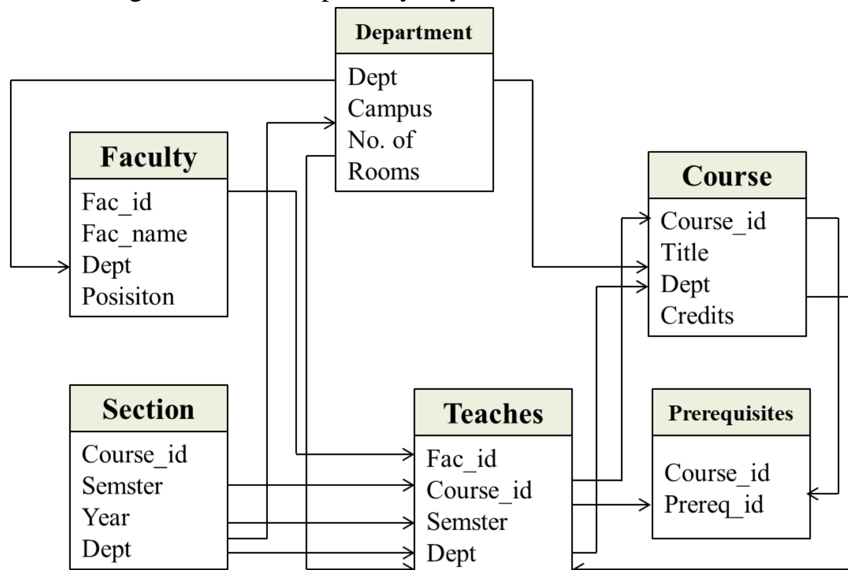


Figure 4.1 Schema Diagram for University Database

Referential integrity constraints other than foreign key constraints are not shown explicitly in schema diagrams. We will study a different diagrammatic representation called the entity-relationship diagram

4.8 Introduction to Views

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

NOTES

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following −

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

4.8.1 Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** syntax is as follows −

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

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example

Consider the CUSTOMERS table having the following schema

Customers (cust_id, cust_name, age, address, mobile)

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS SELECT name, age
FROM  CUSTOMERS;
```

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

4.8.2 Views with Check Option

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following code block has an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION.

```
CREATE VIEW CUSTOMERS_VIEW AS SELECT name, age
FROM  CUSTOMERS WHERE age IS NOT NULL WITH CHECK
OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

4.8.3. Updating a View

A view can be updated under certain conditions which are given below −

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Ramesh.

```
SQL > UPDATE CUSTOMERS_VIEW SET AGE = 35 WHERE
name = 'Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

4.9 Destroying / altering Tables and Views

4.9.1 Altering the Tables

The SQL ALTER TABLE command is used to add, delete or modify columns in an existing table. You should also use the ALTER TABLE command to add and drop various constraints on an existing table.

Syntax

The basic syntax of an ALTER TABLE command to add a New Column in an existing table is as follows.

```
ALTER TABLE table_name ADD column_name datatype;
```

NOTES

The basic syntax of an ALTER TABLE command to DROP COLUMN in an existing table is as follows.

ALTER TABLE table_name DROP COLUMN column_name;

The basic syntax of an ALTER TABLE command to change the DATA TYPE of a column in a table is as follows.

ALTER TABLE table_name MODIFY COLUMN column_name datatype;

The basic syntax of an ALTER TABLE command to add a NOT NULL constraint to a column in a table is as follows.

ALTER TABLE table_name MODIFY column_name datatype NOT NULL;

The basic syntax of ALTER TABLE to ADD UNIQUE CONSTRAINT to a table is as follows.

ALTER TABLE table_name ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);

The basic syntax of an ALTER TABLE command to ADD CHECK CONSTRAINT to a table is as follows.

ALTER TABLE table_name ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);

The basic syntax of an ALTER TABLE command to ADD PRIMARY KEY constraint to a table is as follows.

ALTER TABLE table_name ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);

The basic syntax of an ALTER TABLE command to DROP CONSTRAINT from a table is as follows.

ALTER TABLE table_name DROP CONSTRAINT MyUniqueConstraint;

Check Your Progress

1. What is domain integrity?
2. What is meant by Querying relational Data?
3. Define: Views

4.9.2 Deleting Rows from a Table / View

Rows of data can be deleted from a table / view.

Following is an example to delete all records in both tables / views.

SQL > DELETE FROM CUSTOMERS;

SQL > DELETE FROM CUSTOMERS_VIEW;

The WHERE clause can be used to delete a record from table / view with conditions. Following is an example to delete a record having AGE = 22 in both tables / views.

SQL > DELETE FROM CUSTOMERS WHERE age = 22;

SQL > DELETE FROM CUSTOMERS_VIEW WHERE age = 22;

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself.

4.9. 3. Dropping Tables / Views

The syntax to drop a table / view are given below −

DROP TABLE table_name;

DROP VIEW view_name;

Following is an example to drop the CUSTOMERS_VIEW and CUSTOMERS table.

DROP TABLE CUSTOMERS;

DROP VIEW CUSTOMERS_VIEW;

4.10 Answers to Check Your Progress Questions

1. Domain integrity means the definition of a valid set of values for an attribute.
2. A query is a request for data or information from a database table or combination of tables. This data may be generated as results returned by Structured Query Language (SQL) or as pictorials, graphs or complex results, e.g., trend analyses from data-mining tools.
3. A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

4.11 Summary

- A **relational database** consists of a collection of tables each of which is assigned a unique name.
- There is a set of permitted values, called the domain of that attribute.
- **Database integrity** refers to the validity and consistency of stored data.
- A superkey is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation.
- Minimal superkeys are called **candidate keys**.
- Term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation.

- **Data integrity** refers to the correctness and completeness of data within a database.
- The create table statement allows you to restrict null values for a column.
- A query is a request for data or information from a database table or combination of tables.
- The SQL ALTER TABLE command is used to add, delete or modify columns in an existing table.
- A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams**.

4.12 Keywords

- Collection of tables having unique name is known as **relational database**.
- The term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row.
- The minimal set of attribute which can uniquely identify a tuple is known as **candidate key.**
- The correctness and completeness of data within a database is known as **Data integrity**.

4.13 Self-Assessment Questions and Exercises

Short Answer Questions:

1. Define: Relational Database
2. Define: Attribute.
3. Name the types of Database integrity constraint?
4. What is Super Key?
5. Write the syntax of Alter table.

Long Answer Questions:

1. Describe the structure of Relational Model.
2. Write short-notes on the following:
    a. Candidate key.
    b. Super key
    c. Primary key
    d. Foreign key
3. Write short-notes on the following:
    a. View table
    b. Altering table
    c. Destroying table

4.14 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011.

UNIT V RELATIONAL ALGEBRA

Structure

5.1 Introduction
5.2 Objectives
5.3 Introduction to Relational Algebra
5.4 Selection and projection set operations
5.5 Renaming
5.6 Joins
5.7 Division
5.8 Examples of Algebra overviews
5.9 Answers to Check Your Progress Questions
5.10 Summary
5.11 Key Words
5.12 Self-Assessment Questions and Exercises
5.13 Further Readings

5.1 Introduction

Relational database systems are expected to be equipped with a query language that can assist its users to query the database instances. There are two kinds of query languages − relational algebra and relational calculus. Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either **unary** or **binary**. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

5.2 Objectives

After reading this chapter, you will be able to understand:

- Relational algebra
- Operations using relational algebra
- Joins
- Division

5.3 Introduction to Relational Algebra

Relational algebra is a widely used procedural query language. It collects instances of relations as input and gives occurrences of relations as output. It uses various operations to perform this action. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) relation instances as arguments and returns a relation instance as the result. This property makes it easy to compose operators to form a complex query—a relational algebra

expression is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions.

The following sections describe the basic operators of the algebra (selection, projection, union, cross-product, and difference), as well as some additional operators that can be defined in terms of the basic operators but arise frequently enough to warrant special attention, in the following sections.

Each relational query describes a step-by-step procedure for computing the desired answer, based on the order in which operators are applied in the query. The procedural nature of the algebra allows us to think of an algebra expression as a recipe, or a plan, for evaluating a query, and relational systems in fact use algebra expressions to represent query evaluation plans.

To explain the relational algebraic operations, the following sections uses the below mentioned schemes of on-line hotel booking portal are used.

Customer (cid, cname, mobile, city)
Hotel (hid, hname, rate)
Reserves (cid, hid, day)

The customer relation having the attributes to store the details of customers of the portal with cid is the key field. The Hotel relation has some fields with hid is the key field. Whereas, for Reserves relation, cid and hid fields are key fields.

Table 5.1 a. 'Customer' Relation

cid	Cname	mobile	City
C1001	B. Raju	8976894567	Chennai
C2546	M. Sunil	8876921234	Salem
C1456	C. Kunal	9444324578	Chennai
C1324	K. Kamal	9443478902	Madurai
C4578	K. Chitra	9897452123	Chennai
C3456	S. Bala	9789012134	Karaikudi

Table 5.1 b. 'Hotel' Relation

hid	hname	Rate
H101	The Conclave	5000.00
H124	Heritage Inn	6750.00
H456	The Holidays	4300.00

Table 5.1 c. 'Reserves' Relation

cid	hid	Day
C1001	H101	8.9.2019
C1456	H456	17.8.2019
C4578	H124	4.9.2019
C1324	H101	6.8.2019
C1001	H101	7.10.2019
C2456	H456	14.9.2019
C1456	H124	28.8.2019

Basic Relational Algebra Operations:

Relational Algebra divided in various groups

Unary Relational Operations
- SELECT (symbol: σ)
- PROJECT (symbol: π)
- RENAME (symbol: *ρ*)

Relational Algebra Operations From Set Theory
1. UNION (U)
2. INTERSECTION (∩),
3. DIFFERENCE (-)
4. CARTESIAN PRODUCT (x)

Binary Relational Operations
- JOIN
- DIVISION

5.4 Selection and projection set operations

The SELECT operation is used for selecting a subset of the tuples according to a given selection condition. Sigma(σ)Symbol denotes it. It is used as an expression to choose tuples which meet the selection condition. Select operation selects tuples that satisfy a given predicate.

$$\sigma_p(r)$$

σ is the predicate
r stands for relation which is the name of the table
p is prepositional logic

The projection eliminates all attributes of the input relation but those mentioned in the projection list. The projection method defines a relation that contains a vertical subset of Relation.

This helps to extract the values of specified attributes to eliminate duplicate values. (pi) The symbol used to choose attributes from a relation. This operation helps you to keep specific columns from a relation and discards the other columns.

$$\pi_{\text{fields}}(r)$$

π is the predicate
r stands for relation which is the name of the table
fields are the attributes of the relation

These operations allow us to manipulate data in a single relation. Consider the Customer relation. We can retrieve rows corresponding to customers from a particular city by using the *σ* operator. The expression, $\sigma_{city='Chennai'}(Customer)$ evaluates to the relation as shown in Table 5.2. The subscript *city='Chennai'* specifies the selection criterion to be applied while retrieving tuples.

NOTES

Table 5.2 Selection {$\sigma_{city='Chennai'}(Customer)$}

cid	Cname	Mobile	City
C1001	B. Raju	8976894567	Chennai
C1456	C. Kunal	9444324578	Chennai
C4578	K. Chitra	9897452123	Chennai

The selection operator σ specifies the tuples to retain through a *selection condition*. In general, the selection condition is a Boolean combination (i.e., an expression using the logical connectives ∧ and ∨) of *terms* that have the form *attribute* op *constant* or *attribute1* op *attribute2*, where op is one of the comparison operators <, <=, =, =, >=, or >. The reference to an attribute can be by position (of the form *.i* or *i*) or by name (of the form *.name* or *name*). The schema of the result of a selection is the schema of the input relation instance.

The projection operator π allows us to extract columns from a relation; for example, we can find out all hotel names and rate by using π. The expression $\pi_{hname,rate}(Hotel)$ as shown in Table 5.3.

Table 5.3 Projection {$\pi_{hname,rate}(Hotel)$}

Hname	Rate
The Conclave	5000.00
Heritage Inn	6750.00
The Holidays	4300.00

Suppose that we wanted to find out only the details (hid and hname) of the hotels with Rate is less than Rs. 6000/-. The expression is

$$\pi_{hid,hname}(\sigma_{rate<6000.00}(Hotel))$$

Figure 5.4 Result of { $\pi_{hid,hname}(\sigma_{rate<6000.00}(Hotel))$ }

hid	hname
H101	The Conclave
H456	The Holidays

Set Operations

The following relations given in 5.5.a Hostel, 5.5.b Student, 5.5.c Mess and 5.5.d All Mess are used to illustrate the set operators:

Table 5.5.a 'Hostel' Relation

Room_No	Name	Address	Phone	Age
1	RAM	CHENNAI	9455123451	18
5	NARESH	MADURAI	9782918192	22
6	SWETA	CHENNAI	9852617621	21
4	SURESH	KARAIKUDI	9156768971	18

Table 5.5.a 'Student' Relation

Stud_No	Name	Address	Phone	Age
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

Table 5.5.c 'Mess' Relation

Room_No	Mess
1	VEG
5	NON-VEG
6	VEG
4	NON-VEG
5	VEG

Table 5.5.d 'All Mess' Relation

Mess
VEG
NON-VEG

The following standard operations on sets are also available in relational algebra: *union* (*U*), *intersection* (∩), *set-difference* (-), and *cross-product* (×).

- Union: *R U S* returns a relation instance containing all tuples that occur in *either* relation instance *R* or relation instance *S* (or both). *R* and *S* must be *unioncompatible*, and the schema of the result is defined to be identical to the schema of *R*.
- Intersection: *R* ∩ *S* returns a relation instance containing all tuples that occur in *both R* and *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.
- Set-difference: *R* - *S* returns a relation instance containing all tuples that occur in *R* but not in *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.
- Cross-product: *R* × *S* returns a relation instance whose schema contains all the fields of *R* (in the same order as they appear in *R*) followed by all the fields of *S* (in the same order as they appear in *S*). The result of *R* × *S* contains one tuple ⟨*r, s*⟩ (the concatenation of tuples *r* and *s*) for each pair of tuples *r* ∈ *R, s* ∈ *S*. The cross-product operation is sometimes called Cartesian product.

We now illustrate these definitions through several examples.
The union of *Hostel(S1)* and *Student(S2)* is shown in Table 5.6.a Fields are listed in order; field names are also inherited from *S*1. In general, fields of *S*2 may have different names; recall that we require only domains to match. Note that the result is a *set* of tuples. Tuples that appear in both *S*1 and *S*2 appear only once in *S*1 ∪ *S*2. Also, *Student* ∪ *Mess* is not a valid operation because the two relations are not union-compatible. The intersection of *S*1 and *S*2 is shown in Table 5.6.b, and the set-difference *S*1 - *S*2 is shown in Table 5.6.c

NOTES

Table 5.6.a Result of Union Operation 'Hostel U Student'

Room_No	Name	Address	Phone	Age
1	RAM	CHENNAI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	KARAIKUDI	9156768971	18
5	NARESH	MADURAI	9782918192	22
6	SWETA	CHENNAI	9852617621	21

Table 5.6.b Result of Intersection Operation 'Hostel ∩ Student'

Room_No	Name	Address	Phone	Age
1	RAM	CHENNAI	9455123451	18
4	SURESH	KARAIKUDI	9156768971	18

Table 5.6.c Result of Set Difference Operation 'Hostel - Student'

Room_No	Name	Address	Phone	Age
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20

The result of the cross-product *Hostel (S1)* × *Mess (R1)* is shown in Table 5.7. The fields in 1 × *R*1 have the same domains as the corresponding fields in *R*1 and *S*1. In Table 5.7, *Room_No* is listed in parentheses to emphasize that it is not an inherited field name; only the corresponding domain is inherited.

Table 5.7 Result of Cross-product Operation '*Hostel X Mess*'

(Room_No)	Name	Address	Phone	Age	(Room_No)	Mess
1	RAM	CHENNAI	9455123451	18	1	VEG
1	RAM	CHENNAI	9455123451	18	5	NON-VEG
1	RAM	CHENNAI	9455123451	18	6	VEG
1	RAM	CHENNAI	9455123451	18	4	NON-VEG
5	NARESH	MADURAI	9782918192	22	1	VEG
5	NARESH	MADURAI	9782918192	22	5	NON-VEG
5	NARESH	MADURAI	9782918192	22	6	VEG
5	NARESH	MADURAI	9782918192	22	4	NON-VEG
6	SWETA	CHENNAI	9852617621	21	1	VEG
6	SWETA	CHENNAI	9852617621	21	5	NON-VEG
6	SWETA	CHENNAI	9852617621	21	6	VEG
6	SWETA	CHENNAI	9852617621	21	4	NON-VEG
4	SURESH	KARAIKUDI	9156768971	18	1	VEG
4	SURESH	KARAIKUDI	9156768971	18	5	NON-VEG
4	SURESH	KARAIKUDI	9156768971	18	6	VEG
4	SURESH	KARAIKUDI	9156768971	18	4	NON-VEG

5.5 Renaming

We introduce a renaming operator ρ for this purpose. The expression $\rho(R\ (F), E)$ takes an arbitrary relational algebra expression E and returns an instance of a (new) relation called R. R contains the same tuples as the result of E, and has the same schema as E, but some fields are renamed. The field names in relation R are the same as in E, except for fields renamed in the *renaming list* F.

To rename STUDENT relation to STUDENT1, we can use rename operator like:

ρ(STUDENT1, STUDENT)

If you want to create a relation STUDENT_NAMES with STUD_NO and NAME from STUDENT, it can be done using rename operator as:

$$\rho\ (\text{STUDENT_NAMES}, \pi_{(\text{STUD_NO, NAME})}(\text{STUDENT}))$$

It is customary to include some additional operators in the algebra, but they can all be defined in terms of the operators that we have defined thus far. (In fact, the renaming operator is only needed for syntactic convenience, and even the ∩ operator is redundant; $R \cap S$ can be defined as $R - (R - S)$.)

We will consider these additional operators, and their definition in terms of the basic operators, in the next two subsections.

5.6 Joins

The *join* operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations. Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-products. Joins have received a lot of attention, and there are several variants of the join operation.

Conditional Joins The most general version of the join operation accepts a *join condition c* and a pair of relation instances as arguments, and returns a relation instance. The *join condition* is identical to a *selection condition* in form. The operation is defined as follows:

For example: Select students whose Room_NO is greater than Stud_NO of Hostel

$$HOSTEL \bowtie_{c\ HOSTEL.Room_NO > STUDENT.Stud_NO} STUDENT$$

NOTES

Thus ⊲⊳ is defined to be a cross-product followed by a selection. In terms of basic operators (cross product and selection):

$$\sigma_{(HOSTEL.Room_NO > STUDENT.Stud_NO)}(HOSTEL\ X\ STUDENT)$$

Table 5.8 Conditional Join

Room No	Name	Address	Phone	Age	Stud No	Name	Address	Phone	Age
2	RAMESH	GURGAON	9652431543	18	1	RAM	CHENNAI	9455123451	18
3	SUJIT	ROHTAK	9156253131	20	1	RAM	CHENNAI	9455123451	18
4	SURESH	KARAIKUDI	9156768971	18	1	RAM	CHENNAI	9455123451	18

Natural Join(⋈): It is a special case of equijoin in which equality condition hold on all attributes which have same name in relations R and S (relations on which join operation is applied). While applying natural join on two relations, there is no need to write equality condition explicitly. Natural Join will also return the similar attributes only once as their value will be same in resulting relation.

Example: Select students whose Stud_NO is equal to Room_NO of Mess as:

STUDENT ⋈ MESS

Table 5.9 Natural Join

Stud_No	Name	Address	Phone	Age	Mess
1	RAM	DELHI	9455123451	18	VEG
4	SURESH	DELHI	9156768971	18	NON-VEG

Natural Join is by default inner join because the tuples which does not satisfy the conditions of join does not appear in result set.

Equijoin

Equijoin is a **special case of conditional join** where only equality condition holds between a pair of attributes. As values of two attributes will be equal in result of equijoin, only one attribute will be appeared in result.

The schema of the result of an equijoin contains the fields of *R* (with the same names and domains as in *R*) followed by the fields of *S* that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from *R* and *S*, they are unnamed in the result relation.

Example: Select students whose Stud_NO is equal to Room_NO of Mess Table

$$\textbf{STUDENT} \bowtie_{\textbf{STUDENT.Stud_NO=MESS.Room_NO}} \textbf{MESS}$$

Table 5.10 Equijoin

Stud_ No	Name	Address	Phone	Age	Room _No	Mess
1	RAM	DELHI	9455123451	18	1	VEG
4	SURESH	DELHI	9156768971	18	4	NON-VEG

5.7 Division

Division operator A÷B can be applied if and only if:

- Attributes of B is proper subset of Attributes of A.
- The relation returned by division operator will have attributes = (All attributes of A – All Attributes of B)
- The relation returned by division operator will return those tuples from relation A which are associated to every B's tuple.

Consider the relation MESS and ALL_MESS as given in Table 5.5.c and Table 5.5.d above.

To apply division operator as

MESS ÷ ALL_MESS

- The operation is valid as attributes in ALL_MESS is a proper subset of attributes in MESS.
- The attributes in resulting relation will have attributes {Room_NO,Mess}-{Mess}=Room_NO
- The tuples in resulting relation will have those Room_NO which are associated with all B's tuple {VEG, NON-VEG}. Room_NO 5 is associated to all tuples of B. So the resulting relation will be:

Table 5.11 Division

Room_No
5

Check Your Progress

1. Define: Relational Algebra
2. What is meant by Join?
3. What is called as Equijoin?

5.8 Examples of Algebra overviews

One More Example of Relational Algebra Queries

(Q) Find the details of customers who have reserved The Hotel Conclave.

This query can be written as follows:

$$\pi_{cid}(\sigma_{hid = 'H101'}(Reserves)) \bowtie CUSTOMER$$

First, we execute the selection operation i.e. $\sigma_{hid = 'H101'}(Reserves)$, then we project the values of cid from the Reserves table. Finally, we perform join operation with Customer relation based on the results of the above operations.

Table 5.12.a Result of Selection operation

cid	Hid	day
C1001	H101	8.9.2019
C1324	H101	6.8.2019
C1001	H101	7.10.2019

Table 5.12.b Result of Projection operation

cid
C1001
C1324
C1001

Table 5.12.c Final Result of the query

cid	Cname	mobile	City
C1001	B. Raju	8976894567	Chennai
C1324	K. Kamal	9443478902	Madurai
C1001	B. Raju	8976894567	Chennai

5.9 Answers to Check Your Progress Questions

1. Relational algebra is a widely used procedural query language. It collects instances of relations as input and gives occurrences of relations as output. It uses various operations to perform this action.
2. The join operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations
3. Equijoin is a special case of conditional join where only equality condition holds between a pair of attributes. As values of two attributes will be equal in result of equijoin, only one attribute will be appeared in result.

5.10 Summary

- Relational algebra is a widely used procedural query language.
- Queries in algebra are composed using a collection of operators.
- The SELECT operation is used for selecting a subset of the tuples according to a given selection condition.
- Sigma(s) Symbol denotes to choose tuples which meet the selection condition.
- The cross-product operation is sometimes called Cartesian product.
- Equijoin is a special case of conditional join where only equality condition holds between a pair of attributes.
- The standard operations on sets are also available in relational algebra like union (U), intersection (n), set-difference (-), and cross-product (×).
- The join operation is the most commonly used way to combine information from two or more relations.

5.11 Keywords

- The relational algebra is a **procedural** query language.
- The **relational algebra** provides a set of operations that take one or more relations as input and return a relation as an output.
- Practical **query languages** such as **SQL** are based on the relational algebra
- The select, project, and rename operations are called **unary** operations
- The **select** operation selects tuples that satisfy a given predicate.
- The **project** operation allows us to produce the relation.

5.12 Self-Assessment Questions and Exercises

Short Answer Questions:

1. Define: Relational algebra
2. What are the relational algebra operations?
3. What are the Unary relation operations?
4. What is mean by Equijoin?
5. What are all the set operations?

Long Answer Questions:

1. Describe the characteristics of various Unary Relational Operations
2. Write short-notes on the following:
    a. UNION (U)
    b. INTERSECTION (∩)
    c. DIFFERENCE (-)
    d. CARTESIAN PRODUCT (x)
3. Write an example of Relational Algebra Queries

5.13 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011.

UNIT VI Relational Calculus

Structure

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

6.1 Introduction

Relational Calculus is a **non-procedural query language** unlike relational algebra. Tuple Calculus provides only the description of the query but it does not provide the methods to solve it. Thus, it explains what to do but not how to do. Relational calculus exists in two forms and those are mentioned below:

- Tuple relational calculus
- Domain relational calculus

6.2 Objectives

This chapter imparts you the fundamentals of:

- Tuple Relational Calculus
- Domain Relational Calculus

6.3 Relational Calculus

Relational calculus is a query language which is non-procedural, and instead of algebra, it uses mathematical predicate calculus. The relational calculus is not the same as that of differential and integral calculus in mathematics but takes its name from a branch of symbolic logic termed as predicate calculus. When applied to databases, it is found in two forms. These are

- Tuple relational calculus which was originally proposed by Codd in the year 1972 and

- Domain relational calculus which was proposed by Lacroix and

Pirotte in the year 1977

In first-order logic or predicate calculus, a predicate is a truth-valued function with arguments. When we replace with values for the arguments, the function yields an expression, called a *proposition*, which will be either true or false.

6.4 Tuple Relational Calculus

Tuple Relational Calculus

A tuple variable is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields. A tuple relational calculus query has the form *{ T | p(T) }*, where *T* is a tuple variable and *p*(*T*) denotes a *formula* that describes *T* ; we will shortly define formulas and queries rigorously. The result of this query is the set of all tuples *t* for which the formula *p*(*T*) evaluates to true with *T* = *t*. The language for writing formulas *p*(*T*) is thus at the heart of TRC and is essentially a simple subset of *first-order logic*.

Syntax of TRC Queries

We now define these concepts formally, beginning with the notion of a formula. Let *Rel* be a relation name, *R* and *S* be tuple variables, *a* an attribute of *R*, and *b* an attribute of *S*. Let op denote an operator in the set $\{<,>,=,\leq, \geq, =\}$.
An atomic formula is one of the following:

a. *R* E *Rel*
b. *R.a* op *S.b*
c. *R.a* op *constant*, or *constant* op *R.a*

A formula is recursively defined to be one of the following, where *p* and *q* are themselves formulas, and *p*(*R*) denotes a formula in which the variable *R* appears: any atomic formula

¬ *p*, *p V q*, *p ^ q*, or *p* ⇒ *q*
∃ *R*(*p*(*R*)), where *R* is a tuple variable
∀ *R*(*p*(*R*)), where *R* is a tuple variable

We observe that every variable in a TRC formula appears in a sub-formula that is atomic, and every relation schema specifies a domain for each field; this observation ensures that each variable in a TRC formula has a well-defined domain from which values for the variable are drawn. That is, each variable has a well-defined *type*, in the programming language sense. Informally, an atomic formula *R* ∈ *Rel* gives *R* the type of tuples in *Rel*, and comparisons such as *R.a* op *S.b* and *R.a* op *constant* induce type restrictions on the field *R.a*. If a variable R does not appear in an atomic formula of the

form *R E Rel* (i.e., it appears only in atomic formulas that are comparisons), we will follow the convention that the type of R is a tuple whose fields include all (and only) fields of R that appear in the formula.

We will not define types of variables formally, but the type of a variable should be clear in most cases, and the important point to note is that comparisons of values having different types should always fail. (In discussions of relational calculus, the simplifying assumption is often made that there is a single domain of constants and that this is the domain associated with each field of each relation.)

A TRC query is defined to be expression of the form *{T / p(T)}*, where *T* is the only free variable in the formula *p*.

Semantics of TRC Queries

What does a TRC query mean? More precisely, what is the set of answer tuples for a given TRC query? The answer to a TRC query *{T / p(T)}*, as we noted earlier, is the set of all tuples *t* for which the formula *p*(*T*) evaluates to true with variable *T* assigned the tuple value *t*. To complete this definition, we must state which assignments of tuple values to the free variables in a formula make the formula evaluate to true.

A query is evaluated on a given instance of the database. Let each free variable in a formula *F* be bound to a tuple value. For the given assignment of tuples to variables, with respect to the given database instance, *F* evaluates to (or simply 'is') true if one of the following holds:

F is an atomic formula $R \in Rel$, and *R* is assigned a tuple in the instance of relation *Rel*.

- *F* is a comparison *R.a* op *S.b*, *R.a* op *constant*, or *constant* op *R.a*, and the tuples assigned to *R* and *S* have field values *R.a* and *S.b* that make the comparison true.
- *F* is of the form ¬*p*, and *p* is not true; or of the form *p* ^ *q*, and both *p* and *q* are true; or of the form *p V q*, and one of them is true, or of the form *p* ⇒ *q* and *q* is true whenever4 *p* is true.
- *F* is of the form ∃*R*(*p*(*R*)), and there is some assignment of tuples to the free variables in *p*(*R*), including the variable *R*,5 that makes the formula *p*(*R*) true.
- *F* is of the form ∀*R*(*p*(*R*)), and there is some assignment of tuples to the free variables in *p*(*R*) that makes the formula *p*(*R*) true no matter what tuple is assigned to *R*.

We now examine the following examples of TRC. For this purpose, the following relation (as same in Table 5.1.a) is used.

Table 6.1 'Customer' Relation

cid	Cname	Mobile	city
C1001	B. Raju	8976894567	Chennai
C2546	M. Sunil	8876921234	Salem
C1456	C. Kunal	9444324578	Chennai
C4578	K. Chitra	9897452123	Chennai
C3456	S. Bala	9789012134	Karaikudi

For example, to specify the range of a tuple variable C as the Customer relation, we write:

Customer (C)

To express the query 'Find the set of all tuples C such that F(C) is true,' we can write:

{C | F(C)}

Here, F is called a formula (well-formed formula, or wff in mathematical logic). For example, to express the query 'Find the cid, cname, mobile and city of all customers who are from Chennai', we can write:

{C | Customer (C) ∧ city = 'Chennai'}

6.5 Domain Relational Calculus

A domain variable is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers). A DRC query has the form $\{\ \langle x1, x2, \ldots, xn\rangle\ |\ p(\ \langle x1,x2,\ldots, xn\rangle\ )\}$, where each *xi* is either a *domain variable* or a constant and $p(\ \langle x1,x2,\ldots, xn\rangle\ )$ denotes a DRC formula whose only free variables are the variables among the *xi*, $1 \leq i \leq n$. The result of this query is the set of all tuples $\langle x1, x2,\ldots,xn\rangle$ for which the formula evaluates to true.

A DRC formula is defined in a manner that is very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables. Let op denote an operator in the set $\{<, >, =, \leq, \geq, =\}$ and let *X* and *Y* be domain variables. An atomic formula in DRC is one of the following:

$\langle x1,x2,\ldots,xn\rangle \in Rel$, where *Rel* is a relation with *n* attributes; each *xi*, $1 \leq i \leq n$ is either a variable or a constant.

X op *Y*

X op *constant*, or *constant* op *X*

A formula is recursively defined to be one of the following, where *p* and *q* are themselves formulas, and *p*(*X*) denotes a formula in which the variable *X* appears:

any atomic formula

$\neg p$, $p \vee q$, $p \wedge q$, or $p \Rightarrow q$

$\exists X(p(X))$, where X is a domain variable

$\forall X (p(X))$, where X is a domain variable

Check Your Progress

1. Define : Relational calculus
2. Difference between DRC and TRC
3. How to express TRC Terms?

6.6 Expressive Power of Algebra and Calculus

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or *declarative*, in that it allows us to describe the set of answers without being explicit about how they should be computed. Relational calculus has had a big influence on the design of commercial query languages such as SQL and, especially, Query-by-Example (QBE).

The variant of the calculus that we present in detail is called the tuple relational calculus (TRC). Variables in TRC take on tuples as values. In another variant, called the domain relational calculus (DRC), the variables range over field values. TRC has had more of an influence on SQL, while DRC has strongly influenced QBE

6.7 Answers to Check Your Progress Questions

1. Relational calculus is a query language which is non-procedural, and instead of algebra, it uses mathematical predicate calculus.
2. A DRC formula is defined in a manner that is very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables.
3. A TRC query is defined to be expression of the form {T | p(T)}, where T is the only free variable in the formula p.

6.8 Summary

- The tuple **relational calculus** and **domain relational calculus** are nonprocedural.
- The relational calculus uses predicate logic to define the result desired without giving any specific **algebraic** procedure for obtaining that result
- A **DRC** formula is defined in a manner that is very similar to the definition of a **TRC** formula.
- **Relational calculus** is an alternative to **relational algebra**.
- The variant of the calculus that we present in detail is called the **tuple relational calculus** (TRC).

6.9 Keywords

- **Relational calculus** is a query language which is non-procedural
- A **TRC** query is defined to be expression of the form {T | p(T)}.
- Relational calculus has had a big influence on the design of commercial query languages such as **SQL** and, especially, **Query-by-Example** (QBE).

6.10 Self-Assessment Questions and Exercises

Short Answer Questions:

1. Define: Relational Calculus
2. What is TRC?
3. State the syntax of TRC Queries.
4. What is DRC?

Long Answer Questions:

1. Explain the concepts of TRC and DRC

6.11 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011.

BLOCK – III

SQL QUERY

UNIT VII FORM OF BASIC SQL QUERY

Structure

7.1 Introduction
7.2 Objectives
7.3 Basics of SQL Queries
7.4 Examples of Basic SQL Queries
7.5 Introduction to Nested Queries
7.6 Correlated Nested Queries Set
7.7 Comparison Operators
7.8 Aggregative Operators
7.9 NULL Values
7.10 Comparison using NULL Values
7.11 Logical connectivity's AND, OR and NOT
7.12 Outer Join
7.13 Disallowing NULL Values
7.14 PL/SQL
7.15 Complex Integrity Constraints in SQL Triggers and Active Databases
7.16 Answers to Check Your Progress Questions
7.17 Summary
7.18 Key Words
7.19 Self-Assessment Questions and Exercises
7.20 Further Readings

7.1 Introduction

SQL stands for Structured Query Language. SQL is a standard language for accessing and manipulating databases. Structure Query Language (SQL) is a database query language used for storing and managing data in Relational DBMS. SQL was the first commercial language introduced for E.F Codd's **Relational** model of database. Today almost all RDBMS (MySql, Oracle, Infomix, Sybase, MS Access) use **SQL** as the standard database query language. SQL is used to perform all types of data operations in RDBMS.

7.2 Objectives

After reading this chapter, you will understand the:

- Fundamentals of SQL
- Query formation using SQL
- Operators used in SQL
- Nested queries and Joins
- PL/SQL

7.3 Basics of SQL Queries

Structured Query Language (SQL) is a standard Database language which is used to create, maintain and retrieve the relational database. SQL is a domain-specific language used in programming and designed for managing data held in a relational database management system, or for stream processing in a relational data stream management system. We shall discuss extensively about each and every SQL queries in the following sections.

7.4 Examples of Basic SQL Queries

This section presents the syntax of a simple SQL query and explains its meaning through a *conceptual evaluation strategy*. A conceptual evaluation strategy is a way to evaluate the query that is intended to be easy to understand, rather than efficient. A DBMS would typically execute a query in a different and more efficient way.

The SQL CREATE TABLE Statement

The CREATE TABLE statement is used to create a new table in a database.

Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

The SQL INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways.

The first way specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:

INSERT INTO *table_name*
VALUES (*value1*, *value2*, *value3*, ...);

The basic form of an SQL query is as follows:

SELECT [DISTINCT] select-list FROM from-list WHERE qualification

Such a query intuitively corresponds to a relational algebra expression involving selections, projections, and cross-products. Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause. Let us consider a simple query.

The relations discussed in the chapter 5 are considered, for explaining the SQL Queries:

Table 7.1 a. 'Customer' Relation

cid	Cname	mobile	city
C1001	B. Raju	8976894567	Chennai
C2546	M. Sunil	8876921234	Salem
C1456	C. Kunal	9444324578	Chennai
C1324	K. Kamal	9443478902	Madurai
C4578	K. Chitra	9897452123	Chennai
C3456	S. Bala	9789012134	Karaikudi
C4896	K. Kamal	9443478902	Madurai

Table 7.1 b. 'Hotel' Relation

hid	hname	Rate
H101	The Conclave	5000.00
H124	Heritage Inn	6750.00
H456	The Holidays	4300.00

Table 7.1 c. 'Reserves' Relation

cid	hid	Day
C1001	H101	8.9.2019
C1456	H456	17.8.2019
C4578	H124	4.9.2019
C1324	H101	6.8.2019
C1001	H101	7.10.2019
C2456	H456	14.9.2019
C1456	H124	28.8.2019

(Q 7.1) Find the names and mobile numbers of all customers.

SELECT DISTINCT C.cname, C.mobile FROM Customer C

The answer is a *set* of rows, each of which is a pair 〈*cname, mobile*〉. If two or more customers have the same name and mobile number, the answer still contains just one pair with that name and mobile number. This query is equivalent to applying the projection operator of relational algebra.

The answer to this query with and without the keyword DISTINCT on the relation 'Customer' is shown in Table 7.2.a and 7.2.b. The only difference is that the tuple for K. Kamal appears twice if DISTINCT is omitted; this is because there are two customers called K. Kamal and same mobile number.

(Q 7.2) Find all customers from Chennai.

SELECT C.cname, C.mobile, c.city FROM Customer AS C WHERE c.city = 'Chennai'

We now consider the syntax of a basic SQL query in more detail.

- The from-list in the FROM clause is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears more than once in the from-list.
- The select-list is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.
- The qualification in the WHERE clause is a Boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form *expression* op *expression*, where op is one of the comparison operators *{<, <=, =, <>, >=, >}*. An *expression* is a *column* name, a *constant*, or an (arithmetic or string) expression.
- The DISTINCT keyword is optional. It indicates that the table computed as an answer to this query should not contain *duplicates*, that is, two copies of the same row. The default is that duplicates are not eliminated.

(Q 7.3) Find the names of customers who reserved hotel with id 'H101'.

It can be expressed in SQL as follows:

SELECT C.cname FROM Customer C, Reserves R WHERE C.cid = R.cid AND R.hid= 'H101'

Let us compute the answer to this query on the 'Reserves (R)' relation and 'Customer (C)' relation. The first step is to construct the cross-product $S \times R$, which is shown in Table 7.2.

Table 7.2 Result of C X R

cid	Cname	mobile	city	cid	hid	day
C1001	B. Raju	8976894567	Chennai	C1001	H101	8.9.2019
C1324	K. Kamal	9443478902	Madurai	C1324	H101	6.8.2019
C4578	K. Chitra	9897452123	Chennai	C1001	H101	7.10.2019

The second step is to apply the qualification *C.cid = R.cid* AND *R.hid=H101*. (Note that the first part of this qualification requires a join operation.) This step eliminates the last row from the instance shown in Table 7.2. The third step is to eliminate unwanted columns; only *cname* appears in the SELECT clause. This step leaves us with the result shown in Table 7.2, which is a table with a single column and, as it happens, just one row.

Table 7.3 Result to Query

Cname
B. Raju
K. Kamal

(Q 7.4) Find the mobile number of customers who have reserved hotel H456.

SELECT C.mobile FROM Customer C, Reserves R WHERE C.cid = R.cid AND R.hid= 'H456'

(Q 7.5) Find the names of customers who have reserved at least one hotel.

SELECT C.cname FROM Customer C, Reserves R WHERE C.cid = R.cid

Expressions and Strings in the SELECT Command

SQL supports a more general version of the select-list than just a list of columns. Each item in a select-list can be of the form *expression* AS *column name*, where *expression* is any arithmetic or string expression over column names (possibly prefixed by range variables) and constants.

(Q 7.6) Find the mobile numbers of customers who are coming from the city name starts with letter 'C'.

SELECT C.mobile FROM Customer C WHERE C.city LIKE 'C%'

UNION, INTERSECT, AND EXCEPT

SQL provides three set-manipulation constructs that extend the basic query form presented earlier. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names UNION, INTERSECT, and EXCEPT.

SQL also provides other set operations: IN (to check if an element is in a given set), op ANY, op ALL (to compare a value with the elements in a given set, using comparison operator op), and EXISTS (to check if a set is empty). IN and EXISTS can be prefixed by NOT, with the obvious modification to their meaning.

(Q 7.7) Find the names of customers who have reserved 'H101' or 'H456'.

SELECT C.cname FROM Customer C, Reserves R WHERE C.cid = R.cid AND (C.hid = 'H101' OR C.hid = 'H456')

This query is easily expressed using the OR connective in the WHERE clause. However, the following query, which is identical except for the use of 'and' rather than 'or' in the English version, turns out to be much more difficult:

(Q 7.8) Find the names of Customer who have reserved both 'H101' and 'H456'.

If we were to just replace the use of OR in the previous query by AND, in analogy to the English statements of the two queries, we would retrieve the names of customers who have reserved a room in both H101 and H456.

SELECT C.cname FROM Customer C, Reserves R WHERE C.cid = R.cid AND (C.hid = 'H101' AND C.hid = 'H456')

(Q 7.9) Find the cid of all customers who have reserved 'H101' but not 'H456'.

SELECT C.cname FROM Customer C, Reserves R WHERE C.cid = R.cid AND C.hid = 'H101' **EXCEPT** (SELECT C1.cname FROM Customer C1, Reserves R1 WHERE C1.cid = R1.cid AND C1.hid = 'H456')

Note that UNION, INTERSECT, and EXCEPT can be used on *any* two tables that are union-compatible, that is, have the same number of columns and the columns, taken in order, have the same types.

7.5 Introduction to Nested Queries

NESTED QUERIES

A nested query is a query that has another query embedded within it; the embedded query is called a subquery.

Introduction to Nested Queries

As an example, let us rewrite the following query, which we discussed earlier, using a nested subquery:

(Q 7.10) Find the names of customers who have reserved 'H101'.

```
SELECT C.cname
FROM Customer C
WHERE C.cid IN ( SELECT R.cid
                 FROM Reserves R
                 WHERE R.hid = 'H101')
```

The nested subquery computes the (multi) set of *cid*s for customers who have reserved hotel H101, and the top-level query retrieves the names of customers whose *cid* is in this set. The IN operator allows us to test whether a value is in a given set of elements; an SQL query is used to generate the set to be tested.

Notice that it is very easy to modify this query to find all customers who have *not* reserved hotel 'H101' — we can just replace IN by NOT IN→

```
SELECT C.cname
FROM Customer C
WHERE C.cid NOT IN (SELECT R.cid
                    FROM Reserves R
                    WHERE R.hid = 'H101')
```

(Q 7.11) Find the names of customers who have reserved a hotel named 'The Conclave'.

```
SELECT C.cname
FROM Customer C
WHERE C.cid IN (SELECT R.cid
                FROM Reserves R
                WHERE R.hid IN (SELECT H.hid
                                FROM Hotel H
                                WHERE H.hname ='The Conclave')
```

- The innermost subquery finds the set of *hid*s of the hotel 'The Conclave' (H101).
- The subquery one level above finds the set of *cid*s of Customer who have reserved H101 (The Conclave).
- The top-level query finds the names of Customer whose *cid* is in this set of *cid*s.

(Q 7.12) Find the names of Customers who have* not *reserved 'Heritage Inn'.

```
SELECT C.cname
FROM Customer C
WHERE C.cid NOT IN (SELECT R.cid
                    FROM Reserves R
                    WHERE R.hid IN (SELECT H.hid
                                    FROM Hotel H
                                    WHERE H.hname ='Heritage Inn')
```

7.6 Correlated Nested Queries Set

In the nested queries that we have seen thus far, the inner subquery has been completely independent of the outer query. In general the inner subquery could depend on the row that is currently being examined in the outer query (in terms of our conceptual evaluation strategy). Let us rewrite the following query once more:

(Q 7.13) Find the names of Customers who have reserved Hotel 'H101'.

SELECT C.cname FROM Customer C WHERE EXISTS (SELECT * FROM Reserves R WHERE R.hid = 'H101'AND R.cid = C.cid)

The EXISTS operator is another set comparison operator, such as IN. It allows us to test whether a set is nonempty. Thus, for each Customer row *S*, we test whether the set of Reserves rows *R* such that *R.hid = H101* AND *C.cid = R.cid* is nonempty. If so, Customer *C* has reserved hotel 'H101', and we retrieve the name. The subquery clearly depends on the current row *C* and must be re-evaluated for each row in Customer. The occurrence of *C* in the subquery (in the form of the literal *C.cid*) is called a *correlation*, and such queries are called *correlated queries*.

7.7 Comparison Operators

Set-Comparison Operators

We have already seen the set-comparison operators EXISTS, IN, and UNIQUE, along with their negated versions. SQL also supports op ANY and op ALL, where op is one of the arithmetic comparison operators *{<, <=, =, <>, >=, >}*. (SOME is also available, but it is just a synonym for ANY.)

(Q 7.14) Find Hotel whose rate is lesser than Hotel 'The Conclave'.

```
SELECT H.hname
FROM Hotel H
WHERE H.rate < ANY (SELECT H1.rate
                    FROM Hotel H1
                    WHERE H1.hname = 'The Conclave')
```

Table 7.14 Result of the Query (Q 7.14)

Hid	hname	Rate
H456	The Holidays	4300.00

(Q 7.15) Find the Hotel with the highest rate.

```
SELECT H.hname
FROM Hotel H
WHERE H.rate >= ALL (SELECT H1.rate FROM Hotel H1)
```

The subquery computes the set of all rate values in Hotel relation. The outer WHERE condition is satisfied only when *H.rate* is greater than or equal to each of these rate values, i.e., when it is the largest rate value.

In the Hotel relation, the condition is only satisfied for *highest rate is Rs. 6750.00*, and the answer includes the *hnames* of Hotels with this rating, i.e., Heritage Inn.

hname
Heritage Inn

Note that IN and NOT IN are equivalent to = ANY and <> ALL, respectively.

7.8 Aggregative Operators

We now consider a powerful class of constructs for computing *aggregate values* such as MIN and SUM. These features represent a significant extension of relational algebra. SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

For explaining the aggregate function the following relation is used:

Student (regno, name, class, age, mobile, address, percentage)

(Q 7.16) Find the average age of all Students.

SELECT AVG (S.age) FROM Student S

(Q 7.17) Find the average age of Students of a particular class 'II UG Physics'.

SELECT AVG (S.age) FROM Student S WHERE S.class = 'II UG Physics'

(Q 7.18) Find the name and age of the oldest Student of II UG Physics.

Consider the following attempt to answer this query:

SELECT S.name, MAX (S.age) FROM Student S WHERE S.class = 'II UG Physics'

The intent is for this query to return not only the maximum age but also the name of the Student having that age. However, this query is illegal in SQL—if the SELECT clause uses an aggregate operation,

then it must use *only* aggregate operations unless the query contains a GROUP BY clause→ (The intuition behind this restriction should become clear when we discuss the GROUP BY clause in later Section). Thus, we cannot use MAX (S.age) as well as S.name in the SELECT clause.

We have to use a nested query to compute the desired answer to 7.18:

```
SELECT S.name, S.age
FROM Student S
WHERE S.age = (SELECT MAX (S2.age)
    FROM Student S2)
```

Observe that we have used the result of an aggregate operation in the subquery as an argument to a comparison operation. Strictly speaking, we are comparing an age value with the result of the subquery, which is a relation. However, because of the use of the aggregate operation, the subquery is guaranteed to return a single tuple with a single field, and SQL converts such a relation to a field value for the sake of the comparison.

(Q 7.19) Count the number of Students.

```
SELECT COUNT (*) FROM Student S
```

We can think of * as shorthand for all the columns (in the cross-product of the from list in the FROM clause). Contrast this query with the following query, which computes the number of distinct Student names.

(Q 7.20) Find the names of Student who are older than the oldest Student

```
SELECT S.name
FROM Student S
WHERE S.age > (SELECT MAX ( S2.age )
FROM Student S2)
```

The above answer could alternatively be written as follows:

```
SELECT S.name FROM Student S
WHERE S.age > ALL (SELECT S2.age
                  FROM Student S2)
```

The GROUP BY and HAVING Clauses

we want to apply aggregate operations to each of a number of groups of rows in a relation, where the number of groups depends on the relation instance (i.e., is not known in advance). For example, consider the following query.

(Q 7.21) Find the age of the youngest Student.

```
SELECT MIN (S.age)
FROM Student S
```

The GROUP BY clause extension includes an optional HAVING clause that can be used to specify qualifications over groups (for example, we may only be interested in percentage > 60.0). The general form of an SQL query with these extensions is:

```
SELECT[ DISTINCT ] select-list
FROM from-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
```

Using the GROUP BY clause, we can write Q 7.21 as follows:

```
SELECT S.rating, MIN (S.age)
FROM Student S
GROUP BY S.percentage
```

Let us consider some important points concerning the new clauses:

The select-list in the SELECT clause consists of (1) a list of column names and (2) a list of terms having the form aggop (*column-name*) AS *new-name*. The optional AS *new-name* term gives this column a name in the table that is the result of the query. Any of the aggregation operators can be used for aggop. Every column that appears in (1) must also appear in grouping-list. The reason is that each row in the result of the query corresponds to one *group*, which is a collection of rows that agree on the values of columns in grouping-list. If a column appears in list (1), but not in grouping-list, it is not clear what value should be assigned to it in an answer row.
The expressions appearing in the group-qualification in the HAVING clause must have a *single* value per group. The intuition is that the HAVING clause determines whether an answer row is to be generated for a given group. Therefore, a column appearing in the group-qualification must appear as the argument to an aggregation operator, or it must also appear in grouping-list. If the GROUP BY clause is omitted, the entire table is regarded as a single group. We will explain the semantics of such a query through an example. Consider the query:

(Q 7.22) Find the age of the youngest Student who is eligible to vote (i.e., is at least 18 years old) for each top percentage with at least two such Student.

```
SELECT S.rating, MIN (S.age) AS minage
FROM Student S
WHERE S.age >= 18
GROUP BY S.percentage
HAVING COUNT (*) > 1
```

7.9 Introduction to Null Values

NULL VALUES

We have assumed that column values in a row are always known. In practice column values can be unknown. For example, when a Student, admitted new to a class, he may not yet have a percentage. Since the definition for the Student table has a *percentage* column, what row should we insert for that student? What is needed here is a special value that denotes *unknown*.

SQL provides a special column value called *null* to use in such situations. We use *null* when the column value is either *unknown* or *inapplicable*. The presence of *null* values complicates many issues, and we consider the impact of *null* values on SQL in this section.

7.10 Comparison using NULL Values

Comparisons Using Null Values

Consider a comparison such as *percentage = 80.0*. If this is applied to the row for an, is this condition true or false? Since new student's percentage is unknown, it is reasonable to say that this comparison should evaluate to the value unknown. In fact, this is the case for the comparisons *percentage* > 80.0 and *percentage* < 80.0 as well. Perhaps less obviously, if we compare two *null* values using <, >, =, and so on, the result is always unknown. For example, if we have *null* in two distinct rows of the Student relation, any comparison returns unknown. SQL also provides a special comparison operator IS NULL to test whether a column value is *null*; for example, we can say *percentage* IS NULL, which would evaluate to true on the row representing new student. We can also say *percentage* IS NOT NULL, which would evaluate to false on the row for new student.

7.11 Logical connectivity AND, OR, and NOT

There are three Logical Operators namely, AND, OR, and NOT. These operators compare two conditions at a time to determine whether a row can be selected for the output. When retrieving data using a SELECT statement, you can use logical operators in the WHERE clause, which allows you to combine more than one condition.

Table 7.15 Logical Operators and its working in SQL queries

Logical Operators	Description
OR	For the row to be selected at least one of the conditions must be true.
AND	For a row to be selected all the specified conditions must be true.
NOT	For a row to be selected the specified condition must be false.

Nested Logical Operators:

You can use multiple logical operators in an SQL statement. When you combine the logical operators in a SELECT statement, the order in which the statement is processed is

1) NOT
2) AND
3) OR

Now, what about boolean expressions such as *percentage* = 80.0 OR *age* < 23 and *percentage* = 80.0 AND *age* < 23? Considering the row for that new student again, because *age* < 23, the first expression evaluates to true regardless of the value of *percentage*, but what about the second? We can only say unknown. But this example raises an important point—once we have *null* values, we use to define the logical operators AND, OR, and NOT using a *three-valued* logic in which expressions evaluate to true, false, or unknown. We extend the usual interpretations of AND, OR, and NOT to cover the case when one of the arguments is unknown as follows. The expression NOT unknown is defined to be unknown. OR of two arguments evaluates to true if either argument evaluates to true, and to unknown if one argument evaluates

7.12 Outer Join

In the SQL outer JOIN all the content of the both tables are integrated together either they are matched or not.

Outer join of three types:

1.Left outer join (also known as left join): this join returns all the rows from left table combine with the matching rows of the right table. If you get no matching in the right table it returns NULL values.

2.Right outer join (also known as right join): this join returns all the rows from right table are combined with the matching rows of left table .If you get no column matching in the left table .it returns null value.

3. **Full Outer Join** returns unmatched rows from both tables.

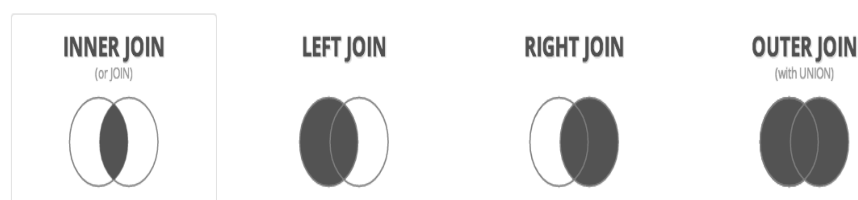


Figure 7.1 Various forms of Joins

7.13 Disallowing NULL Values

To disallow NULL values in any of the columns, add NOT NULL to the definition of each one.

This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

CREATE TABLE Student (ID Number Primary Key, Name varchar2 (25) NOT NULL, Mobile Number(10) NOT NULL);

The above CREATE statement enforces that, ID is the primary key attribute and values of both Name and Mobile should not be a NULL.

7.14 PL/SQL

PL/SQL Introduction

PL/SQL is a block structured language that enables developers to combine the power of SQL with procedural statements.All the statements of a block are passed to oracle engine all at once which increases processing speed and decreases the traffic.

Disadvantages of SQL:

- SQL doesn't provide the programmers with a technique of condition checking, looping and branching.
- SQL statements are passed to Oracle engine one at a time which increases traffic and decreases speed.
- SQL has no facility of error checking during manipulation of data.

Features of PL/SQL:

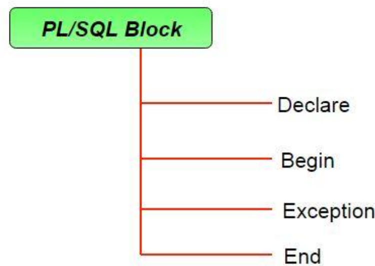
1. PL/SQL is basically a procedural language, which provides the functionality of decision making, iteration and many more features of procedural programming languages.
2. PL/SQL can execute a number of queries in one block using single command.
3. One can create a PL/SQL unit such as procedures, functions, packages, triggers, and types, which are stored in the database for reuse by applications.
4. PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
5. Applications written in PL/SQL are portable to computer hardware or operating system where Oracle is operational.
6. PL/SQL Offers extensive error checking.

Differences between SQL and PL/SQL:

SQL	PL/SQL
SQL is a single query that is used to perform DML and DDL operations.	PL/SQL is a block of codes that used to write the entire program blocks/ procedure/ function, etc.
It is declarative, that defines what needs to be done, rather than how things need to be done.	PL/SQL is procedural that defines how the things needs to be done.
Execute as a single statement.	Execute as a whole block.
Mainly used to manipulate data.	Mainly used to create an application.
Cannot contain PL/SQL code in it.	It is an extension of SQL, so it can contain SQL inside it.

Structure of PL/SQL Block:

PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL. The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other.



Typically, each block performs a logical action in the program. A block has the following structure:

```
DECLARE
   declaration statements;
BEGIN
   executable statements
EXCEPTIONS
   exception handling statements
END;
```

- Declare section starts with **DECLARE** keyword in which variables, constants, records as cursors can be declared which stores data temporarily. It basically consists definition of PL/SQL identifiers. This part of the code is optional.
- Execution section starts with **BEGIN** and ends with **END** keyword.This is a mandatory section and here the program logic is written to perform any task like loops and conditional statements. It supports all DML commands, DDLcommands and SQL*PLUS built-in functions as well.

- Exception section starts with **EXCEPTION** keyword.This section is optional which contains statements that are executed when a run-time error occurs. Any exceptions can be handled in this section.

PL/SQL identifiers

There are several PL/SQL identifiers such as variables, constants, procedures, cursors, triggers etc.

1. **Variables**:
   Like several other programming languages, variables in PL/SQL must be declared prior to its use. They should have a valid name and data type as well.

Syntax for declaration of variables:

variable_name datatype [NOT NULL := value];

```
SQL> SET SERVEROUTPUT ON;

SQL> DECLARE
   var1 INTEGER;
   var2 REAL;
   var3 varchar2(20) ;

BEGIN
   null;
END;
/
```

Output:
PL/SQL procedure successfully completed.

Explanation:

- **SET SERVEROUTPUT ON**: It is used to display the buffer used by the dbms_output.
- **var1 INTEGER :** It is the declaration of variable, named ***var1*** which is of integer type. There are many other data types that can be used like float, int, real, smallint, long etc. It also supports variables used in SQL as well like NUMBER(prec, scale), varchar, varchar2 etc.
- **PL/SQL procedure successfully completed.:** It is displayed when the code is compiled and executed successfully.
- **Slash (/) after END;:** The slash (/) tells the SQL*Plus to execute the block.

INITIALISING VARIABLES:

The variables can also be initialised just like in other programming languages.

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
    var1 INTEGER := 2 ;
    var3 varchar2(20) := 'I Love RDBMS' ;

  BEGIN
    null;

  END;
```

```
/
```

Output:
PL/SQL procedure successfully completed.

Explanation:

- **Assignment operator (:=)** : It is used to assign a value to a variable.

2. **Displaying Output**:

   The outputs are displayed by using DBMS_OUTPUT which is a built-in package that enables the user to display output, debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers.

Let us see an example to see how to display a message using PL/SQL :

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
    var varchar2(40) := 'I love GeeksForGeeks' ;

  BEGIN
    dbms_output.put_line(var);

  END;
  /
```

Output:
I love GeeksForGeeks

PL/SQL procedure successfully completed.

Explanation:

- *dbms_output.put_line* : This command is used to direct the PL/SQL output to a screen.

3. **Using Comments**:

   Like in many other programming languages, in PL/SQL also, comments can be put within the code which has no effect in the code. There are two syntaxes to create comments in PL/SQL :

   - **Single Line Comment:** To create a single line comment , the symbol – – is used.
   - **Multi Line Comment:** To create comments that span over several lines, the symbol /* and */ is used.

Example to show how to create comments in PL/SQL :

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE

    -- I am a comment, so i will be ignored.
    var varchar2(40) := 'I love RDBMS' ;
```

```
BEGIN
  dbms_output.put_line(var);

END;
/
```

Output:
I love RDBMS

PL/SQL procedure successfully completed.

4. **Taking input from user**:

   Just like in other programming languages, in PL/SQL also, we can take input from the user and store it in a variable. Let us see an example to show how to take input from users in PL/SQL:

```
SQL> SET SERVEROUTPUT ON;

SQL> DECLARE

    -- taking input for variable a
    a number := &a;

    -- taking input for variable b
    b varchar2(30) := &b;

  BEGIN
    null;

  END;
  /
```

```
Output:
Enter value for a: 24
old   2: a number := &a;
new   2: a number := 24;
Enter value for b: 'RDBMS'
old   3: b varchar2(30) := &b;
new   3: b varchar2(30) := ' RDBMS ';
```

PL/SQL procedure successfully completed.

5. **(***)** ***Let us see an example on PL/SQL to demonstrate all above concepts in one single block of code.***

```
--PL/SQL code to print sum of two numbers taken from the user.
SQL> SET SERVEROUTPUT ON;

SQL> DECLARE

    -- taking input for variable a
    a integer := &a ;

    -- taking input for variable b
```

```
    b integer := &b ;
    c integer ;

BEGIN
    c := a + b ;
    dbms_output.put_line('Sum of '||a||' and '||b||' is = '||c);

END;
/
```

Enter value for a: 2
Enter value for b: 3

Sum of 2 and 3 is = 5

PL/SQL procedure successfully completed.

Check Your Progress

4. Define: SQL Queries
5. What are Nested Queries?
6. What is meant by NULL Value?

7.15 Complex Integrity Constraints in SQL Triggers and Active Databases.

TRIGGERS

A trigger is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an active database. A trigger description contains three parts:

Event: A change to the database that activates the trigger.

Condition: A query or test that is run when the trigger is activated.

Action: A procedure that is executed when the trigger is activated and its condition is true.

A trigger can be thought of as a 'daemon' that monitors a database, and is executed when the database is modified in a way that matches the *event* specification. An insert, delete or update statement could activate a trigger, regardless of which user or application invoked the activating statement; users may not even be aware that a trigger was executed as a side effect of their program.
A *condition* in a trigger can be a true/false statement (e.g., all customers balance are less than Rs. 1,000) or a query. A query is interpreted as *true* if the answer set is nonempty, and *false* if the query has no answers. If the condition part evaluates to true, the action associated with the trigger is executed.

A trigger *action* can examine the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement

activating the trigger, execute new queries, and make changes to the database. In fact, an action can even execute a series of data-definition commands (e.g., create new tables, change authorizations) and transaction-oriented commands (e.g., commit), or call host language procedures.

An important issue is when the action part of a trigger executes in relation to the statement that activated the trigger. For example, a statement that inserts records into the Students table may activate a trigger that is used to maintain statistics on how many students younger than 18 are inserted at a time by a typical insert statement. Depending on exactly what the trigger does, we may want its action to execute *before* changes are made to the Students table, or *after*: a trigger that initializes a variable used to count the number of qualifying insertions should be executed before, and a trigger that executes once per qualifying inserted record and increments the variable should be executed after each record is inserted (because we may want to examine the values in the new record to determine the action).

Examples of Triggers in SQL

The examples shown below, written using Oracle 8 syntax for defining triggers, illustrate the basic concepts behind triggers. The trigger called *init count* initializes a counter variable before every execution of an INSERT statement that adds tuples to the Students relation. The trigger called *incr count* increments the counter for each inserted tuple that satisfies the condition *age* < 18.

```
CREATE TRIGGER init count BEFORE INSERT ON Students /* Event */
DECLARE
count INTEGER;
BEGIN /* Action */
count := 0;
END

CREATE TRIGGER incr count AFTER INSERT ON Students/* Event */
WHEN (new.age < 18) /* Condition; 'new' is just-inserted tuple */
FOR EACH ROW
BEGIN /* Action; a procedure in Oracle's PL/SQL syntax */
count := count + 1;
END
```

The first example trigger executes before the activating statement, and the other example executes after. A trigger can also be scheduled to execute *instead of* the activating statement, or in *deferred* fashion, at the end of the transaction containing the activating statement, or in *asynchronous* fashion, as part of a separate transaction.

The example illustrates another point about trigger execution: A user must be able to specify whether a trigger is to be executed once per modified record or once per activating statement. If the action depends on individual changed records, for example, we have to examine the *age* field of the inserted Students record to decide whether to increment the count, the triggering event should be defined to occur for each modified record; the FOR EACH ROW clause is used to do this. Such a trigger is called a row-level trigger. On the

other hand, the *init count* trigger is executed just once per INSERT statement, regardless of the number of records inserted, because we have omitted the FOR EACH ROW phrase. Such a trigger is called a statement-level trigger.

In the above example, the keyword new refers to the newly inserted tuple. If an existing tuple were modified, the keywords old and new could be used to refer to the values before and after the modification.

Such a trigger is shown in below example and is an alternative to the triggers shown in above. The keyword clause NEW TABLE enables us to give table name (InsertedTuples) to the set of newly inserted tuples. The FOR EACH STATEMENT clause specifies a statement-level trigger and can be omitted because it is the default. This definition does not have a WHEN clause; if such a clause is included, it follows the FOR EACH STATEMENT clause, just before the action specification.

The trigger is evaluated once for each SQL statement that inserts tuples into Students, and inserts a single tuple into a table that contains statistics on modifications to database tables. The first two fields of the tuple contain constants (identifying the modified table, Students, and the kind of modifying statement, an INSERT), and the third field is the number of inserted Students tuples with *age* < 18.

```
CREATE TRIGGER set count AFTER INSERT ON Students /* Event */
REFERENCING NEW TABLE AS InsertedTuples
FOR EACH STATEMENT
INSERT /* Action */
INTO StatisticsTable(ModifiedTable, ModificationType, Count) SELECT
'Students', 'Insert', COUNT *
FROM InsertedTuples I
WHERE I.age < 18
```

7.16 Answers to Check Your Progress Questions

1. Structured Query Language (SQL) is a standard Database language which is used to create, maintain and retrieve the relational database.
2. A nested query is a query that has another query embedded within it; the embedded query is called a subquery.
3. SQL provides a special column value called null to use in such situations. To use null when the column value is either unknown or inapplicable

7.17 Summary

- **Structured Query Language** (SQL) is a standard Database language
- Each item in a **select-list** can be of the form of expression.
- **SQL** also provides other **set operations**.
- A **nested query** is a query that has another query **embedded** within it

- The occurrence of C in the subquery (in the form of the literal C.cid) is called a **correlation**.
- Using **null** when the column value is either **unknown** or **inapplicable**.
- There are three Logical Operators namely, **AND, OR, and NOT.**
- In the SQL **outer JOIN** all the content of the both tables are integrated together either they are matched or not.
- Add **NOT NULL** to the definition of each one which is disallow the **NULL** values in any of the columns
- SQL is a single query that is used to perform **DML** and **DDL** operations.
- **PL/SQL** is a block of codes that used to write the entire program blocks/ procedure/ function.
- A **trigger** is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the **DBA**.

7.18 Keywords

- A simple **SQL** query and explains its meaning through a conceptual evaluation strategy
- The embedded query is called a **subquery**.
- **Correlated Queries** are used for the purpose of row-by-row processing
- Join returns all the rows from the left table combine with the matching rows of the right table is known **Left Outer Join**.
- Join returns all the rows from right table are combined with the matching rows of left table is known as **Right Outer Join**.
- There are several PL/SQL identifiers such as **variables, constants, procedures, cursors, triggers** etc.
- A database that has a set of associated **triggers** is called an **active database**.

7.19 Self-Assessment Questions and Exercises

Short Answer Questions:

1. Define: SQL
2. What is meant by Nested Query?
3. What are the Aggregative Operators?

4. What is a NULL value?
5. What is an Outer Join?
6. List the limitations of SQL.
7. What is Trigger?

Long Answer Questions:

1. Explain the Basic SQL queries with examples.
2. Describe the Logical Operators and its working in SQL queries
3. Describe the types of Outer join
4. Explain the significances of PL/SQL.
5. Write short-notes on the following
    a. Structure of PL/SQL Block
    b. Feature of PL/SQL
6. Explain the concept of Complex Integrity Constraints using Triggers

7.20 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011.

UNIT VIII NORMAL FORMS

Structure

8.1 Introduction
8.2 Objectives
8.3 Problems caused by redundancy
8.4 Decompositions
8.5 Problem related to decomposition
8.6 Reasoning about FDS
8.7 FIRST, SECOND, THIRD Normal Forms
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

8.1 Introduction

Normalization is the process of minimizing **redundancy** from a relation or set of relations. Redundancy in relation may cause insertion, deletion and updation anomalies. So, it helps to minimize the redundancy in relations. **Normal forms** are used to eliminate or reduce redundancy in database tables.

8.2 Objectives

After reading this chapter, you will get insights about:

- Fundamentals of Normalization and Normal forms
- Dependency preservation in relations
- Decompositions and its problems
- Normal forms: 1NF, 2NF, 3 NF and BCNF

8.3 Problems caused by redundancy

Problems Caused by Redundancy

Redundancy means having multiple copies of same data in the database. This problem arises when a database is not normalized. Storing the same information redundantly, that is, in more than one place within a database, can lead to several problems: **Redundant storage:** Some information is stored repeatedly. Problems caused due to redundancy are: Insertion anomaly, Deletion anomaly, and Updation anomaly.

- **Insertion anomalies:** It may not be possible to store some information unless some other information is stored as well.
- **Deletion anomalies:** It may not be possible to delete some information without losing some other information as well.
- **Updation anomalies:** If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.

Suppose a relation of student details attributes are:

Student (Regno, name, mobile, learning_centre, course).

- **Insertion Anomaly –** If a student detail has to be inserted whose course is not being decided yet then insertion will not be possible till the course is decided for student. This problem happens when the insertion of a data record is not possible without adding some additional unrelated data to the record.
- **Deletion Anomaly** – If the details of students in this table is deleted then the details of learning_centre will also get deleted which should not occur by common sense. This anomaly happens when deletion of a data record results in losing some unrelated information that was stored as part of the record that was deleted from a table.
- **Updation Anomaly –** Suppose if student changes his/her mobile number then changes will have to be all over the database which will be time-consuming and computationally costly. If updation do not occur at all places then database will be in inconsistent state.

Let us consider whether the use of *null* values can address some of these problems. Clearly, *null* values cannot help eliminate redundant storage or update anomalies. It appears that they can address insertion and deletion anomalies. Ideally, we want schemas that do not permit redundancy, but at the very least we want to be able to identify schemas that do allow redundancy. Even if we choose to accept a schema with some of these drawbacks, perhaps owing to performance considerations, we want to make an informed decision.

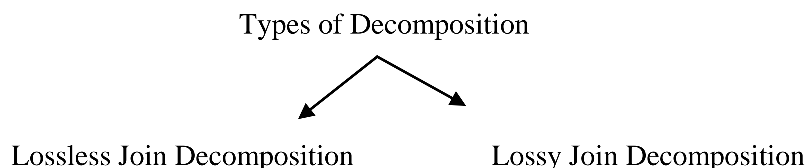
8.4 Decompositions

DECOMPOSITIONS

The process of breaking up or dividing a single relation into two or more sub relations is called as decomposition of a relation.

Types of Decomposition-

Decomposition of a relation can be completed in the following two ways-



Lossless decomposition-

Lossless decomposition ensures-

- No information is lost from the original relation during decomposition.
- When the sub relations are joined back, the same relation is obtained that was decomposed. Every decomposition must always be lossless.

Lossless Join Decomposition-

Consider there is a relation R which is decomposed into sub relations R_1 , R_2 , …. , R_n. This decomposition is called lossless join decomposition when the join of the sub relations results in the same relation R that was decomposed. For lossless join decomposition, we always have-

$$R_1 \bowtie R_2 \bowtie R_3 \ldots\ldots \bowtie R_n = R$$

where ⋈ is a natural join operator

Example-

Consider the following relation R (A , B , C)-

Table 8.1 R (A , B , C)

A	B	C
1	2	1
2	5	3
3	3	3

Consider this relation is decomposed into two sub relations R_1 (A , B) and R_2 (B , C). The two sub relations are-

Table 8.2 R1 (A , B)

A	B
1	2
2	5
3	3

Table 8.3 R2 (B , C)

B	C
2	1
5	3
3	3

Now, let us check whether this decomposition is lossless or not.

For lossless decomposition, we must have-

$$R_1 \bowtie R_2 = R$$

Now, if we perform the natural join (⋈) of the sub relations R_1 and R_2 , we get-

Table 8.4 Natural Join of **R_1 and R_2**

A	B	C
1	2	1
2	5	3
3	3	3

This relation is same as the original relation R.

Thus, we conclude that the above decomposition is lossless join decomposition.

NOTE-

- Lossless join decomposition is also known as **non-additive join decomposition.**
- This is because the resultant relation after joining the sub relations is same as the decomposed relation.
- No extraneous tuples appear after joining of the sub-relations.

Lossy Join Decomposition-

Consider there is a relation R which is decomposed into sub relations R_1 , R_2 , …. , R_n.

- This decomposition is called lossy join decomposition when the join of the sub relations does not result in the same relation R that was decomposed.

- The natural join of the sub relations is always found to have some extraneous tuples.
- For lossy join decomposition, we always have-

$$R_1 \bowtie R_2 \bowtie R_3 \ldots\ldots \bowtie R_n \supset R$$

where ⋈ is a natural join operator

Example-

Consider the following relation RL (A , B , C)-

Table 8.5 RL (A , B , C)

A	B	C
1	2	1
2	5	3
3	3	3

Consider this relation is decomposed into two sub relations as RL_1(A , C) and RL_2(B , C)- The two sub relations are-

Table 8.6 RL_1 (A , C)

A	C
1	1
2	3
3	3

Table 8.7 RL_2 (B , C)

B	C
2	1
5	3
3	3

NOTES

Now, let us check whether this decomposition is lossy or not. For lossy decomposition, we must have-

$$RL_1 \bowtie RL_2 \supset RL$$

Now, if we perform the natural join (⋈) of the sub relations RL_1 and RL_2 we get-

Table 8.8 Natural Join of **RL_1 and RL_2**

A	B	C
1	2	1
2	5	3
2	3	3
3	5	3
3	3	3

This relation is not same as the original relation R and contains some extraneous tuples. Thus, we conclude that the above decomposition is lossy join decomposition.

NOTE-

Lossy join decomposition is also known as **careless decomposition.**

- This is because extraneous tuples get introduced in the natural join of the sub-relations.
- Extraneous tuples make the identification of the original tuples difficult.

Determining Whether Decomposition Is Lossless Or Lossy-

Consider a relation R is decomposed into two sub relations R_1 and R_2.

Then,

- If all the following conditions satisfy, then the decomposition is lossless.
- If any of these conditions fail, then the decomposition is lossy.

Condition-01:

Union of both the sub relations must contain all the attributes that are present in the original relation R.

Thus,

$R1 \cup R2 = R$

Condition-02:

- Intersection of both the sub relations must not be null.
- In other words, there must be some common attribute which is present in both the sub relations.

Thus,

$R1 \cap R2 \neq \emptyset$

Condition-03:

Intersection of both the sub relations must be a super key of either R_1 or R_2 or both.

Thus,

R1 ∩ R2 = Super key of R1 or R2

8.5 Problem related to decomposition

Problems Related to Decomposition

Unless we are careful, decomposing a relation schema can create more problems than it solves. Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problems (if any) does a given decomposition cause?

To help with the first question, several *normal forms* have been proposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. Considering the normal form of a given relation schema can help us to decide whether or not to decompose it further. If we decide that a relation schema must be decomposed further, we must choose a particular decomposition (i.e., a particular collection of smaller relations to replace the given relation). With respect to the second question, two properties of decompositions are of particular interest. The *lossless-join* property enables us to recover any instance of the decomposed relation from corresponding instances of the smaller relations.

The *dependency-preservation* property enables us to enforce any constraint on the original relation by simply enforcing some constraints on each of the smaller relations. That is, we need not perform joins of the smaller relations to check whether a constraint on the original relation is violated.

Dependency Preservation-

Dependency preservation ensures-

- None of the functional dependencies that holds on the original relation are lost.
- The sub relations still hold or satisfy the functional dependencies of the original relation.

8.6 Reasoning about FDS

FUNCTIONAL DEPENDENCIES

Functional Dependency is when one attribute determines another attribute in a DBMS system. Functional Dependency plays a vital role to find the difference between good and bad database design.

Table 8.9 Employee Relation

Employee number	Employee Name	Salary	City
e1	Sunil	50000	Chennai
e2	Francis	38000	Bengaluru
e3	Akmal	25000	Punjab

In this example, if we know the value of Employee number, we can obtain Employee Name, city, salary, etc.

By this, we can say that the city, Employee Name, and salary are functionally depended on Employee number.

A functional dependency is denoted by an arrow →

The functional dependency of X on Y is represented by X →Y

A functional dependency (FD) is a kind of IC that generalizes the concept of a *key*. Let R be a relation schema and let X and Y be nonempty sets of attributes in R. We say that an instance r of R satisfies the FD $X \rightarrow Y$ 1 if the following holds for every pair of tuples $t1$ and $t2$ in r:

$$\text{If } t1{:}X = t2{:}X, \text{ then } t1{:}Y = t2{:}Y .$$

We use the notation $t1{:}X$ to refer to the projection of tuple $t1$ onto the attributes in X, in a natural extension of our TRC notation $t{:}a$ for referring to attribute a of tuple t. An FD $X \rightarrow Y$ essentially says that if two tuples agree on the values in attributes X, they must also agree on the values in attributes Y.

A primary key constraint is a special case of an FD. The attributes in the key play the role of X, and the set of all attributes in the relation plays the role of Y. Note, however, that the definition of an FD does not require that the set X be minimal; the additional minimality condition must be met for X to be a key.

If $X \rightarrow Y$ holds, where Y is the set of all attributes, and there is some subset V of X such that $V \rightarrow Y$ holds, then X is a *super key*; if V is a strict subset of X, then X is not a key. In the rest of this chapter, we will see several examples of FDs that are not key constraints.

Key terms

Table 8.10 Key terms for functional dependency

Key Terms	Description
Axiom	Axioms is a set of inference rules used to infer all the functional dependencies on a relational database.
Decomposition	It is a rule that suggests if you have a table that appears to contain two entities which are determined by the same primary key then you should consider breaking them up into two different tables.
Dependent	It is displayed on the right side of the functional dependency diagram.
Determinant	It is displayed on the left side of the functional dependency Diagram.
Union	It suggests that if two tables are separate, and the PK is the same, you should consider putting them. together

Rules of Functional Dependencies

Below given are the Three most important rules for Functional Dependency:

- Reflexive rule: If X is a set of attributes and Y is_subset_of X, then X holds a value of Y.
- Augmentation rule: When x → y holds, and c is attribute set, then ac → bc also holds. That is adding attributes which do not change the basic dependencies.
    - Transitivity rule: This rule is very much similar to the transitive rule in algebra if x → y holds and y → z holds, then x → z also holds. X → y is called as functionally that determines y.

Closure of a Set of FDs

The set of all FDs implied by a given set F of FDs is called the closure of F and is denoted as $F+$. An important question is how we can infer, or compute, the closure of a given set F of FDs. The answer is simple and elegant. The following three rules, called Armstrong's Axioms, can be applied repeatedly

to infer all FDs implied by a set F of FDs. We use X, Y, and Z to denote *sets* of attributes over a relation schema R:

- Reflexivity: If X Y, then $X \rightarrow Y$.
- Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z.
- Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Armstrong's Axioms are sound in that they generate only FDs in $F+$ when applied to a set F of FDs. They are complete in that repeated application of these rules will generate all FDs in the closure $F+$. (We will not prove these claims.) It is convenient to use some additional rules while reasoning about $F+$:

- Union: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.
- Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$.

These additional rules are not essential; their soundness can be proved using Armstrong's Axioms.

Types of Functional Dependencies

- **Multivalued dependency:**
- **Trivial functional dependency**:
- **Non-trivial functional dependency**:
- **Transitive dependency:**

Multivalued dependency in DBMS

Multivalued dependency occurs in the situation where there are multiple independent multivalued attributes in a single table. A multivalued dependency is a complete constraint between two sets of attributes in a relation. It requires that certain tuples be present in a relation.

Table 8.11 Cars Relation

Car_model	Mf_year	Color
H001	2017	Metallic
H001	2017	Green
H005	2018	Metallic
H005	2018	Blue
H010	2015	Metallic
H033	2012	Gray

In this example, mf_year and color are independent of each other but dependent on car_model. In this example, these two columns are said to be multivalue dependent on car_model.

This dependence can be represented like this:

car_model -> mf_year

car_model-> colour

Trivial Functional dependency:

The Trivial dependency is a set of attributes which are called a trivial if the set of attributes are included in that attribute.

So, X -> Y is a trivial functional dependency if Y is a subset of X.

Table 8.12 Employee Relation

Emp_id	Emp_name
AS555	Satya Nadella
AS811	Sundar Pichai
AS999	Tim Cook

Consider this table with two columns Emp_id and Emp_name.

{Emp_id, Emp_name} -> Emp_id is a trivial functional dependency as Emp_id is a subset of {Emp_id,Emp_name}.

Non trivial functional dependency in DBMS

Functional dependency which also known as a nontrivial dependency occurs when A->B holds true where B is not a subset of A. In a relationship, if attribute B is not a subset of attribute A, then it is considered as a non-trivial dependency.

Table 8.12 Company and CEO Relation

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Apple	Tim Cook	57

(Company} -> {CEO} (if we know the Company, we knows the CEO name)

But CEO is not a subset of Company, and hence it's non-trivial functional dependency.

Transitive dependency:

A transitive is a type of functional dependency which happens when t is indirectly formed by two functional dependencies.

Table 8.13 Company and CEO Relation

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Alibaba	Jack Ma	54

NOTES

{Company} -> {CEO} (if we know the compay, we know its CEO's name)

{CEO } -> {Age} If we know the CEO, we know the Age

Therefore according to the rule of rule of transitive dependency:

{ Company} -> {Age} should hold, that makes sense because if we know the company name, we can know his age.

Note: You need to remember that transitive dependency can only occur in a relation of three or more attributes.

Advantages of Functional Dependency

- Functional Dependency avoids data redundancy. Therefore same data do not repeat at multiple locations in that database
- It helps you to maintain the quality of data in the database
- It helps you to defined meanings and constraints of databases
- It helps you to identify bad designs
- It helps you to find the facts regarding the database design

8.7 FIRST, SECOND, THIRD Normal Forms

NORMAL FORMS

Given a relation schema, we need to decide whether it is a good design or whether we need to decompose it into smaller relations. Such a decision must be guided by an understanding of what problems, if any, arise from the current schema. To provide such guidance, several normal forms have been proposed. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.

The normal forms based on FDs are fi*rst normal form (1NF)*, *second normal form (2NF)*, *third normal form (3NF)*, and *Boyce-Codd normal form (BCNF)*. These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF, every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF. A relation is in first normal form if every field contains only atomic values, that is, not lists or sets. This requirement is implicit in our de nition of the relational model. Although some of the newer database systems are relaxing this requirement, in this chapter we will assume that it always holds. 2NF is mainly of historical interest. 3NF and BCNF are important from a database design standpoint.

While studying normal forms, it is important to appreciate the role played by FDs. Consider a relation schema *R* with attributes *ABC*. In the absence of any ICs, any set of ternary tuples is a legal instance and there is no potential for

redundancy. On the other hand, suppose that we have the FD $A \rightarrow B$. Now if several tuples have the same A value, they must also have the same B value. This potential redundancy can be predicted using the FD information. If more detailed ICs are specified, we may be able to detect more subtle redundancies as well.

First Normal Form

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Example: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

Table 8.14 Employee Address Relation

emp_id	emp_name	emp_address	emp_mobile
101	Roy	Salem	8912312390
102	Suresh	Karur	8812121212 9900012222
103	Mahesh	Chennai	7778881212
104	Sunil	Theni	9990000123 8123450987

Two employees (Suresh & Sunil) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says "each attribute of a table must have atomic (single) values", the emp_mobile values for employees Suresh & Sunil violates that rule.

To make the table complies with 1NF we should have the data like this:

Table 8.15 Employee Address Relation in 1 NF

emp_id	emp_name	emp_address	emp_mobile
101	Roy	Salem	8912312390
102	Suresh	Karur	8812121212
102	Suresh	Karur	9900012222
103	Mahesh	Chennai	7778881212
104	Sunil	Theni	9990000123
104	Sunil	Theni	8123450987

Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

- Table is in 1NF (First normal form)
- No non-prime attribute is dependent on the proper subset of any candidate key of table.

NOTES

An attribute that is not part of any candidate key is known as non-prime attribute.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

Table 8.16 Teacher Relation in 1NF and not in 2NF

teacher_id	Subject	teacher_mobile
111	Maths	8912312390
111	Physics	8912312390
222	Biology	9990000123
333	Physics	7778881212
333	Chemistry	7778881212

Candidate Keys: {teacher_id, subject}
Non prime attribute: teacher_mobile

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_mobile is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this:

Table 8.17 Teacher-Details Relation in both 1NF and 2NF

teacher_id	teacher_mobile
111	8912312390
222	9990000123

Table 8.18 Teacher-Subject Relation in both 1NF and 2NF

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- Transitive functional dependency of non-prime attribute on any super key should be removed.

An attribute that is not part of any candidate key is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one of the following conditions hold:

- X is a super key of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee_details that looks like this:

Table 8.19 Employee_Details Relation

emp_id	emp_name	pincode	emp_state	emp_city	emp_loc
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urrapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Super keys: {emp_id}, {emp_id, emp_name}, {emp_id, emp_name, pincode}…so on
Candidate Keys: {emp_id}
Non-prime attributes: all attributes except emp_id are non-prime as they are not part of any candidate keys.

Here, emp_state, emp_city & emp_district dependent on pincode. And, pincode is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

Table 8.20 Emp Relation

emp_id	emp_name	pincode
1001	John	282005
1002	Ajeet	222008
1006	Lora	282007
1101	Lilly	292008
1201	Steve	222999

NOTES

Table 8.21 Emp_pincode Relation

pincode	emp_state	emp_city	emp_loc
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urrapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

The definition of 3NF is similar to that of BCNF, with the only difference being the third condition. Every BCNF relation is also in 3NF. To understand the third condition, recall that a key for a relation is a *minimal* set of attributes that uniquely determines all other attributes. *A* must be part of a key (any key, if there are several). It is not enough for *A* to be part of a super key, because the latter condition is satisfied by each and every attribute→ Finding all keys of a relation schema is known to be an NP-complete problem, and so is the problem of determining whether a relation schema is in 3NF.

Partial dependencies are illustrated in Figure 8.1, partial and transitive dependencies are illustrated. Note that in Figure 8.1, the set *X* of attributes may or may not have some attributes in common with KEY; the diagram should be interpreted as indicating only that *X* is not a subset of KEY.

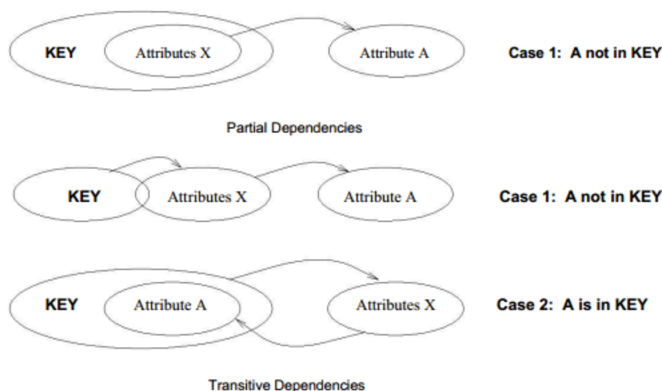


Fig. 8.1 Partial and Transitive Dependencies

The motivation for 3NF is rather technical. By making an exception for certain dependencies involving key attributes, we can ensure that every relation schema can be decomposed into a collection of 3NF relations using only decompositions that have certain desirable properties. Such a guarantee does not exist for BCNF relations; the 3NF definition weakens the BCNF requirements just enough to make this guarantee possible.

Check Your Progress

1. What is meant by Redundancy?
2. What is called non-additive join decomposition?
3. What is the purpose of FDS?

8.8 BCNF

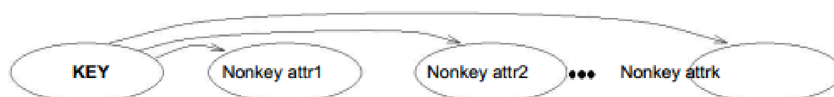
Boyce-Codd Normal Form

Let *R* be a relation schema, *X* be a subset of the attributes of *R*, and let *A* be an attribute of *R*. *R* is in Boyce-Codd normal form if for every FD $X \rightarrow A$ that holds over *R*, one of the following statements is true:

- *A 2 X*; that is, it is a trivial FD, or
- *X* is a super key.

Note that if we are given a set *F* of FDs, according to this definition, we must consider each dependency $X \rightarrow A$ in the closure $F+$ to determine whether *R* is in BCNF. However, we can prove that it is sufficient to check whether the left side of each dependency in *F* is a super key (by computing the attribute closure and seeing if it includes all attributes of *R*).

Intuitively, in a BCNF relation the only nontrivial dependencies are those in which a key determines some attribute(s). Thus, each tuple can be thought of as an entity or relationship, identified by a key and described by the remaining attributes. Kent puts this colorfully, if a little loosely: Each attribute must describe [an entity or relationship identified by] the key, the whole key, and nothing but the key." If we use ovals to denote attributes or sets of attributes and draw arcs to indicate FDs, a relation in BCNF has the structure illustrated in Figure, considering just one key for simplicity. (If there are several candidate keys, each candidate key can play the role of KEY in the figure, with the other attributes being the ones not in the chosen candidate key.)



FDs in a BCNF Relation

BCNF ensures that no redundancy can be detected using FD information alone. It is thus the most desirable normal form (from the point of view of redundancy) if we take into account only FD information.

NOTES

Boyce Codd normal form (BCNF)

It is an advance version of 3NF that's why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every functional dependency X->Y, X should be the super key of the table.

Example: Suppose there is a company wherein manager manages in **more than one department**. They store the data like this:

Table 8.22 Manager_Department Relation

Mg_id	Mg_city	Mg_dept	Mg-type	No of Emps
1001	Chennai	Production & Planning	D001	200
1001	Chennai	Stores	D001	250
1002	Pune	Design & Technical Support	D134	100
1002	Pune	Purchasing Department	D134	600

Functional dependencies in the table above:
Mg_id -> Mg-city
Mg_dept -> {Mg_type, No of Emps}

Candidate key: {Mg_id, Mg_dept}

The table is not in BCNF as neither Mg_id nor Mg_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:
Table 8.23.a Manager_City Relation

Mg_id	Mg_city
1001	Chennai
1002	Pune

Table 8.23.b Department Relation

Mg_dept	Mg-type	No of Emps
Production & Planning	D001	200
Stores	D001	250
Design & Technical Support	D134	100
Purchasing Department	D134	600

Table 8.23.c Manager-Department Relation

Mg_id	Mg_dept
1001	Production & Planning
1001	Stores
1002	Design & Technical Support
1002	Purchasing Department

Functional dependencies:
Mg_id -> Mg_city
Mg_dept -> {Mg_type, No of Emps}

Candidate keys:
For first table: Mg_id
For second table: Mg_dept
For third table: {Mg_id, Mg_dept}

This is now in BCNF as in both the functional dependencies left side part is a key.

Thus, if a relation is in BCNF, every field of every tuple records a piece of information that cannot be inferred (using only FDs) from the values in all other fields in (all tuples of) the relation instance.

8.9 Answers to Check Your Progress Questions

1. Redundancy means having multiple copies of same data in the database. This problem arises when a database is not normalized.
2. Lossless join decomposition is also known as non-additive join decomposition.
3. Functional Dependency is a vital role to find the difference between good and bad database design.

8.10 Summary

- Functional Dependency is when one attribute determines another attribute in a DBMS system.
- Axiom, Decomposition, Dependent, Determinant, Union are key terms for functional dependency
- Four types of functional dependency are 1) Multivalued 2) Trivial 3) Non-trivial 4) Transitive
- Multivalued dependency occurs in the situation where there are multiple independent multivalued attributes in a single table
- The Trivial dependency occurs when a set of attributes which are called a trivial if the set of attributes are included in that attribute
- Nontrivial dependency occurs when A→B holds true where B is not a subset of A
- A transitive is a type of functional dependency which happens when it is indirectly formed by two functional dependencies
- Normalization is a method of organizing the data in the database which helps you to avoid data redundancy
- Functional dependency helps you to maintain the quality of data in the database

8.11 Keywords

- **Redundancy** means having multiple copies of same data in the database.
- The process of breaking up or dividing a single relation into two or more sub relations is called as **decomposition** of a relation.
- **Lossless join** decomposition is also known as non-additive join decomposition.
- **Functional Dependency** is when one attribute determines another attribute in a **DBMS** system.
- A **multivalued dependency** is a complete constraint between two sets of attributes in a relation.
- A transitive is a type of functional dependency which happens when it is indirectly formed by two functional dependencies.

8.12 Self-Assessment Questions and Exercises

Short Answer Questions:

1. Define: Redundancy
2. What is meant by Decompositions?
3. What is a Functional Dependency?
4. Name the different types of Functional Dependency
5. What is meant by Normal Forms?

Long Answer Questions:

1. Write the Problems Caused by Redundancy?
2. Write the types of Decompositions?
3. Write short notes on
    a. Functional Dependencies
    b. Key terms for Functional Dependency
    c. Rules of Functional Dependencies
4. Write short notes on
    a. Multivalued dependency
    b. Trivial functional dependency
    c. Non-trivial functional dependency
    d. Transitive dependency
5. Write short notes on First, Second, Third Normal Forms

8.13 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011.

UNIT IX JOINS

Structure

9.1 Introduction
9.2 Objectives
9.3 Lossless Join Decomposition
9.4 Dependency preserving Decomposition
9.5 Schema refinement in Database Design
9.6 Multi valued Dependencies
9.7 FORTH Normal Form
9.8 Answers to Check Your Progress Questions
9.9 Summary
9.10 Key Words
9.11 Self-Assessment Questions and Exercises
9.12 Further Readings

9.1 Introduction

A join is an SQL operation performed to establish a connection between two or more database tables based on matching columns, thereby creating a relationship between the tables. Most complex queries in an SQL database management system involve join commands. There are different types of joins. The type of join a programmer uses determines which records the query selects.

9.2 Objectives

This chapter will impart the fundamentals about:

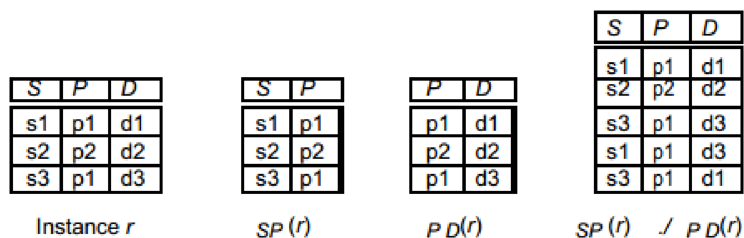
- Dependency preserving Decomposition
- Schema refinement in Database Design
- Multi-valued Dependencies
- 4NF

9.3 Lossless Join Decomposition

Lossless-Join Decomposition

Let *R* be a relation schema and let *F* be a set of FDs over *R*. A decomposition of *R* into two schemas with attribute sets *X* and *Y* is said to be a lossless-join decomposition with respect to F if for every instance *r* of *R* that satisfies the dependencies in *F*, *X* (*r*) ./ *Y* (*r*) = *r*.

This definition can easily be extended to cover a decomposition of *R* into more than two relations. It is easy to see that *r X* (*r*) ./ *Y* (*r*) always holds. In general, though, the other direction does not hold. If we take projections of a relation and recombine them using natural join, we typically obtain some tuples that were not in the original relation. This situation is illustrated in Figure 15.11.



S	P	D
s1	p1	d1
s2	p2	d2
s3	p1	d3

Instance *r*

S	P
s1	p1
s2	p2
s3	p1

SP (*r*)

P	D
p1	d1
p2	d2
p1	d3

P D(*r*)

S	P	D
s1	p1	d1
s2	p2	d2
s3	p1	d3
s1	p1	d3
s3	p1	d1

SP (*r*) ./ *P D*(*r*)

Instances Illustrating Lossy Decompositions

By replacing the instance *r* shown in Figure 15.11 with the instances *SP* (*r*) and *P D*(*r*), we lose some information. In particular, suppose that the tuples in *r* denote relationships. We can no longer tell that the relationships (*s*1*; p*1*; d*3) and (*s*3*; p*1*; d*1) do not hold. The decomposition of schema *SPD* into *SP* and *PD* is therefore a `lossy' decomposition if the instance *r* shown in the figure is legal, that is, if this instance could arise in the enterprise being modeled. *All decompositions used to eliminate redundancy* must *be lossless.* The following simple test is very useful: Let *R* be a relation and *F* be a set of FDs that hold over *R*. The decomposition of *R* into relations with attribute sets *R*1 and *R*2 is lossless if and only if *F* + contains either the FD *R*1 \ *R*2 → *R*1 or the FD *R*1 \ *R*2 → *R*2.

In other words, the attributes common to *R*1 and *R*2 must contain a key for either *R*1 or *R*2. If a relation is decomposed into two relations, this test is a necessary and sufficient condition for the decomposition to be lossless-join. If a relation is decomposed into more than two relations, an efficient (time polynomial in the size of the dependency set) algorithm is available to test whether or not the decomposition is lossless, but we will not discuss it.

Consider the Hourly Emps relation again. It has attributes *SNLRWH*, and the FD *R* → *W* causes a violation of 3NF. We dealt with this violation by decomposing the relation into *SNLRH* and *RW*. Since *R* is common to both decomposed relations, and *R* → *W* holds, this decomposition is lossless-join.

This example illustrates a general observation:

If an FD *X* → *Y* holds over a relation *R* and *X* *Y* is empty, the decomposition of *R* into *R* - *Y* and *XY* is lossless.

X appears in both *R* -*Y* (since *X* \ *Y* is empty) and *XY*, and it is a key for *XY*. Thus, the above observation follows from the test for a lossless-join decomposition.

Another important observation has to do with repeated decompositions. Suppose that a relation *R* is decomposed into *R1* and *R2* through a lossless-

join decomposition, and that *R1* is decomposed into *R11* and *R12* through another lossless-join decomposition. Then the decomposition of *R* into *R11*, *R12*, and *R2* is lossless-join; by joining *R11* and *R12* we can recover *R1*, and by then joining *R1* and *R2*, we can recover *R*.

9.4 Dependency preserving Decomposition

Dependency-Preserving Decomposition

Consider the Contracts relation with attributes *CSJDPQV* from Section 15.4.1. The given FDs are $C \rightarrow CSJDPQV$, $JP \rightarrow C$, and $SD \rightarrow P$. Because *SD* is not a key the dependency $SD \rightarrow P$ causes a violation of BCNF.

We can decompose Contracts into two relations with schemas *CSJDQV* and *SDP* to address this violation; the decomposition is lossless-join. There is one subtle problem, however. We can enforce the integrity constraint $JP \rightarrow C$ easily when a tuple is inserted into Contracts by ensuring that no existing tuple has the same *JP* values (as the inserted tuple) but different *C* values. Once we decompose Contracts into *CSJDQV* and *SDP*, enforcing this constraint requires an expensive join of the two relations whenever a tuple is inserted into *CSJDQV*. We say that this decomposition is not dependency-preserving.

Intuitively, a *dependency-preserving decomposition* allows us to enforce all FDs by examining a single relation instance on each insertion or modification of a tuple. (Note that deletions cannot cause violation of FDs.) To de ne dependency-preserving decompositions precisely, we have to introduce the concept of a projection of FDs.

Let *R* be a relation schema that is decomposed into two schemas with attribute sets *X* and *Y*, and let *F* be a set of FDs over *R*. The projection of F on *X* is the set of FDs in the closure $F+$ (not just $F \rightarrow$) that involve only attributes in *X*. We will denote the projection of *F* on attributes *X* as *FX* . Note that a dependency $U \rightarrow V$ in $F+$ is in *FX* only if *all* the attributes in *U* and *V* are in *X*.

The decomposition of relation schema *R* with FDs *F* into schemas with attribute sets *X* and *Y* is dependency-preserving if $(FX [FY)+ = F +$. That is, if we take the dependencies in *FX* and *FY* and compute the closure of their union, we get back all dependencies in the closure of *F*. Therefore, we need to enforce only the dependencies in *FX* and *FY* ; all FDs in $F+$ are then sure to be satisfied. To enforce *FX* , we need to examine only relation *X* (on inserts to that relation). To enforce *FY* , we need to examine only relation *Y*.

9.5 Schema refinement in Database Design

We now present an overview of the problems that schema refinement is intended to address and a refinement approach based on decompositions. Redundant storage of information is the root cause of these problems. Although decomposition can eliminate redundancy, it can lead to problems of its own and should be used with caution.

9.6 Multivalued Dependencies

Multivalued Dependencies

Suppose that we have a relation with attributes *course*, *teacher*, and *book*, which we denote as *CTB*. The meaning of a tuple is that teacher *T* can teach course *C*, and book *B* is a recommended text for the course. There are no FDs; the key is *CTB*. However, the recommended texts for a course are independent of the Faculty. The instance shown in the following table illustrates this situation.

Table 9.1 Course – Teacher – Book Relation

Course	Teacher	Book
Physics101	Raj	Mechanics
Physics101	Raj	Optics
Physics101	Kumar	Mechanics
Physics101	Kumar	Optics
Math301	Sunil	Mechanics
Math301	Sunil	Vectors
Math301	Sunil	Geometry

BCNF Relation with Redundancy That Is Revealed by MVDs

There are three points to note here:

- The relation schema *CTB* is in BCNF; thus we would not consider decomposing it further if we looked only at the FDs that hold over *CTB*.
- There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.
- The redundancy can be eliminated by decomposing *CTB* into *CT* and *CB*.

The redundancy in this example is due to the constraint that the texts for a course are independent of the Faculty, which cannot be expressed in terms of FDs. This constraint is an example of a *multivalued dependency*, or MVD. Ideally, we should model this situation using two binary relationship sets,

Faculty with attributes *CT* and Text with attributes *CB*. Because these are two essentially independent relationships, modeling them with a single ternary relationship set with attributes *CTB* is inappropriate. Given the subjectivity of ER design, however, we might create a ternary relationship. A careful analysis of the MVD information would then reveal the problem.

Let *R* be a relation schema and let *X* and *Y* be subsets of the attributes of *R*. Intuitively, the multivalued dependency $X \rightarrow\rightarrow Y$ is said to hold over *R* if, in every legal instance *r* of *R*, each *X* value is associated with a set of *Y* values and this set is independent of the values in the other attributes. Formally, if the MVD $X \rightarrow\rightarrow Y$ holds over *R* and $Z = R - XY$, the following must be true for every legal instance *r* of *R*:

If $t1\ 2\ r$, $t2\ 2\ r$ and $t1{:}X = t2{:}X$, then there must be some $t3\ 2\ r$ such that $t1{:}XY = t3{:}XY$ and $t2{:}Z = t3{:}Z$.

Following relation illustrates this definition. If we are given the first two tuples and told that the MVD $X \rightarrow\rightarrow Y$ holds over this relation, we can infer that the relation instance must also contain the third tuple. Indeed, by interchanging the roles of the rst two tuples treating the first tuple as *t*2 and the second tuple as *t*1|we can deduce that the tuple *t*4 must also be in the relation instance.

X	Y	Z	
a	*b*1	*c*1	| tuple *t*1
a	*b*2	*c*2	| tuple *t*2
a	*b*1	*c*2	| tuple *t*3
a	*b*2	*c*1	| tuple *t*4

Illustration of MVD Definition

This table suggests another way to think about MVDs: If $X \rightarrow\rightarrow Y$ holds over *R*, then $Y\ Z\ (\ X{=}x(R)) = Y\ (\ X{=}x(R))\ Z\ (\ X{=}x(R))$ in every legal instance of *R*, for any value *x* that appears in the *X* column of *R*. In other words, consider groups of tuples in *R* with the same *X*-value, for each *X*-value. In each such group consider the projection onto the attributes *YZ*. This projection must be equal to the cross-product of the projections onto *Y* and *Z*. That is, for a given *X*-value, the *Y*-values and *Z*-values are independent. (From this definition it is easy to see that $X \rightarrow\rightarrow Y$ must hold whenever $X \rightarrow Y$ holds. If the FD $X \rightarrow Y$ holds, there is exactly one *Y*-value for a given *X*-value, and the conditions in the MVD definition hold trivially)

Returning to our *CTB* example, the constraint that course texts are independent of Faculty members can be expressed as $C \rightarrow\rightarrow T$. In terms of the definition of MVDs, this constraint can be read as follows:

\If (there is a tuple showing that) *C* is taught by teacher *T*,

and (there is a tuple showing that) C has book B as text,
then (there is a tuple showing that) C is taught by T and has text B.

Given a set of FDs and MVDs, in general we can infer that several additional FDs and MVDs hold. A sound and complete set of inference rules consists of the three Armstrong Axioms plus ve additional rules. Three of the additional rules involve only MVDs:

- MVD Complementation: If $X \rightarrow\rightarrow Y$, then $X \rightarrow\rightarrow R - XY$.
- MVD Augmentation: If $X \rightarrow\rightarrow Y$ and $W Z$, then $WX \rightarrow\rightarrow YZ$.
- MVD Transitivity: If $X \rightarrow\rightarrow Y$ and $Y \rightarrow\rightarrow Z$, then $X \rightarrow\rightarrow (Z\text{-}Y)$

As an example of the use of these rules, since we have $C \rightarrow\rightarrow T$ over CTB, MVD complementation allows us to infer that $C \rightarrow\rightarrow CT\ B - CT$ as well, that is, $C \rightarrow\rightarrow B$. The remaining two rules relate FDs and MVDs:
Replication: If $X \rightarrow Y$, then $X \rightarrow\rightarrow Y$.

- Coalescence: If $X \rightarrow\rightarrow Y$ and there is a W such that $W \setminus Y$ is empty, $W \rightarrow Z$, and $Y Z$, then $X \rightarrow Z$.

Observe that replication states that every FD is also an MVD.

Check Your Progress

1. What is meant by loseless join decomposition?
2. What is meant by Schema Refinement?
3. Define: MVD

9.7 FORTH Normal Form

Fourth Normal Form

Fourth normal form is a direct generalization of BCNF. Let R be a relation schema, X and Y be nonempty subsets of the attributes of R, and F be a set of dependencies that includes both FDs and MVDs. R is said to be in fourth normal form (4NF) if for every MVD $X \rightarrow\rightarrow Y$ that holds over R, one of the following statements is true:

$Y X$ or $XY = R$, or
X is a Superkey.

In reading this definition, it is important to understand that the definition of a key has not changed the key must uniquely determine all attributes through FDs alone. $X \rightarrow\rightarrow Y$ is a trivial MVD if $Y X R$ or $XY = R$; such MVDs always hold.

The relation *CTB* is not in 4NF because $C \to\to T$ is a nontrivial MVD and *C* is not a key. We can eliminate the resulting redundancy by decomposing *CTB* into *CT* and *CB*; each of these relations is then in 4NF.

To use MVD information fully, we must understand the theory of MVDs. However, the following result due to Date and Fagin identifies conditions detected using only FD information→|under which we can safely ignore MVD information. That is, using MVD information in addition to the FD information will not reveal any redundancy. Therefore, if these conditions hold, we do not even need to identify all MVDs.

If a relation schema is in BCNF, and at least one of its keys consists of a single attribute, it is also in 4NF.

An important assumption is implicit in any application of the preceding result: *The set of FDs identified thus far is indeed the set of all FDs that hold over the relation*. This assumption is important because the result relies on the relation being in BCNF, which in turn depends on the set of FDs that hold over the relation.

Following relation shows three tuples from an instance of *ABCD* that satisfies the given MVD $B \to\to C$. From the definition of an MVD, given tuples $t1$ and $t2$, it follows

B	*C*	*A*	*D*	
b	*c*1	*a*1	*d*1	| tuple *t*1
b	*c*2	*a*2	*d*2	| tuple *t*2
b	*c*1	*a*2	*d*2	| tuple *t*3

Three Tuples from a Legal Instance of ABCD

that tuple $t3$ must also be included in the instance. Consider tuples $t2$ and $t3$. From the given FD $A \to BCD$ and the fact that these tuples have the same *A*-value, we can deduce that $c1 = c2$. Thus, we see that the FD $B \to C$ must hold over *ABCD* whenever the FD $A \to BCD$ and the MVD $B \to\to C$ hold. If $B \to C$ holds, the relation *ABCD* is not in BCNF (unless additional FDs hold that make *B* a key)

9.8 Answers to Check Your Progress Questions

1. It is the ability to ensure that any instance of the original relation can be identified from corresponding instances in the smaller relations.
2. Schema refinement is intended to address and a refinement approach based on decompositions.
3. **Multivalued Dependency (MVD)** occurs when two attributes in a table are independent of each other but, both depend on a third attribute. A multivalued dependency consists of at least two attributes that are dependent on a third attribute that's why it always requires at least three attributes.

9.9 Summary

- A relation R with set of functional dependencies F. If R is decomposed into relations R1 and R2, then this decomposition is said to be lossless decomposition
- All decompositions used to eliminate redundancy must be lossless.
- A dependency-preserving decomposition allows us to enforce all FDs by examining a single relation instance on each insertion or modification of a tuple.
- The schema refinement is intended to address and a refinement approach based on decompositions.
- Multivalued dependency occurs when there are more than one independent multivalued attributes in a table.
- Fourth normal form is a direct generalization of BCNF.

9.10 Keywords

- The attributes common to relations *R*1 and *R*2 must contain a key for either *R*1 or *R*2. If a relation is decomposed into two relations, this test is a necessary and sufficient condition for the decomposition to be **lossless-join**.
- Redundant storage of information will leads to problems in dependency preservation.
- Decomposition can eliminate redundancy.
- A dependency-preserving decomposition allows us to enforce all FDs by examining a single relation instance on each insertion or modification of a tuple.
- The schema refinement is intended to address and a refinement approach based on decompositions.
- Multivalued Dependency (MVD) occurs when two attributes in a table are independent of each other but, both depend on a third attribute.
- Multivalued dependency occurs when there are more than one independent multivalued attributes in a table.
- Fourth normal form is a direct generalization of BCNF.

9.11 Self-Assessment Questions and Exercises

Short Answer Questions:

1. What is meant by Lossless Decomposition?
2. Define: Schema Refinement.

Long Answer Questions:

1. Explain the 4NF with example.
2. Elaborate on MVD.

9.12 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011

UNIT X INTRODUCTION

Structure

10.1 Introduction
10.2 Objectives
10.3 Transaction Concept
10.4 Transaction State
10.5 Implementation of Atomicity and Durability
10.6 Concurrent
10.7 Executions
10.8 Serializability
10.9 Implementation of Isolation
10.10 Answers to Check Your Progress Questions
10.11 Summary
10.12 Key Words
10.13 Self-Assessment Questions and Exercises
10.14 Further Readings

10.1 Introduction

A transaction is a logical unit of processing in a DBMS which entails one or more database access operation. In a nutshell, database transactions represent real-world events of any enterprise. All types of database access operation which are held between the beginning and end transaction statements are considered as a single logical transaction. During the transaction the database is inconsistent. Only once the database is committed the state is changed from one consistent state to another.

10.2 Objectives

This chapter leads to understand:

- Basics of Transaction and its states
- ACID Properties
- Uses and issues related to Concurrent transactions
- Serializability

10.3 Transaction Concept

What is a Transaction?

A transaction is an event which occurs on the database. Generally a transaction reads a value from the database or writes a value to the database. If you have any concept of Operating Systems, then we can say that a transaction is analogous to processes.

Although a transaction can both read and write on the database, there are some fundamental differences between these two classes of operations. A read operation does not change the image of the database in any way. But a write operation, whether performed with the intention of inserting, updating or deleting data from the database, changes the image of the database. That is, we may say that these transactions bring the database from an image which existed before the transaction occurred (called the **Before Image** or **BFIM**) to an image which exists after the transaction occurred (called the **After Image** or **AFIM**).

10.4 Transaction State

Transaction States

There are the following six states in which a transaction may exist:

Active: The initial state when the transaction has just started execution.

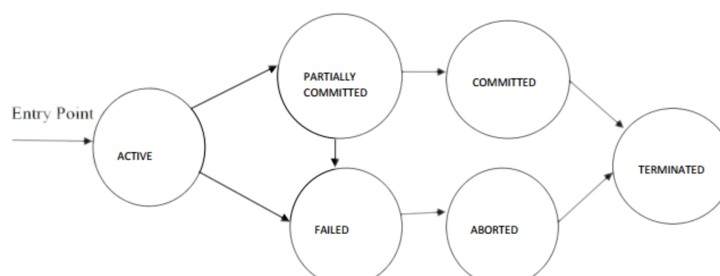
Partially Committed: At any given point of time if the transaction is executing properly, then it is going towards it COMMIT POINT. The values generated during the execution are all stored in volatile storage.

Failed: If the transaction fails for some reason. The temporary values are no longer required, and the transaction is set to **ROLLBACK**. It means that any change made to the database by this transaction up to the point of the failure must be undone. If the failed transaction has withdrawn Rs. 100/- from account A, then the ROLLBACK operation should add Rs 100/- to account A.

Aborted: When the ROLLBACK operation is over, the database reaches the BFIM. The transaction is now said to have been aborted.

Committed: If no failure occurs then the transaction reaches the COMMIT POINT. All the temporary values are written to the stable storage and the transaction is said to have been committed.

Terminated: Either committed or aborted, the transaction finally reaches this state. The whole process can be described using the following diagram



Transaction Characteristics

Every transaction has three characteristics: access mode, diagnostics size, and isolation level. The diagnostics size determines the number of error conditions that can be recorded. If the access mode is READ ONLY, the transaction is not allowed to modify the database. Thus, INSERT, DELETE, UPDATE, and CREATE commands cannot be executed. If we have to execute one of these commands, the access mode should be set to READ WRITE. For transactions with READ ONLY access mode, only shared locks need to be obtained, thereby increasing concurrency. The isolation level controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concurrency at the cost of increasing the transaction's exposure to other transactions' uncommitted changes. Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The effect of these levels is summarized in Table 10.1 given below. In this context, dirty read and unrepeatable read are defined as usual. Phantom is defined to be the possibility that a transaction retrieves a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself. The highest degree of isolation from the effects of other

Table 10.1 Transaction Isolation Levels

Level	Dirty Read	Unrepeatable	Read Phantom
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

transactions is achieved by setting isolation level for a transaction T to SERIALIZABLE. This isolation level ensures that T reads only the changes made by committed transactions, that no value read or written by T is changed by any other transaction until T is complete, and that if T reads a set of values based on some search condition, this set is not changed by other transactions until T is complete (i.e., T avoids the phantom phenomenon).

In terms of a lock-based implementation, a SERIALIZABLE transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged, and holds them until the end, according to Strict 2PL.

REPEATABLE READ ensures that T reads only the changes made by committed transactions, and that no value read or written by T is changed by any other transaction until T is complete. However, T could experience the phantom phenomenon; for example, while T examines all Customer records with rating=1, another transaction might add a new such Customer record, which is missed by T.

A REPEATABLE READ transaction uses the same locking protocol as a SERIALIZABLE transaction, except that it does not do index locking, that is, it locks only individual objects, not sets of objects.

READ COMMITTED ensures that T reads only the changes made by committed transactions, and that no value written by T is changed by any other transaction until T is complete. However, a value read by T may well be modified by another transaction while T is still in progress, and T is, of course, exposed to the phantom problem.

A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete.
(This guarantee relies on the fact that every SQL transaction obtains exclusive locks before writing objects and holds exclusive locks until the end.)

A READ UNCOMMITTED transaction T can read changes made to an object by an ongoing transaction; obviously, the object can be changed further while T is in progress, and T is also vulnerable to the phantom problem.

A READ UNCOMMITTED transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions; so much so that SQL prohibits such a transaction from making any changes itself - a READ UNCOMMITTED transaction is required to have an access mode of READ ONLY. Since such a transaction obtains no locks for reading objects, and it is not allowed to write objects (and therefore never requests exclusive locks), it never makes any lock requests.

The SERIALIZABLE isolation level is generally the safest and is recommended for most transactions. Some transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance.

For example, a statistical query that finds the average Customer age can be run at the READ COMMITTED level, or even the READ UNCOMMITTED level, because a few incorrect or missing values will not significantly affect the result if the number of Customer is large. The isolation level and access mode can be set using the SET TRANSACTION command. For example, the following command

declares the current transaction to be SERIALIZABLE and READ ONLY:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY

When a transaction is started, the default is SERIALIZABLE and READ WRITE

10.5 Implementation of Atomicity and Durability

Atomicity: This means that either all of the instructions within the transaction will be reflected in the database, or none of them will be reflected. Say for example, we have two accounts A and B, each containing Rs 1000/-. We now start a transaction to deposit Rs 100/- from account A to Account B.

Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;

Fine, is not it? The transaction has 6 instructions to extract the amount from A and submit it to B. The AFIM will show Rs 900/- in A and Rs 1100/- in B.

Now, suppose there is a power failure just after instruction 3 (Write A) has been complete. What happens now? After the system recovers the AFIM will show Rs 900/- in A, but the same Rs 1000/- in B. It would be said that Rs 100/- evaporated in thin air for the power failure. Clearly such a situation is not acceptable.

The solution is to keep every value calculated by the instruction of the transaction not in any stable storage (hard disc) but in a volatile storage (RAM), until the transaction completes its last instruction. When we see that there has not been any error we do something known as a **COMMIT** operation. Its job is to write every temporarily calculated value from the volatile storage on to the stable storage. In this way, even if power fails at instruction 3, the post recovery image of the database will show accounts A and B both containing Rs 1000/-, as if the failed transaction had never occurred

Durability:

It states that once a transaction has been complete the changes it has made should be permanent.

As we have seen in the explanation of the Atomicity property, the transaction, if completes successfully, is committed. Once the COMMIT is done, the changes which the transaction has made to the database are immediately written into permanent storage. So, after the transaction has been committed successfully, there is no question of any loss of information even if the power fails. Committing a transaction guarantees that the AFIM has been reached.

There are several ways Atomicity and Durability can be implemented. One of them is called Shadow Copy. In this scheme a database pointer is used to point to the BFIM of the database. During the transaction, all the temporary changes are recorded into a Shadow Copy, which is an exact copy of the original database plus the changes made by the transaction, which is the AFIM. Now, if the transaction is required to COMMIT, then the database pointer is updated to point to the AFIM copy, and the BFIM copy is discarded. On the other hand, if the transaction is not committed, then the database pointer is not updated. It keeps pointing to the BFIM, and the AFIM is discarded. This is a simple scheme, but takes a lot of memory space and time to implement. If you study carefully, you can understand that Atomicity and Durability is essentially the same thing, just as Consistency and Isolation is essentially the same thing.

10.6 Concurrent

A schedule is a collection of many transactions which is implemented as a unit. Depending upon how these transactions are arranged in within a schedule, a schedule can be of two types:

- Serial: The transactions are executed one after another, in a non-preemptive manner.
- Concurrent: The transactions are executed in a preemptive, time shared method.

In Serial schedule, there is no question of sharing a single data item among many transactions, because not more than a single transaction is executing at any point of time. However, a serial schedule is inefficient in the sense that the transactions suffer for having a longer waiting time and response time, as well as low amount of resource utilization.
In concurrent schedule, CPU time is shared among two or more transactions in order to run them concurrently. However, this creates the possibility that more than one transaction may need to access a single data item for read/write purpose and the database could contain inconsistent value if such accesses are not handled properly. Let us explain with the help of an example.

10.7 Concurrent Execution

Let us consider there are two transactions T1 and T2, whose instruction sets are given as following.
T1 is the same as we have seen earlier, while T2 is a new transaction.

T1
Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;

T2
Read A;
Temp = A * 0.1;
Read C;
C = C + Temp;
Write C;

T2 is a new transaction which deposits to account C 10% of the amount in account A. If we prepare a serial schedule, then either T1 will completely finish before T2 can begin, or T2 will completely finish before T1 can begin. However, if we want to create a concurrent schedule, then some Context Switching need to be made, so that some portion of T1 will be executed, then some portion of T2 will be executed and so on. For example say we have prepared the following concurrent schedule.

T1	T2
Read A;	
A = A – 100;	
Write A;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Read B;	
B = B + 100;	
Write B;	

No problem here. We have made some Context Switching in this Schedule, the first one after executing the third instruction of T1, and after executing the last statement of T2. T1 first deducts Rs 100/- from A and writes the new

value of Rs 900/- into A. T2 reads the value of A, calculates he value of Temp to be Rs 90/- and adds the value to C. The remaining part of T1 is executed and Rs 100/- is added to B.

It is clear that a proper Context Switching is very important in order to maintain the Consistency and Isolation properties of the transactions. But let us take another example where a wrong Context Switching can bring about disaster. Consider the following example involving the same T1 and T2

T1	T2
Read A;	
A = A – 100;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Write A;	
Read B;	
B = B + 100;	
Write B;	

This schedule is wrong, because we have made the switching at the second instruction of T1. The result is very confusing. If we consider accounts A and B both containing Rs 1000/- each, then the result of this schedule should have left Rs 900/- in A, Rs 1100/- in B and add Rs 90 in C (as C should be increased by 10% of the amount in A). But in this wrong schedule, the Context Switching is being performed before the new value of Rs 900/- has been updated in A. T2 reads the old value of A, which is still Rs 1000/-, and deposits Rs 100/- in C. C makes an unjust gain of Rs 10/- out of nowhere.

In the above example, we detected the error simple by examining the schedule and applying common sense. But there must be some well formed rules regarding how to arrange instructions of the transactions to create error free concurrent schedules. This brings us to our next topic, the concept of Serializability.

10.8 Serializability

When several concurrent transactions are trying to access the same data item, the instructions within these concurrent transactions must be ordered in some way so as there are no problem in accessing and releasing the shared data item. There are two aspects of serializability which are described here:

Conflict Serializability

Two instructions of two different transactions may want to access the same data item in order to perform a read/write operation. Conflict Serializability deals with detecting whether the instructions are conflicting in any way, and specifying the order in which these two instructions will be executed in case there is any conflict. A **conflict** arises if at least one (or both) of the instructions is a write operation. The following rules are important in Conflict Serializability:

1. If two instructions of the two concurrent transactions are both for read operation, then they are not in conflict, and can be allowed to take place in any order.

2. If one of the instructions wants to perform a read operation and the other instruction wants to perform a write operation, then they are in conflict, hence their ordering is important. If the read instruction is performed first, then it reads the old value of the data item and after the
reading is over, the new value of the data item is written. It the write instruction is performed first, then updates the data item with the new value and the read instruction reads the newly updated value.

3. If both the transactions are for write operation, then they are in conflict but can be allowed to take place in any order, because the transaction do not read the value updated by each other. However, the value that persists in the data item after the schedule is over is the one written by the instruction that performed the last write.

It may happen that we may want to execute the same set of transaction in a different schedule on another day. Keeping in mind these rules, we may sometimes alter parts of one schedule (S1) to create another schedule (S2) by swapping only the non-conflicting parts of the first schedule. The conflicting parts cannot be swapped in this way because the ordering of the conflicting instructions is important and cannot be changed in any other schedule that is derived from the first. If these two schedules are made of the same set of transactions, then both S1 and S2 would yield the same result if the conflict resolution rules are maintained while creating the new schedule. In that case the schedule S1 and S2 would be called **Conflict Equivalent**.

View Serializability:

This is another type of serializability that can be derived by creating another schedule out of an existing schedule, involving the same set of transactions. These two schedules would be called View Serializable if the following rules are followed while creating the second schedule out of the first. Let us consider that the transactions T1 and T2 are being serialized to create

two different schedules www.eazynotes.com Sabyasachi De Page No. 9 S1 and S2 which we want to be **View Equivalent** and both T1 and T2 wants to access the same data item.

1. If in S1, T1 reads the initial value of the data item, then in S2 also, T1 should read the initial value of that same data item.
2. If in S1, T1 writes a value in the data item which is read by T2, then in S2 also, T1 shouldwrite the value in the data item before T2 reads it.
3. If in S1, T1 performs the final write operation on that data item, then in S2 also, T1 should perform the final write operation on that data item.

Except in these three cases, any alteration can be possible while creating S2 by modifying S1.

10.9 Implementation of Isolation:

In case multiple transactions are executing concurrently and trying to access a sharable resource at the same time, the system should create an ordering in their execution so that they should not create any anomaly in the value stored at the sharable resource.

There are several ways to achieve this and the most popular one is using some kind of locking mechanism. Again, if you have the concept of Operating Systems, then you should remember the semaphores, how it is used by a process to make a resource busy before starting to use it, and how it is used to release the resource after the usage is over. Other processes intending to access that same resource must wait during this time. Locking is almost similar. It states that a transaction must first lock the data item that it wishes to access, and release the lock when the accessing is no longer required. Once a transaction locks the data item, other transactions wishing to access the same data item must wait until the lock is released.

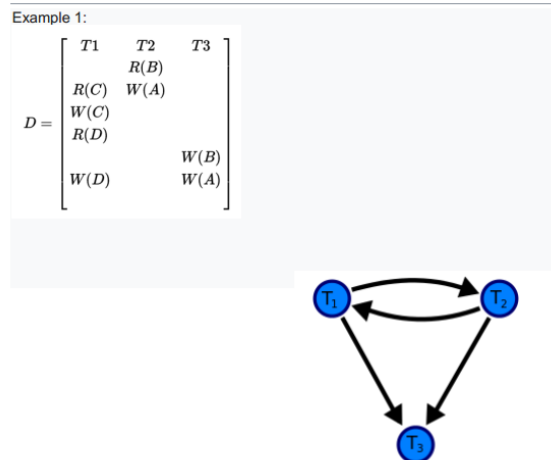
Check Your Progress

1. What is a Transaction?
2. What is meant by Partially Committed state?
3. Define: Atomicity

Precedence graph

A **precedence graph**, also named **conflict graph** and **serializability graph**, is used in the context of **concurrency control in databases**. The precedence graph for a schedule S contains: A node for each committed

transaction in S An arc from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions. Precedence graph example



A precedence graph of the schedule D, with 3 transactions. As there is a cycle (of length 2; with two edges) through the committed transactions T1 and T2, this schedule (history) is *not* Conflict serializability. Testing Serializability with Precedence Graph The drawing sequence for the precedence graph:-

1. For each transaction Ti participating in schedule S, create a node labelled Ti in the precedence graph. So the precedence graph contains T1, T2, T3
2. For each case in S where Ti executes a write_item(X) then Tj executes a read_item(X), create an edge (Ti → Tj) in the precedence graph. This occursnowhere in the above example, as there is no read after write.
3. For each case in S where Ti executes a read_item(X) then Tj executes a write_item(X), create an edge (Ti → Tj) in the precedence graph. This results in a directed edge from T1 to T2.
4. For each case in S where Ti executes a write_item(X) then Tj executes a write_item(X), create an edge (Ti → Tj) in the precedence graph. This results in directed edges from T2 to T1, T1 to T3, and T2 to T3.
5. The schedule S is conflict serializable if the precedence graph has no cycles. As T1 and T2 constitute a cycle, then we cannot declare S as serializable or not and serializability has to be checked

10.10 Answers to Check Your Progress Questions

1. A transaction is an event which occurs on the database. Generally a transaction reads a value from the database or writes a value to the database.
2. At any given point of time if the transaction is executing properly, then it is going towards it COMMIT POINT. The values generated during the execution are all stored in volatile storage.

3. Atomicity means that either all of the instructions within the transaction will be reflected in the database, or none of them will be reflected.

10.11 Summary

- A **transaction** reads a value from the database or writes a value to the database.
- In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or none are known as **atomicity**.
- Durability is the ACID property which guarantees that transactions that have committed will survive permanently.
- The **transactions** are executed in a preemptive, time shared method is known as **concurrent**.
- **Serializability** is a concept that helps us to check which schedules are serializable.
- A **precedence graph**, also named conflict graph and serializability graph, is used in the context of concurrency control in databases.

10.12 Key Words

- A **transaction** is a unit of program execution that accesses and possibly updates various data items.
- The **initial state** when the transaction has just started **execution**.
- **COMMIT POINT** is the transaction is executing properly
- The transaction fails for some reason is **Failed** state
- The **ROLLBACK** operation leads to **Aborted** state
- The transaction finally reaches the state **Terminated** either **committed** or **aborted**
- The **SERIALIZABLE** isolation level is generally the safest and is recommended for most transactions.

10.13 Self-Assessment Questions and Exercises

Short answer questions

1. Define: Transaction
2. Name of the Transaction states
3. What is the serializability?
4. Define: Precedence graph

Long answer questions

1. Describe the properties of Transaction
2. What are the implementation of atomicity and durability
3. Write short notes on
   a. Conflict Serializability
   b. View Serializability:
4. Describe the precedence graph with an example

10.14 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011

UNIT XI PROTOCOLS

Structure

11.1 Introduction
11.2 Objectives
11.3 Lock Based Protocols
11.4 Timestamp Based Protocols
11.5Validation Based Protocols
11.6 Multiple Granularity
11.7 Answers to Check Your Progress Questions
11.8 Summary
11.9 Key Words
11.10 Self-Assessment Questions and Exercises
11.11 Further Readings

11.1 Introduction

Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another. Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical database, would have a mix of reading and WRITE operations and hence the concurrency is a challenge. Concurrency control is used to address such conflicts which mostly occur with a multi-user system. It helps you to make sure that database transactions are performed concurrently without violating the data integrity of respective databases. Therefore, concurrency control is a most important element for the proper functioning of a system where two or multiple database transactions that require access to the same data, are executed simultaneously.

11.2 Objectives

After reading this chapter, you can able to understand:

- Fundamentals of Concurrent transactions
- Uses and issues related to Concurrent transactions
- Protocols to manage concurrent transactions (Lock, time-stamp, validation and etc.)

11.3 Lock Based Protocols

A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**. If a node is locked in an intention mode, explicit locking is done at a lower level of the tree (that is, at a finer granularity).

Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, *Q*—must traverse a path in the tree from the root to *Q*. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-ode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX) mode**, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is shown as follows

Table 11.1: Compatibility matrix

	IS	IX	S	SIX	X
IS	True	True	True	True	False
IX	True	True	false	false	False
S	True	False	True	False	False
SIX	True	False	false	False	False
X	False	false	false	false	false

The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction *Ti* that attempts to lock a node *Q* must follow these rules:

1. Transaction *Ti* must observe the lock-compatibility function of Figure above.

2. Transaction *Ti* must lock the root of the tree first, and can lock it in anymode.

3. Transaction *Ti* can lock a node *Q* in S or IS mode only if *Ti* currently has the parent of *Q* locked in either IX or IS mode.

4. Transaction *Ti* can lock a node *Q* in X, SIX, or IX mode only if *Ti* currently has the parent of *Q* locked in either IX or SIX mode.

5. Transaction *Ti* can lock a node only if *Ti* has not previously unlocked any node (that is, *Ti* is two phase).

6. Transaction *Ti* can unlock a node *Q* only if *Ti* currently has none of the children of *Q* locked.

Locking: Top-Down and Bottom-up

Observe that the multiple-granularity protocol requires that locks be acquired in ***top-down*** (root-to-leaf) order, whereas locks must be released in ***bottom-up*** (leafto-root) order. As an illustration of the protocol:

• Suppose that transaction *T*21 reads record *ra*2 in file *Fa* . Then, *T*21 needs to lock the database, area *A*1, and *Fa* in IS mode (and in that order), and finally to lock *ra*2 in S mode.

• Suppose that transaction *T*22 modifies record *ra*9 in file *Fa* . Then, *T*22 needs to lock the database, area *A*1, and file *Fa* (and in that order) in IX mode, and finally to lock *ra*9 in X mode.

• Suppose that transaction *T*23 reads all the records in file *Fa* . Then, *T*23 needs to lock the database and area *A*1 (and in that order) in IS mode, and finally to lock *Fa* in S mode.

• Suppose that transaction *T*24 reads the entire database. It can do so after locking the database in S mode.

11.4 Timestamp Based Protocols

Timestamp-Based Protocols

The locking protocols that we have described thus far determine the order between every pair of conflicting transactions at execution time by the first lock that both members of the pair request that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a timestamp-ordering scheme.

Timestamps

With each transaction T_i in the system, we associate a unique fixed timestamp, denoted by TS (4). This timestamp is assigned by the database system before the transaction T_i starts execution. If a transaction T_i has been assigned timestamp TS(T_i), and a new transaction Q enters the system, then TS(4) < TS(4).

There are two simple methods for implementing this scheme:

1. Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
2. Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if TS (4) <TS(T), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Q. To implement this scheme, we associate with each data item Q two timestamp values: o W-timestamp (Q) denotes the largest timestamp of any transaction that executed write (Q) successfully. o R-timestamp(8) denotes the largest timestamp of any transaction that executed read(Q) successfully. These timestamps are updated whenever a new read (Q) or write (Q) instruction is executed.

11.5Validation Based Protocols

Validation-Based Protocols

In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state. A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes less overhead. A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for monitoring the system.

We assume that each transaction T_i executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order,

1. **Read phase:** During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variables local to T_i. It performs all write operations on temporary local variables, without updates of the actual database.

2. **Validation phase:** Transaction T_i performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serializability.

3. **Write phase:** If transaction T_i succeeds in validation (step2), then the system applies the actual updates to the database. Otherwise, the system rolls back T_i.

Each transaction must go through the three phases in the order shown. However, all three phases of concurrently executing transactions can be interleaved. To perform the validation test, we need to know when the various

phases of transactions4 took place. We shall, therefore, associate three different timestamps with transaction Z:

1. Start (T_i), the time when I started its execution.
2. Validation (T_i), the time when 4 finished its read phase and started its validation phase.
3. Finish (T_i), the time when fr finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation (T_i). Thus, the value $TS(T_i)$ = Validation(T_i) and, if $TS(T_j) < TS(T_k)$, then any produced schedule must be equivalent to a serial schedule in which transaction T_j appears before transaction T_k. The reason we have chosen Validation (T_i), rather than Start(T_i), as the timestamp of transaction T_i is that we can expect faster response time provided that conflict rates among transactions are indeed low.

The validation test for transaction T_i requires that, for all transactions T_i with $TS(T_i) < TS(T_j)$, one of the following two conditions must hold:

1. Finish (4) < Start (Q). Since I complete its execution before Q started, the serializability order is indeed aintained.
2. The set of data items written by T_i does not intersect with the set of data items read by T_j, and T_i completes its write phase before T_j starts its validation phase (Start(T_j) < Finish(T_i) < Validation(T_j)). This condition ensures that

T_{14}	T_{15}
read (B)	
	read (B)
	B:=B – 50
	read (A)
	A:=A+50
read(A)	
<validate>	
display (A+B)	
	<validate>
	write(B)
	write(A)

Figure 11.2 schedule produced by using validation

the writes of T_i and T*j* do not overlap. Since the writes of T_i do not affect the read of T_j, and since T_j cannot affect the read of T_i, the serializability order is indeed maintained.

As an illustration, consider again transactions T_{14} and T_{15}. Suppose that $TS(T_{14}) < TS(T_{15})$. Then, the validation phase succeeds in the schedule 5 in Figure 16.15.Note that the writes to the actual variables are performed only after the validation phase of T_{15}. Thus, fia reads the old values of B and A, and this schedule is serializable.

The validation scheme automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed. However, there is a possibility of starvation of long transactions, due to a sequence of conflicting short transactions that cause repeated restarts of the long transaction. To avoid starvation, conflicting transactions must be temporarily blocked, to enable the long transaction to finish.

This validation scheme is called the optimistic concurrency-control scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end. In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

Check Your Progress

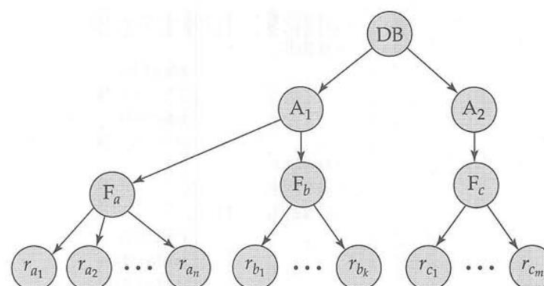
1. What is called as Locking protocol?
2. What is MGL?
3. What is meant by Concurrency control?

11.6 Multiple Granularity

Multiple Granularity

In the concurrency-control schemes described thus 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. For example, if a transaction T_i needs to access the entire database, and a locking protocol is used, then T_i must lock each item in the database. Clearly, executing these locks is time-consuming. It would be better if T_i could issue a single lock request to lock the

Figure 11.3 Granularity hierarchy



entire database. On the other hand, if transaction T_i needs to access only a few data items, it should not be required to lock the entire database, since

otherwise concurrency is lost. What is needed is a mechanism to allow the system to define multiple levels of granularity. We can make one by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree. Note that the tree that we describe here is significantly different from that used by the tree protocol .A non leaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.

As an illustration, consider the tree of Figure 11.3, which consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type area; the database consists of exactly these areas. Each area in turn has nodes of type file as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type record. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

Each node in the tree can be locked individually. As we did in the two-phase locking protocol, we shall use **shared** and **exclusive** lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction T_i gets an explicit lock on file F_c of Figure 11.3, in exclusive mode, then it has an implicit lock in exclusive mode on all the records belonging to that file. It does not need to lock the individual records of F_c explicitly.

Suppose that transaction T_i wishes to lock record r_{b6} of file F_b. Since T_i has locked F_b explicitly, it follows that r_{b6} is also locked (implicitly). But, when T_j issues a lock request for $r_{b6,}$ r_{b6} is not explicitly locked→ How does the system determine whether T_j can lock r_{b6} ? T_j must traverse the tree from the root to record r_{b6}.If any node in that path is locked in an incompatible mode, then T_j must be delayed.

Table 11.3 Transaction States

	IS	IX	S	SIX	X
IS	True	True	True	True	False
IX	True	True	False	false	False
S	True	False	True	False	False
SIX	True	False	False	False	False
X	False	false	False	false	false

Suppose now that transaction T_k wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however, that T_k should not succeed in locking the root node, since T_i is currently holding a lock on part of the tree (specifically, on file F_b). But how does the system determine if the root node can be locked? One possibility is for it to search the entire tree. This solution, however, defeats the whole purpose of the multiple-granularity

locking scheme. A more efficient way to gain this knowledge is to introduce a new class of lock modes, called intention lock modes. If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node-say, Q-must traverse a path in the tree from the root to Q. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in intention-shared (IS) mode, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX)** mode, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The **multiple-granularity** locking protocol, which ensures serializability, is this:

Each transaction T_i can lock a node Q by following these rules:

1. It must observe the lock-compatibility function
2. It must lock the root of the tree first, and can lock it in any mode.
3. It can lock a node Q in S or IS mode only if it currently has the parent of Q locked in either IX or IS mode.
4. It can lock a node Q in X, SIX, or IX mode only if it currently has the parent of Q locked in either IX or SIX mode.
5. It can lock a node only if it has not previously unlocked any node (that is, 4 is two phase).
6. It can unlock a node Q only if it currently has none of the children of Q locked

Observe that the multiple-granularity protocol requires that locks be acquired in *top down* (root-to-leaf) order, whereas locks must be released in *bottom-up* (leaf-to-root) order.

- Suppose that transaction T_{18} reads record r_{a2}, in file F_a. Then, T_{18} needs to lock the database, areaA_1, and F_a in IS mode (and in that order), and finally to lock r_{a2}, in S mode.
- Suppose that transaction T_{19} modifies record r_{a9} in file F_a. Then, T_{19} needs to lock the database, area A_1, and file F_a in IX mode, and finally to lock r_{a9} in X mode.
- Suppose that transaction T_{20} reads all the records in fiie F_a. Then, T_{20} needs to lock the database and area A_1 (in that order) in IS mode, and finally to lock F_a in S mode.
- Suppose that transaction T_{21} reads the entire database. It can do so after locking the database in S mode.

We note that transactions T_{18}, T_{20}, and 7_{21} can access the database concurrently. Tansaction T_{19} can execute concurrently with T_{18} is, but not with either T_{20} or T_{21}. This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of

- Short transactions that access only a few data items
- Long transactions that produce reports from an entire file or set of files

There is a similar locking protocol that is applicable to database systems in which data granularities are organized in the form of a directed acyclic graph. Seethe bibliographical notes for additional references. Deadlock is possible in the protocol that we have, as it is in the two-phase locking protocol. There are techniques to reduce deadlock frequency in the multiple-granularity protocol, and also to eliminate deadlock entirely. These techniques are referenced in the bibliographical notes.

11.7 Answers to Check Your Progress Questions

1. The protocol that indicates when a transaction may lock and unlock each of the data items is called as locking protocol. Locking protocols restrict the number of schedules.
2. Multiple granularity locking (MGL) is a locking method used in database management systems (DBMS) and relational databases. In MGL, locks are set on objects that contain other objects. A lock on such as a shared or exclusive lock locks the targeted node as well as all of its descendants.
3. Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another.

11.8 Summary

- **Lock-based protocols** allow transactions to obtain a lock on every object before a **'write'** operation is performed.
- The protocol uses either system time or logical counter as a **timestamp** protocol
- **Validation based protocol** is used to avoiding **concurrency** in transactions
- Validation scheme is called the optimistic concurrency-control scheme
- **Multiple granularity** locking is usually used with non-strict two-phase locking to guarantee **serializability**

11.9 Key Words

- A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**.
- The multiple-granularity locking protocol uses these lock modes to ensure **serializability**
- The value of the system clock as the **timestamp**
- The timestamps of the transactions determine the serializability order

- A **concurrency-control scheme** imposes overhead of code execution and possible delay of transactions
- The multiple-granularity tree represents the data associated with its descendants

11.10 Self-Assessment Questions and Exercises

Short Answer Questions

1. What is meant by lock based protocols?
2. Define: Timestamp
3. What are validation based protocols?

Long Answer Questions

1. Explain the working of Lock Based Protocols
2. Discuss about Timestamp Protocols
3. Describe the Multiple Granularity

11.11 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011

UNIT XII RECOVERY AND ATOMICITY

Structure

12.1 Introduction
12.2 Objectives
12.3 Log- Based Recovery
12.4 Recovery with Concurrent Transactions
12.5 Buffer Management
12.6 Failure with loss of non-volatile storage
12.7Advance Recovery systems
12.8 Remote Backup systems
12.9 Answers to Check Your Progress Questions
12.10 Summary
12.11 Key Words
12.12 Self-Assessment Questions and Exercises
12.13 Further Readings

12.1 Introduction

DBMS is a highly complex system with hundreds of transactions being executed every second. The durability and robustness of a DBMS depends on its complex architecture and its underlying hardware and system software. If it fails or crashes amid transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data. When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

12.2 Objectives

This chapter leads to understand:

- Various recovery techniques
- Buffer management
- Remote backup systems

12.3Log- Based Recovery

The most widely used structure for recording database modifications is the 1og.The log is a sequence of log records, recording all the update activities in the database. There are several types of log records. An update log record describes a single database write. It has these fields:

- **Transaction identifier** is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- **Old value** is the' value of the data item prior to the write.
- **New value** is the value that the data item will have after the write.

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. We denote the various types of log records as:

- <T_i.start>. Transaction T_i has started.
- <T_i, X_j, V_1, V_2>. Transaction T_i has performed a write on data item X_j. X_j had value V_1 before the write, and will have value v_2 after the write.
- <T_i commit>. Transaction T_i has committed.
- <T_i abort >. Transaction T_i, has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to undo a modification that has already been output to the database. We undo it by using the old-value field in log records.

For log records to be useful for recovery from system and disk failures, the log must reside in stable storage. For now, we assume that every log record is written to the end of the log on stable storage as soon as it is created.

12.4 Recovery with Concurrent Transactions

Recovery with Concurrent Transactions

Until now, we considered recovery in an environment where only a single transaction at a time is executing. We now discuss how we can modify and extend the log-based recovery scheme to deal with multiple concurrent transactions. Regardless of the number of concurrent transactions, the system has a single disk buffer and a single tog. All transactions share the buffer blocks. We allow immediate modification, and permit a buffer block to have data items updated by one or more transactions.

12.4.1 Interaction with Concurrency Control

The recovery scheme depends greatly on the concurrency-control scheme that is used. To roll back a failed transaction, we must undo the updates performed by the 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 restored to its old value. Using the log-based schemes for recovery, we restore the value by using the undo

information in a log record. Suppose now that a second transaction T_1 has performed yet another update on Q before T_0 is rolled back. Then, the update 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, no other transaction may update the same data item until *T* has committed or been rolled back. We can ensure this requirement easily by using strict two-phase locking-that is, two-phase locking with exclusive locks held until the end of the transaction.

12.4.2 Transaction Rollback

We roll back a failed transaction, T_i, by using the log. The system scans the log backward; for every log record of the form <T_i, X_j, V_1, V_2) 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< T_i, start> is found. scanning the log backward is important, since a transaction may have updated a data item more than once. As an illustration, consider the pair of log records

<T_i, A, 10, 20>
<T_i, A, 20, 30>

The log records represent a modification of data item A by T_i, followed by another modification of A by T_i. Scanning the log backward sets , 4 correctly to 10. If the log were scanned in the forward direction, A would be set to 20, which is incorrect.

If strict two-phase locking is used for concurrency control, locks held by a transaction T may be released only after the transaction has been rolled back as described. Once transaction T (that is being rolled back) has updated a data item, no other transaction could have updated the same data item. Therefore, restoring the old value of the data item will not erase the effects of any other transaction.

12.4.3 Checkpoints

We used checkpoints to reduce the number of log records that the system must scan when it recovers from a crash. Since we assumed no concurrency, it was necessary to consider only the following transactions during recovery:

- Those transactions that started after the most recent checkpoint
- The one transaction, if any, that was active at the time of the most recent check-point.

The situation is more complex when transactions can execute concurrently, since several transactions may have been active at the time of the most recent checkpoint. In a concurrent transaction-processing system, we require that the check point log record be of the form <checkpoint L>, where L is a list of transactions active at the time of the checkpoint. Again, we assume that

NOTES

transactions do not perform updates either on the buffer blocks or on the log while the checkpoint is in progress. The requirement that transactions must not perform any updates to buffer blocks or to the log during checkpointing can be bother some, since transaction processing will have to halt while a checkpoint is in progress. A fuzzy checkpoint is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out.

12.4.4 Restart Recovery

When the system recovers from a crash, it constructs two lists: The undo-list consists of transactions to be undone, and the redo-list consists of transactions to be redone.

The system constructs the two lists as follows: initially, they are both empty. The system scans the log backward, examining each record, until it finds the first <checkpoint>record:

For each record found of the form <T_i commit>, it adds T_i to redo-list.
For each record found of the form < T_i Start>, if T_t is not in redo-list, then it adds T_i to undo-list

When the system has examined all the appropriate log records, it checks the list L in the checkpoint record. For each transaction T_i in L, if T_i is not in redo-list then it adds 4 to the undo-list.

Once the redo-list and undo-list have been constructed, the recovery proceeds as follows:

1. The system rescans the log from the most recent record backward, and performs an undo for each log record that belongs to transaction T_i on the undo list. Log records of transactions on the redo-list are ignored in this phase. The scan stops when the < T_i start> records have been found for every transaction T_i in the undo-list.

2. The system locates the most recent<checkpoint L> record on the log. Notice that this step may involve scanning the log forward, if the checkpoint record was passed in step 1.

3. The system scans the log forward from the most recent<checkpointL> record, and performs redo for each log record that belongs to a transaction T_i that is on the redo-list. It ignores1ogrecordsof transactions on the undo-list in this phase.

It is important in step 1 to process the log backward, to ensure that the resulting state of the database is correct.

After the system has undone all transactions on the undo-list, it redoes those transactions on the redo-list. It is important, in this case, to process the log forward. When the recovery process has completed, transaction processing resumes.

It is important to undo the transaction in the undo-list before redoing transactions in the redo-list, using the algorithm in steps 1 to B; otherwise, a problem may occur. Suppose that data item A initiatty has the value 10.Suppose that a transaition I updated data item A to 20and aborted;transactionrollback would restoreA to the value 10.Supposethat another transactionQ. then updated data item,4 to 30 and committed, following which the system crashed.The stateof the log at the time of the crash is

<T_i, A, 10, 20>
<T_j, A, 10, 30>
< T_j commit>

If the redo pass is performed first, A wilt be set to 30; then, in the undo pass,A will be set to 10,which is wrong. The final value of Q should be 30, which we can ensure by performing undo before performing redo.

12.5 Buffer Management

In this section, we consider several subtle details that are essential to the implementation of a crash-recovery scheme that ensures data consistency and imposes a minimal amount of overhead on interactions with the database.

12.5.1 Log-Record Buffering

So far, we have assumed that every log record is output to stable storage at the time it is created. This assumption imposes a high overhead on system execution for several reasons: Typically, output to stable storage is in units of blocks. In most cases, a log record is much smaller than a block. Thus, the output of each log record translates to

The cost of outputting a block to stable storage is sufficiently high that it is desirable to output multiple log records at once. To do so, we write log records to a log buffer in main memory, where they stay temporarily until they are output to stable storage. Multiple log record scan be gathered in the log buffer, and output to stable storage in a single output operation. The order of log records in the stable storage must be exactly the same as the order in which they were written to the log buffer. As a result of log buffering, a log record may reside in only main memory (volatile storage) for a considerable

time before it is output to stable storage. Since such log records are lost if the system crashes, we must impose additional requirements on the recovery techniques to ensure transaction atomicity:
Transaction T_i enters the commit state after the <T*i* commit> log record has been output to stable storage.

- Before the <T_i commit> log record can be output to stable storage, all log records pertaining to transaction I must have been output to stable storage.
- Before a block of data in main memory can be output to the database(in nonvolatile storage),all log records pertaining to data in that block must have been output to stable storage.

This rule is called the write-ahead logging WAL) rule. (Strictly speaking, the WAL rule requires only that the undo information in the log have been output to stable storage, and permits the redo information to be written later. The difference is relevant in systems where undo information and redo information are stored in separate log records.) The three rules state situations in which certain log records must have been output to stable storage. There is no problem resulting from the output of log records earlier than necessary. Thus, when the system finds it necessary to output a log record to stable storage, it outputs an entire block of log records, if there are enough log records in main memory to fill a block. If there are insufficient log records to fill the block, all log records in main memory are combined into a partially full block, and are output to stable storage. Writing the buffered log to disk is sometimes referred to as a log force.

12.5.2 Database Buffering

We described the use of a two-level storage hierarchy. The system stores the database in non volatile storage (disk), and brings blocks of data into main memory as needed. Since main memory is typically much smaller than the entire database, it may be necessary to overwrite a block B_1 in main memory when another block B_2 needs to be brought into memory. If Br has been modified, B_1 must be output prior to the input of B_2.

The rules for the output of log records limit the freedom of the system to output blocks of data. If the input of block B_2 causes blocks B_1 to be chosen for output, all log records pertaining to data in .B_1 must be output to stable storage before B_1 is output. Thus, the sequence of actions by the system would be:

- Output log records to stable storage until all log records pertaining to block B_1 have been output.
- Output block B_1 to disk.
- Input block B_2 from disk to main memory.

It is important that no writes to the block B_1 be in progress while the system carries out this sequence of actions. We can ensure that there are no writes in progress by using a special means of locking: Before a transaction performs a write on a data item, it must acquire an exclusive lock on the block in which the data item resides. The lock can be released immediately after the update has been performed. Before a block is output, the system obtains an exclusive lock on the block, to ensure that no transaction is updating the block. It releases the lock once the block output has completed. Locks that are held for a short duration are often called **latches**. Latches are treated as distinct from locks used by the concurrency-control system. As a result, they may be released without regard to any locking protocol, such as two-phase locking, required by the concurrency-control system.

To illustrate the need for the write-ahead logging requirement, consider our banking example with transactions T_0 and T_1. Suppose that the state of the log is

<7_0 start>
<T_0, A, 1000, 950>

and that transactionT_0 issues a read(*B*).Assume that the block on which *B* resides is not in main memory, and that main memory is full. Suppose that the block on which *A* resides is chosen to be output to disk. If the system outputs this block to disk and then a crash occurs, the values in the database for accounts *A, B,* and *C* are *$950, $2000*, and *$700*, respectively. This database state is inconsistent. However, because of the WAL requirements, the log record

<Ts, A,1000, 950>

must be output to stable storage prior to output of the block on which *A* resides. The system can use the log record during recovery to bring the database back to a consistent state.

12.5.3 Operating System Role in Buffer Management

We can manage the database buffer by using one of two approaches:

1. The database system reserves part of main memory to serve as a buffer that it, rather than the operating system, manages.

This approach has the drawback of limiting flexibility in the use of main memory. The buffer must be kept small enough that other applications have sufficient main memory available for their needs. Flow ever, even when the other applications are not running, the database will not be able to make use of all the available memory. Likewise, non database applications may not use that part of main memory reserved for the database buffer, even if some of the pages in the database buffer are not being used.

NOTES

2. The database system implements its buffer within the virtual memory provided by the operating system. Since the operating system knows about the memory requirements of all processes in the system, ideally it should be in charge of deciding what buffer blocks must be force-output to disk, and when. The operating systems could not write out the database buffer pages itself, but instead should request the database system to force-output the buffer blocks. The database system in turn would force-output the buffer blocks to the database, after writing relevant log records to stable storage.

Unfortunately, almost all current-generation operating systems retain complete control of virtual memory. The operating system reserves space on disk for storing virtual-memory pages that are not currently in main memory; this space is called **swap space**. If the operating system decides to output a block B_x , that block is output to the swap space on disk, and there is no way for the database system to get control of the output of buffer blocks.

Therefore, if the database buffer is in virtual memory, transfers between database files and the buffer in virtual memory must be managed by the database system, which enforces the write-ahead logging requirements that we discussed.

This approach may result in extra output of data to disk. If a block B_x is output by the operating system, that block is not output to the database. Instead, it is output to the swap space for the operating system's virtual memory. When the database system needs to output B_x, the operating system may need first to input B_x from its swap space. Thus, instead of a single output of B_x, there may be two outputs of B, (one by the operating system, and one by the database system)and one extra input of B_x.

Although both approaches suffer from some drawbacks, one or the other must be chosen unless the operating system is designed to support the requirements of database logging. Only a few current operating systems, such as the Mach operating system, support these requirements.

12.6 Failure with loss of non-volatile storage

Failure with Loss of Nonvolatile Storage

Until now, we have considered only the case where a failure results in the loss of information residing in volatile storage while the content of the nonvolatile storage remains intact. Although failures in which the content of nonvolatile storage is lost are rare, we nevertheless need to be prepared to deal with this type of failure. In this section, we discuss only disk storage. Our discussions apply as well to other nonvolatile storage types.

The basic scheme is to dump the entire contents of the database to stable storage periodically-say, once per day. For example, we may dump the database to one or more magnetic tapes. If a failure occurs that results in the loss of physical database blocks, the system uses the most recent dump in restoring the database to a previous consistent state. Once this restoration has been accomplished, the system uses the log to bring the database system to the most recent consistent state.

More precisely, no transaction may be active during the dump procedure, and a procedure similar to checkpointing must take place:

1. Output all log records currently residing in main memory onto Stable storage.
2. Output all buffer blocks onto the disk.
3. Copy the contents of the database to stable storage.
4. Output a log record<dump> onto the stable storage.

Steps1, 2, and 4 correspond to the three steps used for checkpoints

To recover from the loss of nonvolatile storage, the system restores the database to disk by using the most recent dump. Then, it consults the log and redoes all the transactions that have committed since the most recent dump occurred. Notice that no undo operations need to be executed.

A dump of the database contents is also referred to as an archival dump, since we can archive the dumps and use them later to examine old states of the database. Dumps of a database and check pointing of buffers are similar.

The simple dump procedure described here is costly for the following two reasons. First, the entire database must be copied to stable storage, resulting in considerable data transfer. second, since transaction processing is halted during the dump procedure, CPU cycles are wasted. Fazzy dump schemes have been developed, which allow transactions to be active while the dump is in progress. They are similar to fuzzy-checkpointing schemes; see the bibliographical notes for more details.

12.7 Advance Recovery systems

Advanced Recovery Techniques

The recovery techniques described that once a transaction updates a data item, no other transaction may update the same data item until the first commits or is rolled back. We ensure the condition by using strict two-phase locking. Although strict two-phase locking is acceptable for records in relations, a

significant decrease in concurrency when applied to certain specialized structures, such as B+-tree index pages.

Several alternative recovery techniques, applicable even with early lock release, have been proposed. These schemes can be used in a variety of applications, not just for recovery of B^+-trees. We first describe an advanced recovery schemes supporting early lock release. We then outline the ARIES recovery scheme, which is widely used in industry. ARIES is more complex than our advanced recovery scheme, but incorporates a number of optimizations to minimize recovery time, and provides a number of other useful features.

Logical Undo Logging

For operations where locks are released early, we cannot perform the undo actions by simply writing back the old value of the data items. Consider a transaction T that inserts an entry into a B^+-tree, and, following the B+-tree concurrency-control protocol, releases some locks after the insertion operation completes, but before the transaction commits. After the locks are released, other transactions may perform further insertions or deletions, thereby causing further changes to the B+-tree nodes.

Even though the operation releases some locks early, It must retain enough locks to ensure that no other transaction is allowed to execute any conflicting operation (such as reading the inserted value or deleting the inserted value). For this reason/ the B+-tree concurrency - control protocol in Section16.9 holds locks on the leaf level of the B+-tree until the end of the transaction.

Now let us consider how to perform transaction rollback. If **physical undo** is used, that is, the old values of the internal B+-tree nodes (before the insertion operation was executed)are written back during transaction rollback, some of the updates performed by later insertion or deletion operations executed by other transactions could be lost. Instead, the insertion operation has to be undone by a **logical undo**-that is, in this case, by the execution of a delete operation.

Therefore, when the insertion operation completes, before it releases any locks, it writes a log record <T_i, O_j, operation-end, U), where the *U* denotes undo information and O_j denotes a unique identifier for (the instance of) the operation. For example, if the operation inserted an entry in a B+-tree, the undo information U would indicate that a deletion operation is to be performed, and would identify the B+-tree and what to delete from the tree. Such logging of information about operations is called logical logging. In contrast, logging of old-value and new-value information is called physical logging, and the corresponding log records are called physical log records. The insertion and deletion operations are examples of a class of operations that require logical undo operations since they release locks early; we call

such operations logical operations. Before a logical operation begins, it writes a log record <T_i,O_j, operation-begin>, where O_j is the unique identifier for the operation. While the system is executing the operation, it does physical logging in the normal fashion for all updates performed by the operation. Thus, the usual old-value and new-value information is written out for each update. When the operation finishes, it writes an operation-end log record as described earlier.

Transaction Rollback

First consider transaction rollback during normal operation (that is, not during recovery from system failure). The system scans the log backward and uses log records belonging to the transaction to restore the old values of data items. Unlike rollback in normal operation, however, rollback in our advanced recovery scheme writes out special redo-only log records of the form <T_i, X_j, *V*> containing the value v being restored to data item X_j during the rollback. These log records are sometimes called compensation log records. Such records do not need undo information, since we will never need to undo such an undo operation.

Whenever the system finds a log record <T_i,O_j, operation-end, U>, it takes special actions:

1. It rolls back the operation by using the undo information U in the log record. It logs the updates performed during the rollback of the operation just like updates performed when the operation was first executed .In other words, the system logs physical undo information for the updates performed during rollback, instead of using compensation log records. This is because a crash may occur while a logical undo is in progress, and on recovery the system has to complete the logical undo; to do so, restart recovery will undo the partial effects of the earlier undo, using the physical undo information.

At the end of the operation rollback, instead of generating a tog record <T_i, O_j, *operation-end, U*>, the database system generates a log record <T_i, O_i, operation-abort>.

2. When the backward scan of the log continues, the system skips all log records of the transaction until it finds the log record <T_i, O_j, operation-begin>. After it finds the operation-begin log record, it processes log records of the transaction in the normal manner again.

Observe that skipping over physical log records when the operation-endlog record is found during rollback ensures that the old values in the physical log record are not used for rollback, once the operation completes.

If the system finds a record <T_i,O_j, operation-abort>,it skips all preceding records until it finds the record <T_i,O_j, operation-begin>. These preceding log

records must be skipped to prevent multiple rollback of the same operation, in case there had been a crash during an earlier rollback, and the transaction had already been partly rolled back. When the transaction of; has been rolled back, the system adds a record <T_i abort> to the log.

If failures occur while a logical operation is in progress, the operation-end log record for the operation will not be found when the transaction is rolled back. However, for every update performed by the operation, undo information-in the form of the old value in the physical log records-is available in the log. The physical log records will be used to roll back the incomplete operation.

Check Your Progress

1. Define: Log
2. What is meant by Checkpoint?
3. What are concurrent transactions?

12.8 Remote Backup Systems

Remote Backup Systems

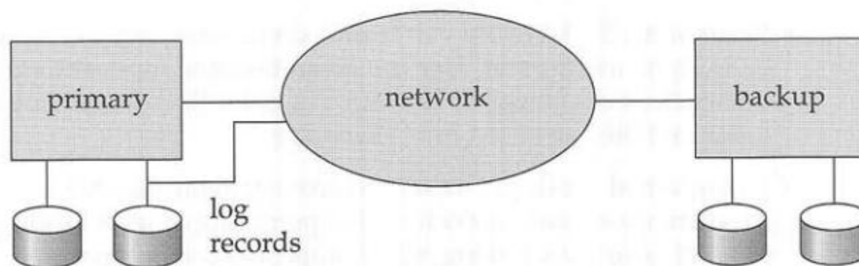
Traditional transaction-processing systems are centralized or client-server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes. Increasingly, there is a need for transaction-processing systems that can function in spite of system failures or environmental disasters. Such systems must provide high availability, that is, the time for which the system is unusable must be extremely small.

We can achieve high availability by performing transaction processing at one site, called the primary site, and having a remote backup site where all the data from the primary site are replicated. The remote backup site is sometimes also called the secondary site. The remote site must be kept synchronized with the primary site, as updates are performed at the primary. We achieve synchronization by sending all log records from primary site to the remote backup site. The remote backup site must be physically separated from the primary-for example, we can locate it in a different state-so that a disaster at the primary does not damage the remote backup site.

When the primary site fails, the remote backup site takes over processing. First, however, it performs recovery, using its (perhaps outdated) copy of the data from the primary, and the log records received from the primary. In effect, the remote backup site is performing recovery actions that would have been performed at the primary site when the latter recovered. Standard recovery algorithms, with minor modifications, can be used for recovery at the remote backup site. Once recovery has been performed, the remote backup site starts processing transactions.

Availability is greatly increased over a single-site system, since the system can recover even if all data at the primary site are lost. The performance of a remote backup system is better than the performance of a distributed system with two-phase commit.

Figure 12.1 Architecture of remote backup system



Several issues must be addressed in designing a remote backup system:

- **Detection of failure**. As in failure-handling protocols for distributed system,it is important for the remote backup system to detect when the primary has failed. Failure of communication lines can fool the remote backup into believing that the primary has failed. To avoid this problem, we maintain several communication links with independent modes of failure between the primary and the remote backup. For example, in addition to the network connection, there may be a separate modem connection over a telephone line, with services provided by different telecommunication companies. These connections may be backed up via manual intervention by operators, who can communicate over the telephone system.

- **Transfer of control**. When the primary fails, the backup site takes over processing and becomes the new primary. When the original primary site recovers, it can either play the role of remote backup, or take over the role of primary site again. In either case, the old primary must receive a log of updates carried out by the backup site while the old primary was down, The simplest way of transferring control is for the old primary to receive redo logs from the old backup site, and to catch up with the updates by applying them locally. The old primary can then act as a remote backup site. If control must be transferred back, the old backup site can pretend to have failed, resulting in the old primary taking over.

- **Time to recover**. If the log at the remote backup grows large, recovery will take a long time. The remote backup site can periodically process the redo log records that it has received, and can perform a checkpoint, so that earlier parts of the log can be deleted. The delay before the remote backup takes over can be significantly reduced as a result.

A **hot-spare** configuration can make takeover by the backup site almost instantaneous. In this configuration, the remote backup site continually processes redo log records as they arrive, applying the updates locally. As soon as the failure of the primary is detected, the backup site completes recovery by rolling back incomplete transactions; it is then ready to process new transactions.

- **Time to commit**. To ensure that the updates of a committed transaction are durable, a transaction must not be declared committed until its log records have reached the backup site. This delay can result in a longer wait to commit a transaction and some systems therefore permit lower degrees of durability.

The degrees of durability can be classified as follows.

- **One-safe.** A transaction commits as soon as its commit log record is written to stable storage at the primary site. The problem with this scheme is that the updates of a committed transaction may not have made it to the backup site, when the backup site takes over processing. Thus, the updates may appear to be lost. When the primary site recovers, the lost updates cannot be merged in directly, since the updates may conflict with later updates performed at the backup site. Thus, human intervention may be required to bring the database to a consistent state.

- **Two-very-safe**. A transaction commits as soon as its commit log record is written to stable storage at the primary and the backup site. The problem with this scheme is that transaction processing cannot proceed if either the primary or the backup site is down. Thus, availability is actually less than in the single-site case, although the probability of data loss is much less.

- **Two-safe**. This scheme is the same as two-very-safe if both primary and backup sites are active. If only the primary is active, the transaction is allowed to commit as soon as its commit log record is written to stable storage at the primary site.

This scheme provides better availability than does two-very-safe, while avoiding the problem of lost transactions faced by the one-safe scheme. It results in a slower commit than the one-safe scheme, but the benefits generally outweigh the cost. Several commercial shared-disk systems provide a level of fault tolerance that is intermediate between centralized and remote backup systems. In these commercial systems, the failure of a CPU does not result in system failure. Instead, other CPUs take over and they carry out recovery. Recovery actions include rollback of transactions running on the failed CPU, and recovery of locks held by those transactions. Since data are on a shared disk, there is no need for transfer of log records.

However, we should safe guard the data from disk failure by using, for example, a RAID disk organization.

An alternative way of achieving high availability is to use a distributed database, with data replicated at more than one site. Transactions are then required to update all replicated of any data item that they update.

12.9 Answers to Check Your Progress Questions

1. The most widely used structure for recording database modifications is the log
2. Checkpoint is a process that writes current in-memory dirty pages (modified pages) and transaction log records to physical disk.
3. In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions.

12.10 Summary

- The **log** is a sequence of log records, recording all the update activities in the database
- Using **checkpoints** to reduce the number of log records that the system must scan when it recovers from a crash.
- A database buffer is a temporary storage area in memory used to hold a copy of a **database block**.
- The basic scheme is to dump the entire content of the database to stable memory periodically.
- The insertion operation has to be undone by a **logical undo**-that is, in this case, by the execution of a delete operation.
- **Rollback** in advanced recovery scheme writes out special redo-only log records of the form <Ti, Xj, V>
- The performance of a **remote backup** system is better than the performance of a distributed system with **two-phase commit**.

12.11 Key Words

- **Transaction** identifier is the unique identifier of the transaction that performed the write operation.
- **Data-item** identifier is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- **Old value** is the' value of the data item prior to the write.
- **New value** is the value that the data item will have after the write.
- A **fuzzy** checkpoint is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out.
- Locks that are held for a short duration are often called **latches**

NOTES

- The operating system reserves space on disk for storing **virtual-memory** pages that are not currently in main memory; this space is called **swap space**.

12.12 Self-Assessment Questions and Exercises

Short Answer Questions:

1. Define: Log Based Recovery
2. What is meant by recovery with concurrent transactions?
3. Define: Checkpoints
4. What is called restart recovery?

Long Answer Questions:

1. Describe several fields of log based record
2. Explain about Buffer management
3. Describe the advanced recovery systems and remote backup systems

12.13 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011

UNIT XIII DATA ON EXTERNAL STORAGE

Structure

13.1 Introduction
13.2 Objectives
13.3 File Organization and Indexing
13.4 Cluster Indexes, Primary and Secondary Indexes
13.5 Index data Structures
13.6 Hash Based Indexing
13.7 Tree base Indexing
13.8 Comparison of File Organizations
13.9 Answers to Check Your Progress Questions
13.10 Summary
13.11 Key Words
13.12 Self-Assessment Questions and Exercises
13.13 Further Readings

13.1 Introduction

A database consists of a huge amount of data. The data is grouped within a table in RDBMS, and each table has related records. A user can see that the data is stored in form of tables, but in actual this huge amount of data is stored in physical memory in form of files. A file is named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tables and optical disks. File Organization refers to the logical relationships among various records that constitute the file, particularly with respect to the means of identification and access to any specific record. In simple terms, Storing the files in certain order is called file Organization. File Structure refers to the format of the label and data blocks and of any logical control record.

13.2 Objectives

After reading this chapter, you will be able to understand:

- File Organization and Indexing
- Indexes (Cluster Indexes, Primary and Secondary Indexes)
- Indexing (Hash Based, Tree based)
- Comparison of various File Organizations

13.3 File Organization and Indexing

File Organization and Indexing

A file is organized logically as a sequence of records. These records are mapped on to disk blocks. Files are provided as basic construction operating

NOTES

systems, so we shall assume the existence of an underlying file system. We need to consider ways of representing logical data models in terms of files.

Although blocks are of a fixed size determined by the physical properties of the disk and by the operating system, record sizes vary. In a relational database, tuples of distinct relations are generally of different sizes.

One approach to mapping the database to files is to use several files, and to store records of only one fixed length in any given file. An alternative is to structure our files so that we can accommodate multiple lengths for records; however, files of fixed length records are easier to implement than are files of variable-length records. Many of the techniques used for the former can be applied to the variable-length case. Thus, we begin by considering a file of fixed-length records.

Organization of Records in Files

Heap file organization. Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.

Sequential file organization. Records are stored in sequential order, according to the value of a "search key" of each record.

Hashing file organization. A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed.

Basic Concept of Indexing:

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by u word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information we are looking for. The words in the index are in sorted order, making it easy to find the word we are looking for. Moreover, the index is much smaller than the book, further reducing the effort needed to find the words we are looking for.

Database-system indices play the same role as book indices in libraries. For example, to retrieve an account record given the account number, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the account record.

Keeping a sorted list of account numbers would not work well on very large databases with millions of accounts, since the index would itself be very big; further even though keeping the index sorted reduces the search time, finding

an account can still be rather time-consuming. Instead, more sophisticated indexing techniques may be used.

There are two basic kinds of indices:

Ordered indices. Based on a sorted ordering of the values.

Hash indices. Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a hash function.

We shall consider several techniques for both ordered indexing and hashing. No one technique is the best. Rather, each technique is best suited to particular database applications. Each technique must be evaluated on the basis of these factors:

Access types: The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.

Access time: The time it takes to find a particular data item, or set of items, using the technique in question.

Insertion time: The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.

Deletion time: The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.

Space overhead: The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

13.4 Cluster Indexes, Primary and Secondary Indexes

Cluster Indexes, Primary and Secondary Indexes

To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it.
The records in the indexed file may themselves be stored in some sorted order, just as books in a library are stored according to some attribute such as

NOTES

the Dewey decimal number. A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a clustering index is an index whose

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Figure 13.1 Sequential file for account records

search key also defines the sequential order of the file. Clustering indices are also called primary indices; the term primary index seems to denote an index on a primary key, but such indices can in fact be built on any search key. The search key of a clustering index is often the primary key, although that is not necessarily so. Indices whose search key specifies an order different from the sequential order of the file are called nonclustering indices, or secondary indices. The terms "clustered, 'and "nonclustered" are often used in place of "clustering" and "nonclustering.,'

we assume that all files are ordered sequentially on some search key. Such files, with a clustering index on the search key, are called index-sequential files. They represent one of the oldest index schemes used. in database systems. They are designed for applications that require both sequential processing of the entire file and random access to individual records.

Figure 12.1 shows a sequential file of account records taken from our banking example. In the example of Figure 12.1, the records are stored in search-key order with branch_name used as the search key.

Secondary Indices

Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A clustering index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be

anywhere in the file and, in general, we cannot find them without searching the entire file.

A secondary index on a candidate key looks just like a dense clustering index, except that the records pointed to by successive values in the index are not stored sequentially. In general, however, secondary indices may have a different structure from clustering indices. If the search key of a clustering index is not a candidate key, it suffices if the index points to the first record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file.

In contrast, if the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value. The remaining records with the same search-key value could be anywhere in the file, since the records are ordered by the search key of the clustering index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.

We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file.

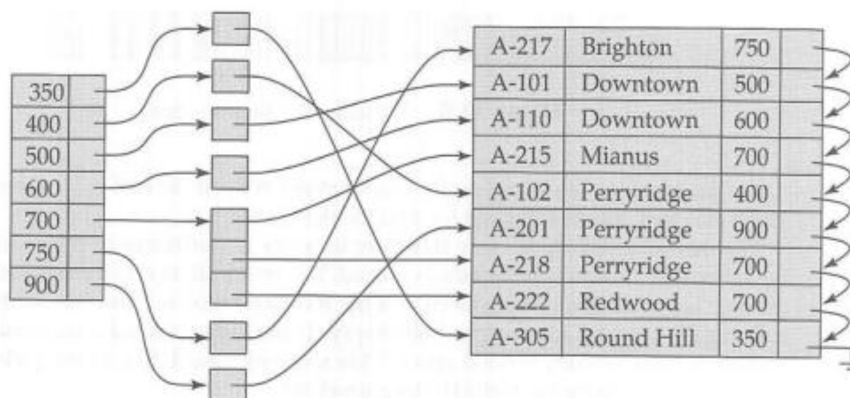


Figure 13.2 Secondary index on account file, on noncandidate key balance

A sequential scan in clustering index order is efficient because records in the file are stored physically in the same order as the index order. However, we cannot (except in rare special cases) store a file physically ordered by both the search key of the clustering index and the search key of a secondary index. Because secondary key order and physical-key order differ, if we attempt to scan the file sequentially in secondary-key order, the reading of each record is likely to require the reading of a new block from disk, which is very slow.

NOTES

The procedure described earlier for deletion and insertion can also be applied to secondary indices; the actions taken are those described for dense indices storing a pointer to every record in the file. If a file has multiple indices, whenever the file is modified, every index must be updated.

Secondary indices improve the performance of queries that use keys other than the search key of the clustering index. However, they impose a significant overhead on modification of the database. The designer of a database decides which secondary indices are desirable on the basis of an estimate of the relative frequency of queries and modifications

13.5 Index Data Structures

Index Definition in SQL

The SQL standard does not provide any way for the database user or administrator to control what indices are created and maintained in the database system. Indices are not required for correctness, since they are redundant data structures. However, indices are important for efficient processing of transactions, including both update transactions and queries. Indices are also important for efficient enforcement bf integrity constraints.

In principle, a database system can decide automatically what indices to create. Flow ever, because of the space cost of indices, as well as the effect of indices on update processing, it is not easy to automatically make the right choices about what indices to maintain. Therefore, most SQL-implementations provide the programmer control over creation and removal of indices via data-definition, language commands.

We illustrate the syntax of these commands next. Although the syntax that we show is widely used and supported by many database systems, it is not part of the SQL: 1999 standard. The SQL- standards (up to SQL: 1999, at least) do not support control of the physical database schema, and have restricted themselves to the logical database schema.

We create an index by the create index command, which takes the form

create index <index-name> on <relation-name> (<attribute-list>)

The *attribute-list* is the list of attributes of the relations that form the search key for the index.

To define an index name branch-index on the branch relation with branch_ name as the search key, we write

create ind,ex branch_indexon brnnch (branch,nqme)

If we wish to declare that the search key is a candidate key, we add the attribute **unique** to the index definition. Thus, the command

create unique index brsnchindexon branch(branch_nnme)

declares branch-name to be a candidate key for branch. If, at the time we enter the create unique index command, branch-name is not a candidate key, the system will display an error message, and the attempt to create the index will fail. If the index creation attempt succeeds, any subsequent attempt to insert a tuple that violates the key declaration will fail. Note that the unique feature is redundant if the database system supports the unique declaration of the SQL standard'

Many database systems also provide a way to specify the type of index to be used (such as B^+ -tree or hashing). Some database systems also permit one of the indices on a relation to be declared to be clustered; the system then stores the relation sorted by the search-key of the clustered index. The index name we specified for an index is required to drop an index. The drop index command takes the form:

drop index <index-name)

13.6 Hash Based Indexing

Hash Indices

Hashing can be used not only for file organization, but also for index-structure creation. A hash index organizes the search keys, with their associated pointers, into a hash file structure. We construct a hash index as follows. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).

The hash function in the figure computes the sum of the digits of the account number modulo 7. The hash index has seven buckets, each of size2 (realistic indices would, of course, have much larger bucket sizes).One of the buckets has three keys mapped to it, so it has an overflow bucket. In this example, account-number is a primary key for account, so each search key has only one associated pointer. In general, multiple pointers can be associated with each key.

Check Your Progress

1. Define: File Organization
2. What is meant by clustered file organization?
3. What is Heap file organization?

NOTES

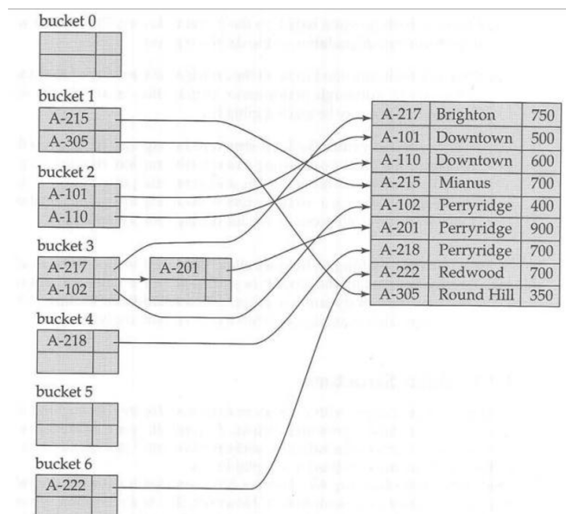


Figure 13.3 Hash index on search key account_number of account file

We use the term hash index to denote hash file structures as well as secondary hash indices. Strictly speaking, hash indices are only secondary index structures. A hash index is never needed as a clustering index structure, since, if a file itself is organized by hashing, there is no need for a separate hash index structure on it. However, since hash file organization provides the same direct access to records that indexing provides, we pretend that a file organized by hashing also has a clustering hash index on it.

Data Structure

Extend able hashing copes with changes in database size by splitting and coalescing buckets as the database grows and shrinks. As a result, space efficiency is retained. Moreover, since there organization is performed on only one bucket at a time, the resulting performance over head is acceptably low.

With extendable hashing, we choose a hash function h with the desirable properties of uniformity and randomness. However, this hash function generates values over a relatively large range-namely, b-bit binary integers. A typical value for b is 32.

We do not create a bucket for each hash value. Indeed, 232 is over 4 billion, and that many buckets is unreasonable for all but the largest databases. Instead, we create bucket son demand, as records are inserted into the file. We do not use the entire b bits of the hash value initially. At any point, we use I bits, where 0 < ? < b. These I bits are used as an offset into an additional table of bucket addresses. The value of I grows and shrinks with the size of the database.

The I appearing above the bucket address table in the figure indicates that I bits of the hash value h(K) are required to determine the correct bucket for K.

This number will, of course, change as the file grows. Although I bits are required to find the correct entry in the bucket address table, several consecutive table entries may point to the same bucket. All such entries will have a common hash prefix, but the length of this prefix may be less than l. Therefore, we associate with each bucket an integer giving the length of the common hash prefix.

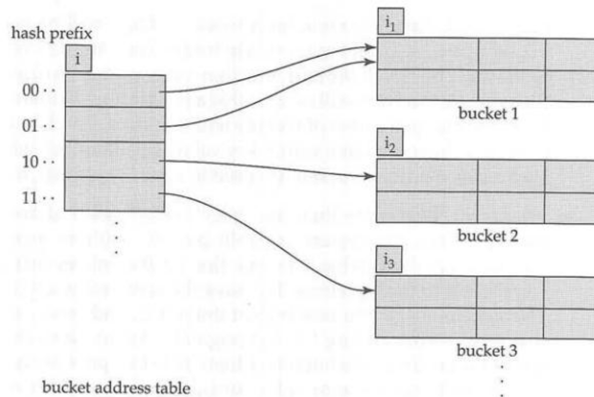


Figure 13.4 General extendable hash structure

13.7 Tree Base Indexing

B+ Tree Index Files

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.

The B^+ -tree index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B^+ -tree index takes the form of a balanced tree in which every path from the root of the tree to a

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

Typical node of a B^+ -tree

leaf of the tree is of the same length. Each non leaf node in the tree has between [n/2] and n children, where r is fixed for a particular tree.

We shall see that the B+-tree structure imposes performance overhead on insertion and deletion, and adds space overhead. The overhead is acceptable even for frequently modified files, since the cost of file reorganization is avoided. Furthermore, since nodes may be as much as half empty (if they have the minimum number of children), there is some wasted space. This space

overhead, too, is acceptable given the performance benefits of the B+-tree structure.

13.8 Comparison of File Organization and Indexes

There are various methods of file organizations. These methods may be efficient for certain types of access/selection meanwhile it will turn inefficient for other selections. Hence it is up to the programmer to decide the best suited file organization method depending on his requirement.

Some of the file organizations are

1. Sequential File Organization
2. Heap File Organization
3. Hash/Direct File Organization
4. Indexed Sequential Access Method
5. B+ Tree File Organization
6. Cluster File Organization

	Sequential	**Heap/ Direct**	**Hash**	**ISAM**	**B+ tree**	**Cluster**
Method of storing	Stored as they come or sorted as they come	Stored at the end of the file. But the address in the memory is random.	Stored at the hash address generated	Address index is appended to the record	Stored in a tree like structure	Frequently joined tables are clubbed into one file based on cluster key
Types	Pile file and sorted file Method		Static and dynamic hashing	Dense, Sparse, multilevel indexing		Indexed and Hash
Design	Simple Design	Simplest	Medium	Complex	Complex	Simple
Storage Cost	Cheap (magnetic tapes)	Cheap	Medium	Costlier	Costlier	Medium

13.9 Answers to Check Your Progress Questions

1. File Organization refers to the logical relationships among various records that constitute the file, particularly with respect to the means of identification and access to any specific record.
2. A file is organized so that the ordering of data records is the same as or close to the ordering of data entries in some index. Then that index is called clustered
3. In heap file organization, the records are inserted at the file's end. When the records are inserted, it doesn't require the sorting and ordering of records.

13.10 Summary

- A **file** is organized logically as a sequence of **records**.
- Based on a **sorted ordering** of the values are the **ordered indicies**
- Based on a **uniform distribution** of the values are the hast indicies
- A file may have several indices, on different **search keys**.
- Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering** indices, or secondary indices.
- A clustering index on the search key, are called **index-sequential** files.
- Secondary indices improve the performance of queries that use keys other than the search key of the **clustering index**.
- Hashing can be used not only for file organization, but also for **index-structure creation**.

13.11 Key Words

- The **File** is a collection of **records**. Using the primary key, we can access the records.
- Records are stored in sequential order, according to the value of a **search key** of each record.
- **Ordered indices**, Based on a sorted ordering of the values.
- **Secondary indices** must be dense, with an index entry for every search-key value, and a pointer to every record in the file.
- **Hashing** can be used not only for file organization, but also for index-structure creation.

13.12 Self-Assessment Questions and Exercises

Short Answer Questions:

1. What is called Indexing?
2. Define: Primary & Secondary Indexes.
3. What is meant by tree base indexing?
4. What is the index data structure?

Long Answer Questions:

1. Describe the Organization of Records in Files
2. Write short notes on
   a. Clustering Indexes
   b. Primary Indexes
   c. Secondary Indexes
3. Describe the comparision of file organizations

13.13 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011

UNIT XIV PERFORMANCE TUNING

Structure

14.1 Introduction
14.2 Objectives
14.3 Intuitions for tree Indexes
14.4 Indexed Sequential Access Methods (ISAM)
14.5 A Dynamic Index Structure
14.6 Answers to Check Your Progress Questions
14.7 Summary
14.8 Key Words
14.9 Self-Assessment Questions and Exercises
14.10 Further Readings

14.1 Introduction

Database tuning describes a group of activities used to optimize and homogenize the performance of a database. It usually overlaps with query tuning, but refers to design of the database files, selection of the database management system (DBMS) application, and configuration of the database's environment (operating system, CPU, etc.). Database tuning aims to maximize use of system resources to perform work as efficiently and rapidly. Most systems are designed to manage their use of system resources, but there is still much room to improve their efficiency by customizing their settings and configuration for the database and the DBMS. Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to what we see in books.

14.2 Objectives

This chapter helps to understand:

- Tree indexing
- Indexed Sequential Access Methods
- Dynamic index structure

14.3 Intuitions for tree Indexes

Index structure:

Indexes can be created using some database columns.

Search key	Data Reference

Figure 14.1 Structure of Index

- The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.
- The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

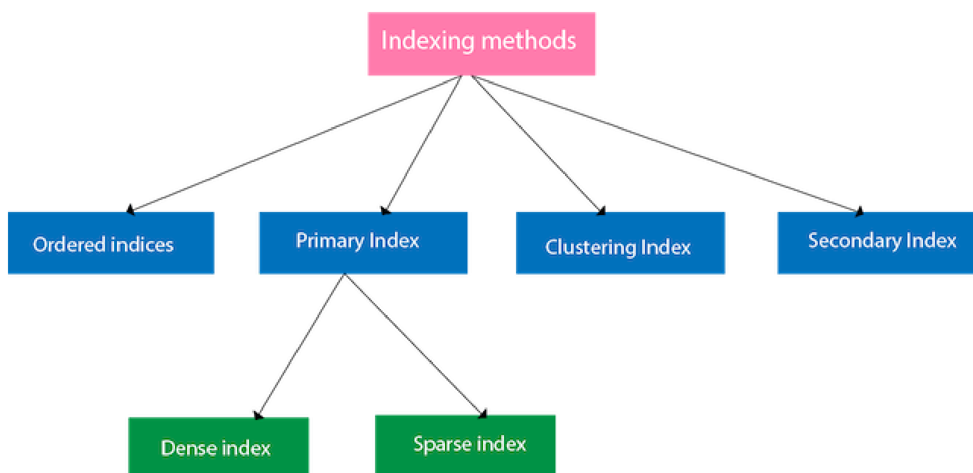


Figure 14.2 Indexing Methods

Ordered indices

The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.

Example: Suppose we have an employee table with thousands of record and each of which is 10 bytes long. If their IDs start with 1, 2, 3....and so on and we have to search student with ID-543.

- In the case of a database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading 543*10=5430 bytes.
- In the case of an index, we will search using indexes and the DBMS will read the record after reading 542*2= 1084 bytes which are very less compared to the previous case.

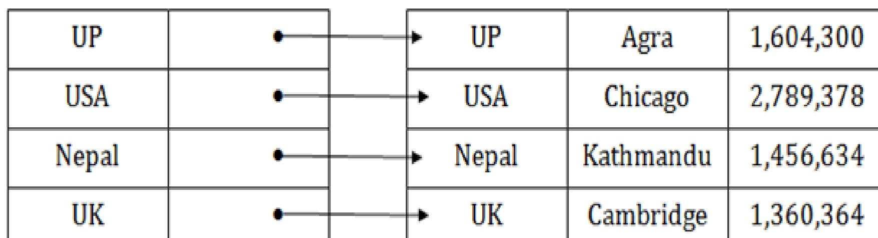
Primary Index

- If the index is created on the basis of the primary key of the table, then it is known as primary indexing. These primary keys are unique to each record and contain 1:1 relation between the records.
- As primary keys are stored in sorted order, the performance of the searching operation is quite efficient.
- The primary index can be classified into two types: Dense index and Sparse index.

NOTES

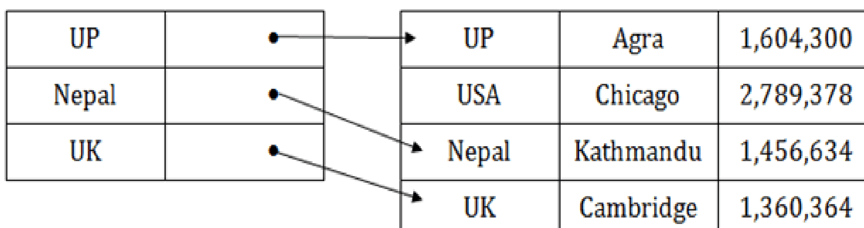
Dense index

- The dense index contains an index record for every search key value in the data file. It makes searching faster.
- In this, the number of records in the index table is same as the number of records in the main table.
- It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.



Sparse index

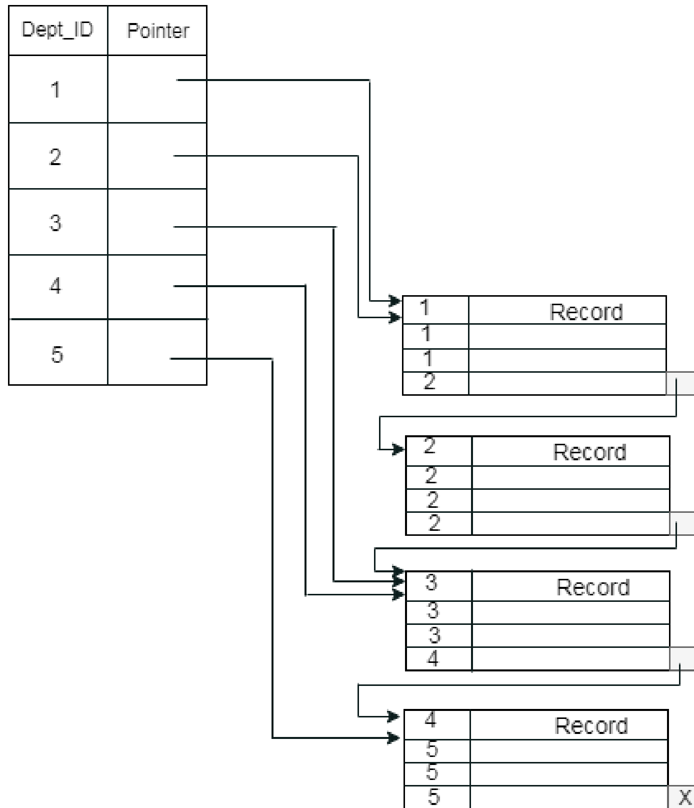
- In the data file, index record appears only for a few items. Each item points to a block.
- In this, instead of pointing to each record in the main table, the index points to the records in the main table in a gap.



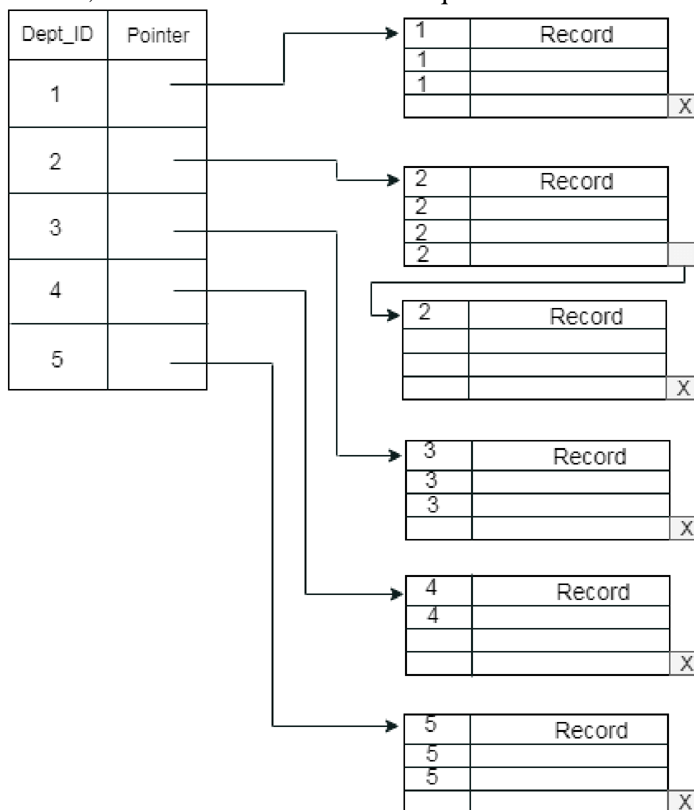
Clustering Index

- A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.
- In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
- The records which have similar characteristics are grouped, and indexes are created for these group.

Example: suppose a company contains several employees in each department. Suppose we use a clustering index, where all employees which belong to the same Dept_ID are considered within a single cluster, and index pointers point to the cluster as a whole. Here Dept_Id is a non-unique key.



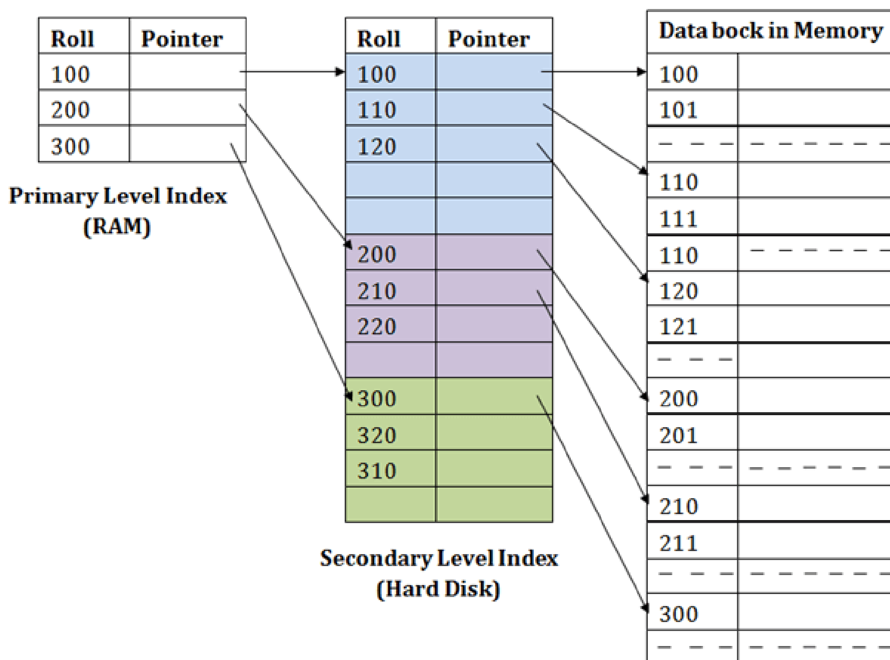
The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate clusters, then it is called better technique.



Secondary Index

In the sparse indexing, as the size of the table grows, the size of mapping also grows. These mappings are usually kept in the primary memory so that address fetch should be faster. Then the secondary memory searches the actual data based on the address got from mapping. If the mapping size grows then fetching the address itself becomes slower. In this case, the sparse index will not be efficient. To overcome this problem, secondary indexing is introduced.

In secondary indexing, to reduce the size of mapping, another level of indexing is introduced. In this method, the huge range for the columns is selected initially so that the mapping size of the first level becomes small. Then each range is further divided into smaller ranges. The mapping of the first level is stored in the primary memory, so that address fetch is faster. The mapping of the second level and actual data are stored in the secondary memory (hard disk).



For example:

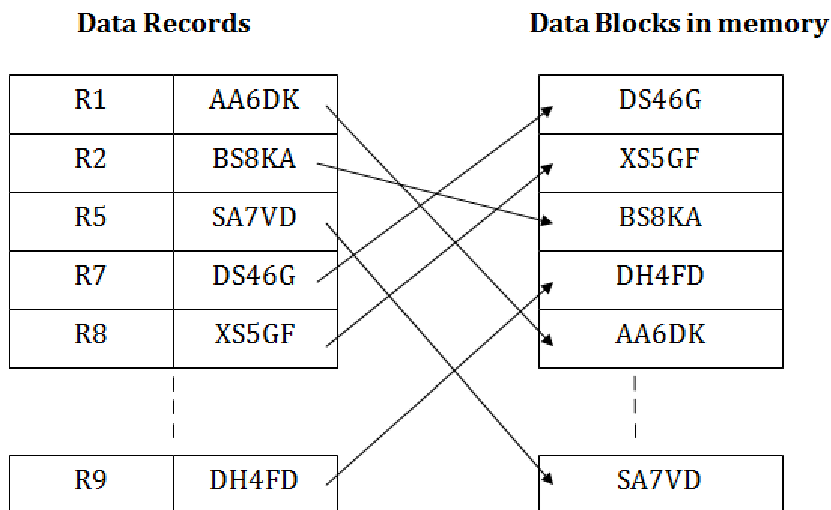
- If you want to find the record of roll 111 in the diagram, then it will search the highest entry which is smaller than or equal to 111 in the first level index. It will get 100 at this level.
- Then in the second index level, again it does max (111) <= 111 and gets 110. Now using the address 110, it goes to the data block and starts searching each record till it gets 111.
- This is how a search is performed in this method. Inserting, updating or deleting is also done in the same manner.

Check Your Progress

1. Define: Ordered Indices.
2. What is called as Dense index?

14.4 Indexed Sequential Access Methods (ISAM)

ISAM method is an advanced sequential file organization. In this method, records are stored in the file using the primary key. An index value is generated for each primary key and mapped with the record. This index contains the address of the record in the file.



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.

Pros of ISAM:

- In this method, each record has the address of its data block, searching a record in a huge database is quick and easy.
- This method supports range retrieval and partial retrieval of records. Since the index is based on the primary key values, we can retrieve the data for the given range of value. In the same way, the partial value can also be easily searched, i.e., the student name starting with 'JA' can be easily searched.

Cons of ISAM

- This method requires extra space in the disk to store the index value.
- When the new records are inserted, then these files have to be reconstructed to maintain the sequence.
- When the record is deleted, then the space used by it needs to be released. Otherwise, the performance of the database will slow down.

NOTES

14.5 B+ Trees: A Dynamic Index Structure

B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B+ tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them. Moreover, the leaf nodes are linked to provide ordered access to the records. The leaf nodes, therefore form the first level of index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.

From the above discussion it is apparent that a B+ tree, unlike a B-tree has two orders, 'a' and 'b', one for the internal nodes and the other for the external (or leaf) nodes.

The structure of the internal nodes of a B+ tree of order 'a' is as follows:

1. Each internal node is of the form : $<P_1, K_1, P_2, K_2, \ldots, P_{c-1}, K_{c-1}, P_c>$ where $c <= a$ and each **P_i is a tree pointer (i.e points to another node of the tree)** and, each **K_i is a key value** (see diagram-I for reference).
2. Every internal node has : $K_1 < K_2 < \ldots < K_{c-1}$
3. For each search field values 'X' in the sub-tree pointed at by P_i, the following condition holds : $K_{i-1} < X <= K_i$, for $1 < i < c$ and, $K_{i-1} < X$, for $i = c$ (See diagram I for reference)
4. Each internal nodes has at most 'a' tree pointers.
5. The root node has, at least two tree pointers, while the other internal nodes have at least \ceil(a/2) tree pointers each.
6. If any internal node has 'c' pointers, c <= a, then it has 'c – 1' key values.

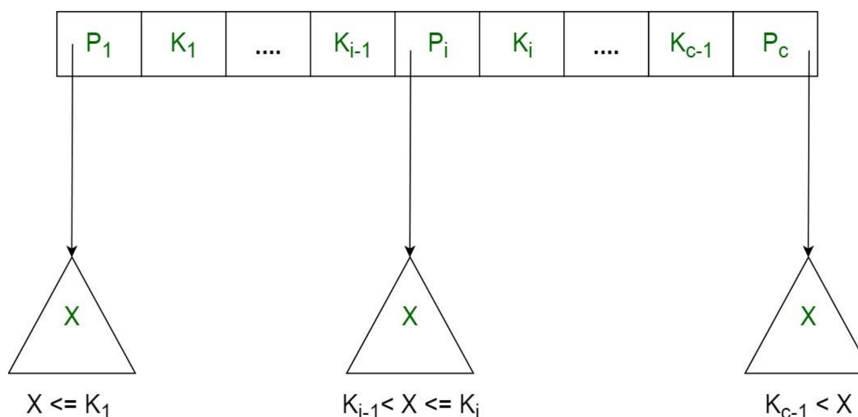


Diagram-I

The structure of the leaf nodes of a B+ tree of order 'b' is as follows:

1. Each leaf node is of the form : $<<K_1, D_1>, <K_2, D_2>, \ldots, <K_{c-1}, D_{c-1}>, P_{next}>$ where c <= b and each **D_i is a data pointer (i.e points to actual record in the disk whose key value is K_i or to a disk file block containing that record)** and, each **K_i is a key value** and, **P_{next} points to next leaf node in the B+ tree** (see diagram II for reference).
2. Every leaf node has : $K_1 < K_2 < \ldots < K_{c-1}$, c <= b
3. Each leaf node has at least \ceil(b/2) values.
4. All leaf nodes are at same level.
5.

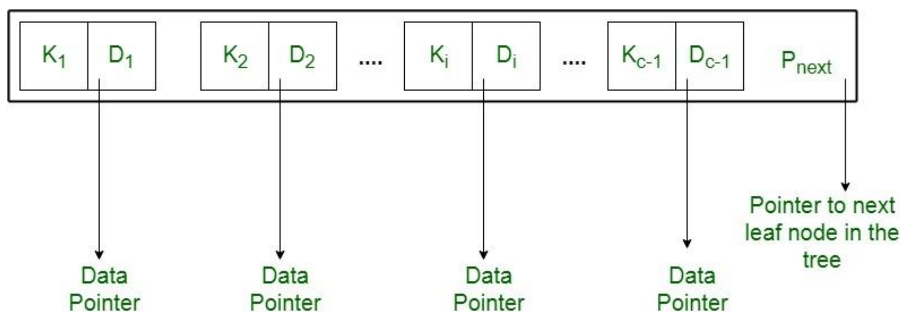
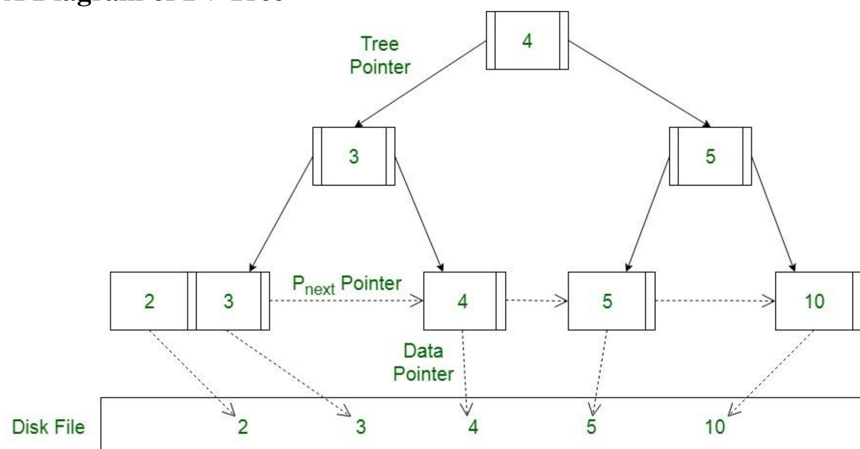


Diagram-II

Using the P_{next} pointer it is viable to traverse all the leaf nodes, just like a linked list, thereby achieving ordered access to the records stored in the disk.

A Diagram of B+ Tree –



14.6 Answers to Check Your Progress Questions

1. The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.
2. The dense index contains an index record for every search key value in the data file. It makes searching faster.

14.7 Summary

- Indexes can be created using some database columns.
- The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.
- In secondary indexing, to reduce the size of mapping, another level of indexing is introduced. In this method, the huge range for the columns is selected initially so that the mapping size of the first level becomes small.
- ISAM method is an advanced sequential file organization. In this method, records are stored in the file using the primary key.
- A B-tree has two orders, 'a' and 'b', one for the internal nodes and the other for the external (or leaf) nodes.

14.8 Key Words

- The index is created on the basis of the primary key of the table, then it is known as primary indexing.
- The dense index contains an index record for every search key value in the data file. It makes searching faster.
- A clustered index can be defined as an ordered data file.
- Secondary indexing is used to reduce the size of mapping
- An index value is generated for each primary key and mapped with the record in the ISAM method.

14.9 Self-Assessment Questions and Exercises

Short Answer Questions:

1. Define: Index
2. What are ordered indices?
3. What is called as secondary index?
4. What is meant by ISAM?

Long Answer Questions:

1. Describe the various indexing methods
2. Explain the advantages and disadvantages of ISAM?
3. Describe the concept of B+ Tree structure.

14.10 Further Readings

- Raghurama Krishnan, Johannes Gehrke, Data base Management Systems, 3rd Edition, TATA McGrawHill.2003.
- Silberschatz, Korth, Data base System Concepts, 6th Edition, Tata McGraw Hill, 2011

MODEL QUESTION PAPER

DISTANCE EDUCATION

BACHELOR OF COMPUTER APPLICATIONS (B.C.A) EXAMINATION

RELATIONAL DATABASE MANAGEMENT SYSTEM
Second Year - Third Semester
(CBCS – 2018-19 Academic Year Onwards)
Time : 3 hours **Max Marks :75**

PART - A (10 x 2=20 Marks)
Answer all questions.

1. Define: Data
2. What are the operations to be done with DML?
3. What is an entity?
4. What is Selection operation with respect to relational algebra?
5. State the syntax to create a table.
6. Define: Transitive dependency.
7. Define: Transaction.
8. What is meant by remote backup system?
9. Define: cluster index.
10. What are ordered indices?

PART - B (5 x 5 Marks = 25 Marks)
Answer all questions choosing either (a) or (b)

11.a). Discuss the levels of abstraction?
OR
11. b). State the characteristics of the Database users.

12.a). What are the integrity constraints? Explain the same.
OR
12. b). Write a note on TRC.

13.a). Write a note on nested queries.
OR
13. b). Discuss about lossless join decomposition.

14.a). Explain the properties of Transaction.
OR
14. b). Write a note on log based recovery techniques.

15.a). Explain the characteristics of various indexes.
OR
15. b). Write a note on B+ tree indexes.

Part – C (3 x 10 = 30 Marks)
Answer any three questions.

16. Describe the architecture of Database System
17. Explain the various relational algebraic operations.
18. Elaborate on various Normal Forms.
19. Explain the working of Lock-based protocols.
20. Describe about ISAM.

****