Database Management Systems

BSc IT
Semester
III

Mumbai University

By: munotes.in

Revision 2023

CONTENT

*****

Syllabus

F.Y.B.Sc. (Information Technology)		**Semester – III**	
Course Name: Database Management Systems		**Course Code: USIT304**	
Periods per week (1 Period is 50 minutes)		**5**	
Credits		**2**	
		Hours	**Marks**
Evaluation System	**Theory**	**2½**	**75**
	Examination	**-**	**25**

Unit	Details	Lectures
I	**Introduction to Databases and Transactions** What is database system, purpose of database system, view of data, relational databases, database architecture, transaction management **Data Models** The importance of data models, Basic building blocks, Business rules, The evolution of data models, Degrees of data abstraction. **Database Design, ER Diagram and Unified Modeling Language** Database design and ER Model: overview, ER Model, Constraints, ER Diagrams, ERD Issues, weak entity sets, Codd's rules, Relational Schemas, Introduction to UML	**12**
II	**Relational database model:** Logical view of data, keys, integrity rules, Relational Database design: features of good relational database design, atomic domain and Normalization (1NF, 2NF, 3NF, BCNF). **Relational Algebra and Calculus** Relational algebra: introduction, Selection and projection, set operations, renaming, Joins, Division, syntax, semantics. Operators, grouping and ungrouping, relational comparison. **Calculus:** Tuple relational calculus, Domain relational Calculus, calculus vs algebra, computational capabilities	**12**
III	**Constraints, Views and SQL** Constraints, types of constrains, Integrity constraints, Views: Introduction to views, data independence, security, updates on views, comparison between tables and views SQL: data definition, aggregate function, Null Values, nested sub queries, Joined relations. Triggers.	**12**

IV	**Transaction management and Concurrency** Control Transaction management: ACID properties, serializability and concurrency control, Lock based concurrency control (2PL, Deadlocks), Time stamping methods, optimistic methods, database recovery management.	**12**
V	**PL-SQL**: Beginning with PL / SQL, Identifiers and Keywords, Operators, Expressions, Sequences, Control Structures, Cursors and Transaction, Collections and composite data types, Procedures and Functions, Exceptions Handling, Packages, With Clause and Hierarchical Retrieval, Triggers.	**12**

Books and References:

Sr. No.	**Title**	**Author/s**	**Publisher**	**Edition**	**Year**
1.	Database System and Concepts	A Silberschatz Sudarshan	McGrawHill	Fifth Edition	
2.	Database Systems	Rob Coronel	Cengage Learning	Twelth Edition	
3.	Programming with PL/SQL for Beginners	H. Dand, R. Patil and T. Sambare	X –Team	First	2011
4.	Introduction to Database System	C.J.Date	Pearson	First	2003

*****

UNIT I

1

INTRODUCTION TO DATABASE AND TRANSACTION

Unit structure

1.0 OBJECTIVES

The objective of the chapter is as follow

- To get familiar with core component of database management systems
- To understand the different architecture of database
- To understand the concept of relational database

1.1 INTRODUCTION

- A database management system (DBMS) is a collection of interrelated data and a set of programs to access those data.
- The collection of data, usually referred to as the database, contains information relevant to an enterprise.
- The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient
- Main purpose if to manage large bodies of information in a structured format

1.2 DATABASE SYSTEM APPLICATIONS

Databases are widely used many application. Some of them are mentioned below

- **Banking:** For customer information, accounts, and loans, and banking transactions.
- **Airlines:** For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner – terminals situated around the world accessed the central database system through phone lines and other data networks.
- **Universities:** For student information, course registrations, and grades.
- **Credit card transactions:** For purchases on credit cards and generation of monthly statements.
- **Telecommunication:** For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.
- **Finance:** For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds.
- **Sales:** For customer, product, and purchase information.

1.3 PURPOSE OF DATABASE SYSTEM

1.3.1 Data redundancy and inconsistency:

- Files and application programs created by different programmers have different structures and written using different programming languages.
- The same information may be duplicated in several places (files).
- This redundancy leads to higher storage and access cost.
- In addition, it may lead to data inconsistency; that is, the various copies of the same data may no longer agree. For example, a changed student address may be reflected in the one department records but not elsewhere in the system.

1.3.2 Difficulty in accessing data: The conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

1.3.3 Data isolation: Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult

1.3.4 Integrity problems: The data values stored in the database must satisfy certain types of consistency constraints

1.3.5 Atomicity problems: A computer system, like any other device, is subject to failure. Atomic—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system

1.3.6 Security problems: Not every user of the database system should be able to access all the data

1.4 VIEW OF DATA

A database system is a collection of interrelated files and a set of programs that allow users to access and modify these files. 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.

1.4.1 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 users of databasesystems are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users interactions with the system:

Physical level: 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: 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.

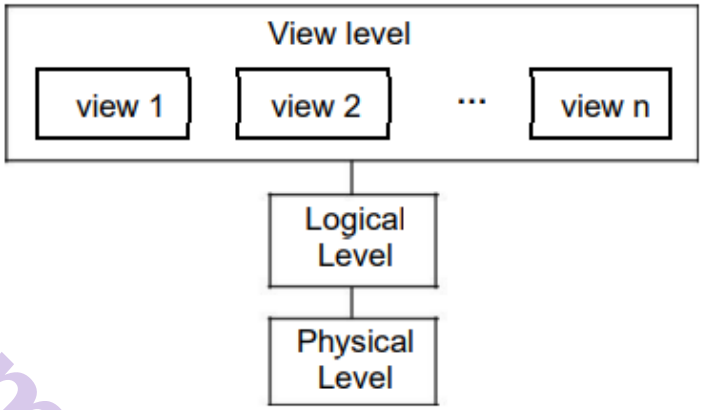


Fig. 1: The three levels of data abstraction.

1.4.2 Instances and Schema:

Databases change over time as information is inserted and deleted.

Instances:

- The collection of information stored in the database at a particular moment is called an **instance** of the database**.**
- The values of the variables in a program at a point in time correspond to an instance of a database schema

Schema:

- The overall design of the database is called the database **schema.**
- Schemas are changed infrequently.
- 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.

1.5 RELATIONAL DATABASES

A relational database is based on the relational model and uses a collection of tables to represent both data and the relationship among those data. It also includes a DML and DDL.

1.5.1 Tables:

- The relational model is an example of a record-based model.
- Record-based models are so named because the database is structured in fixed-format records of several types.
- Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes.
- It is possible to create schemas in the relational model that have problems such as unnecessarily duplicated information. The relational model hides such low-level implementation details from database developers and users
- The columns of the table correspond to the attributes of the record type.
- The relational model hides such low-level implementation details from database developers and users.

ID	name	dept_name	salary
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32143	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
3o456	Gold	Physic s	87000
76543	Singh	Finance	80000

(a) The instructor table.

dept_name	building	budget
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	35000
Music	Packard	80000
Finance	Painter	120000
History	Painter	130000
Phyics	Watson	70000

(b) The department table

Fig. 2 : A sample relational database.

1.6 DATA-MANIPULATION LANGUAGE

The SQL query language is nonprocedural. A query takes as input several tables (possibly only one) and always returns a single table. Here is an example of an SQL query that finds the names of all instructors in the History department:

select instructor.name
from instructor
where instructor.dept_name = 'History'

The query specifies that those rows from the table instructor where the dept nameis

History must be retrieved, and the name attribute of these rows must be displayed.

1.7 DATA-DEFINITION LANGUAGE

SQL provides a rich DDL that allows one to define tables, integrity constraints, assertions, etc.

For instance, the following SQL DDL statement defines the department table:

create table department
(depLnamechar(20),
building char (15),
budget numeric (12,2))

Execution of the above DDL statement creates the department table with three columns :dept_name, building, and budget, each of which has a specific data type associated with it. In addition, the DDL statement updates the data dictionary, which contains metadata. The schema of a table is an example of metadata

1.8 DATABASE ACCESS FROM APPLICATION PROGRAMS

SQL is not as powerful as a universal Turing machine; that is, there are some computations that are possible using a general-purpose programming language but are not possible using SQL. SQL also does not support actions such as input users, output to displays, or communication over the network. Such computations are supposed to be written in host language such as C, C++ or java with embedded SQL queries that access the data in the database. Application programs are programs that are used to interact with the database in this fashion

- To access the database?
- DML statements need to be executed from the host language. There are two ways to do this:
- By providing an application program interface (set of procedures) that can be used to send DML and DDL statements to the database and retrieve the results.
- The Open Database Connectivity (ODBC) standard for use with the C language is a commonly used application program interface standard. The Java Database Connectivity ODBC) standard provides corresponding features to the Java language.
- By extending the host language syntax to embed DML calls within the host language program. Usually a special character prefaces DML calls, and a preprocessor, called the DML precompiler, converts the DML statements to normal procedure calls in the host language.

1.9 DATABASE DESIGN

- Database systems are designed to manage large bodies of information. These large bodies of information do not exist in isolation. They are part of the operation of some enterprise whose end product may be information from the database or may be some device or service for which the database plays only a supporting role.
- Database design mainly involves the design of the database schema. The design of a complete database application environment that meets the needs of the enterprise being modelled requires attention to a broader set of issues.

1.9.1 Design Process:

- A high-level data model provides the database designer with a conceptual framework in which to specify the data requirements of the database users, and how the database will be structured to fulfill these requirements.
- The initial phase of database design, then, is to characterize fully the data needs of the prospective database users. The database designer needs to interact extensively with domain experts and users to carry out this task. The outcome of this phase is a specification of user requirements.
- The designer chooses a data model, and by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this conceptual-design phase provides a detailed overview of the enterprise.
- In a specification of functional requirements, users describe the kinds of operations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. At this stage of conceptual design, the designer can review the schema to ensure it meets functional requirements.
- In the logical-design phase, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The designer uses the resulting system-specific database schema in the subsequent physical-design phase, in which the physical features of the database are specified.

1.10 DATABASEARCHITECTURE

Database System Structure:

- 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.
- Most users of a database system today are not present at the site of the database system, but connect to it through a network.
- This can be differentiated between client machines, on which remote database users work, and server machines, on which the database system runs
- Database applications are usually partitioned into two or three parts,
- In a 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 a 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.

Application Architectures

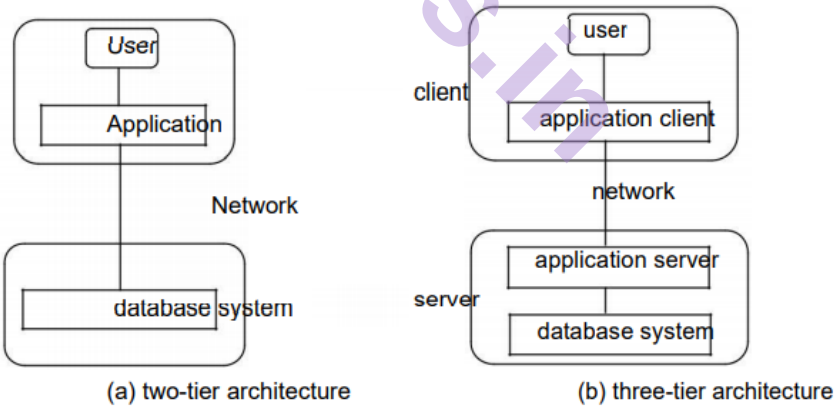


Fig. 2 : Two–tier and three-tier architectures.

1.11 TRANSACTION MANAGEMENT

1.11.1 Atomicity:

- Several operations on the database form a single logical unit of work.

- Consider an example of funds transfer, in which one department account(say A) is debited and another department account (say B) is credited.
- It is essential that either both the credit and debit occur, or that neither occur.
- This all-or-none requirement is called atomicity.

1.11.2 Consistency:

- In addition, it is essential that the execution of the funds transfer preserve the consistency of the database. That is, the value of the sum of the balances of A and B must be preserved. This correctness requirement is called consistency.

1.11.3 Durability:

- Finally, after the successful execution of a funds transfer, the new values of the balances of accounts A and B must persist, despite the possibility of system failure. This persistence requirement is called durability.
- 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 data base consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates.

1.11.4 Recovery Manager:

- Ensuring the atomicity and durability properties is the responsibility of the database system itself specifically, of the recovery manager.

1.11.5 Failure recovery:

- The database must be restored to the state in which it was before the transaction in question started executing. The database system must therefore perform failure recovery, that is, detect system failures and restore the database to the state that existed prior to the occurrence of the failure.

1.11.6 Concurrency-control manager:

- When several transactions update the database concurrently, the consistency of data may no longer be preserved, even though each individual transaction is correct. It is the responsibility of the concurrency-control manager to control the interaction among the concurrent transactions, to ensure the consistency of the database.
- The transaction manager consists of the concurrency-control manager and the recovery manager

1.12 LET US SUM UP

- A database-management system (DBMS) consists of a collection of interrelated data and a collection of programs to access that data. The data describe one particular enterprise.
- The primary goal of a DBMS is to provide an environment that is both convenient and efficient for people to use in retrieving and storing information
- A data-manipulation language (DML) is a language that enables users to access or manipulate data. Nonprocedural DMLs, which require a user to specify only what data are needed, without specifying exactly how to get those data, are widely used today.
- A data-definition language (DDL) is a language for specifying the database schema and as well as other properties of the data
- Transaction management ensures that the database remains in a consistent (correct) state despite system failures. The transaction manager ensures that concurrent transaction executions proceed without conflicting.

1.13 LIST OF REFERENCE

A Silberschatz, H Korth, S Sudarshan, "Database System and Concepts", fifth Edition McGraw- Hill

1.14 BIBLIOGRAPHY

- Database Systems ,RobCoronel,Cengage Learning.
- Programming with PL/SQL for Beginners, H. Dand, R. Patil and T. Sambare,X-Team.
- Introduction to Database System,C.J.Date,Pearson

1.15 UNIT END EXERCISE

1. Define database system and explain the purpose of database system in detail
2. Mention the views of database.
3. Explain DML and DDL in detail
4. Write a note on transaction management systems
5. Explain database architecture in detail.

*****

2

DATA MODELS

Unit Structure

2.0 OBJECTIVES

The objective of the chapter is as follow

- To get familiar with core data models in database management systems
- To understand the different types of database models
- To understand the concept of basic building blocks and business rules.

2.1 INTRODUCTION

- Data model gives an idea of how the final system or software will look after when the development is completed

- This concept is exactly like real world modelling in which before constructing any project (Buildings, Bridges, Towers) engineers create a model for it and gives the idea of how a project will look like after construction
- A data model is an overview of a software system which describes how data can be represented and accessed from software system after its complete implementation.

2.2 DATA MODELS

- Data models define data elements and relationships among various data elements for a specified system
- A data model is a way of finding tool for both business and IT professionals which uses a set of symbols and text to precisely explain a subset of real information to improve communication within the organisation and thereby lead to more flexible and stable application environment
- Data model is a simple abstraction of complex real world data gathering environment

2.2.1 Importance of Data Models:

- A data model is a set of concepts that can be used to describe the structure of data in a database.
- Data models are used to support the development of information systems by providing the definition and format of data to be involved in future system.
- Data model is acting like a guideline for development also gives an idea about possible alternatives to achieve targeted solution.
- A data model can sometimes be referred to as data structure especially in the context of programming languages.

2.2.2 Advantages of Data Models:

- Data model prevents the system from future risk and failure by defining structure of data in advance.
- As we got an idea of final system at the beginning of development itself so we can reduce the cost of project by proper planning and cost estimation as actual system is not yet developed.
- Data repetition and data type compatibility can be checked and removed with help of data model.
- We can improve Graphical User Interface (GUI) of system by making its model and get it approved by its future user so it will be simple for them to operate system and make entire system effective.

2.3 FILE MANAGEMENT SYSTEMS

- All data were permanently stored on a computer system, such as payroll and accounting records, was stored in individual files.
- A file management system, usually provided by the computer manufacturer as part of the computer's operating system, kept track of the names and locations of the files.
- The file management system basically had no data model; it knew nothing about the internal contents of files.
- To the file management system, a file containing a word processing document and a file containing payroll data appeared the same.
- Knowledge about the contents of a file which data is contained and how the data was organized was embedded in the application programs that used the file
- The problems of maintaining large file-based systems led in the late 1960s to the development of database management systems.
- The idea behind these systems was simple: take the definition of a file's content and structure out of the individual programs, and store it, together with the data, in a database.
- Using the information in the database, the DBMS that controlled it could take a much more active role in managing both the data and changes to the database structure.

2.3.1 Hierarchical Databases:

- This was developed by joined efforts of IBM and North American Rockwell known as Information management system.
- It was the first DBMS model
- The data is sorted hierarchically, either in top down or bottom up approach of designing.
- This model uses pointers to navigate between stored data.
- This model represents data as a hierarchical tree.
- Let us consider simple organizational structure as given below.
- CEO is root node having DU Heads below that managers who manages multiple project lead who organizes developer as shown below
- CEO -> Program manager (PM) -> Project Manager (PrM) -> Project Leader (PL) -> Team Leader (TL) -> Developer

Advantage:

Conceptual Simplicity:

Relationships between various levels is logically very simple. Hence database structure becomes easier to view.

Database Security:

Security is given by DBMS system itself it does not depends on whether programmer has given security or not.

Simple Creation, Updation and Access:

This model is simple to construct with help of pointers or similar concepts and very simple to understand also adding and deleting record is easy tin tree structure using pointers. This file system is faster and easy data retrieval through higher level records in tree structure.

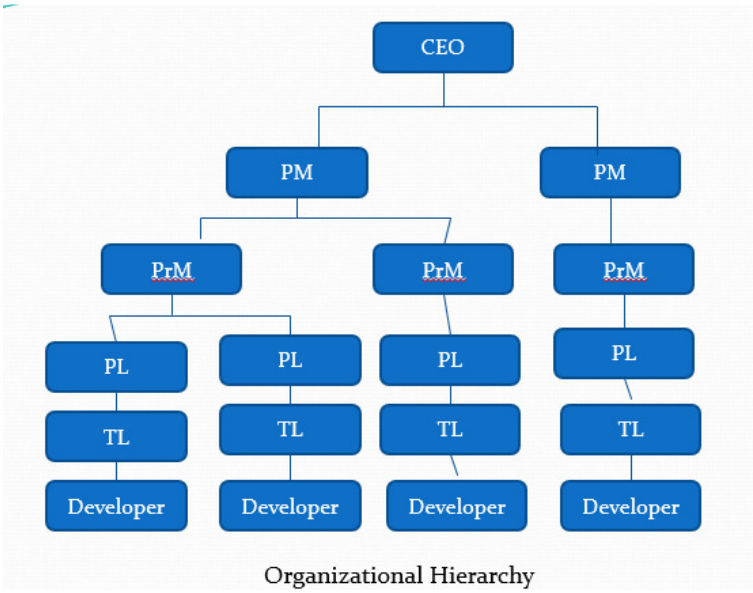
Disadvantages:

Complex implementation:

Only data independence is not enough for designer and programmers to build database system they need to have knowledge of physical data storage which may be complex.

Complex application programming:
Programmers must know how data is stored and path of data storage

Limitations in implementation:
1: N relationship can be implemented but implementing M:N relationship is difficult



Organizational Hierarchy

2.3.2 Network Databases:

- The simple structure of a hierarchical database became a disadvantage when the data had a more complex structure.
- To deal with applications such as order processing, a new network data model was developed. The network data model extended the hierarchical model by allowing a record to participate in multiple parent/child relationships
- Like the hierarchical model, this model also uses pointers toward data but there is no need of parent to child association so it does not necessarily use a downward tree structure. This model uses network databases
- This database model is similar to hierarchical model up to some aspect.
- A relationship between any two record types is called as a set.
- EXAMPLE – IDS (Integrated Data Store) one of the products based on network models developed by IBM and North American Rockwell

Advantages:

Simple design:
The network model is simple and easy to design and understand.

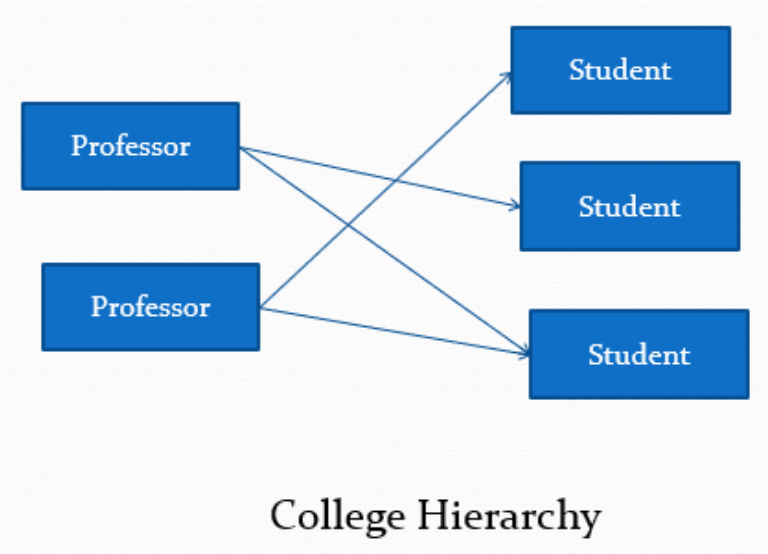
Ability to handle many types of relationship:
The network model can handle the one-t o-many or many-to-many or other relationships.
Hence network model manages multiuser user environment

Ease of data access:
In a network model an application can access a root (parent) record and all the member records within a set(child)

Disadvantages:

System complexity:
Data are accessed one record at a time hence the complexity for the system increases for accessing multiple records at a time.

Lack of structural independence:
Any changes made to the database structure (or data) requires the application programs to be modified before it can access data.



College Hierarchy

2.4 BASIC BUILDING BLOCK

The basic building block for any of model is Entities, Attributes , relationships and constraints.

2.4.1 Entity:

A fundamental component of a model. An entity is having its own independent existence in real world.

E.g.: A Student, Faculty, Subject having independent existence.
An entity may be an object with a physical existence, or it may have logical existence.

E.g.: Entities like Department, Section, adult(age>18) may have physical existence or it may have only logical existence.

2.4.2 Attributes:

Each entity has its own properties which describes that entity, such properties are called as attributes

A particular entity will have some value for each of its attributes

– Employee entity may be described by attributes name, age, phone etc.

2.4.3 Relationships:

It is an association among several entities for e. g. Employee works for Department.

The degree of the relationship is the number of participating entity types in a particular relation

Data model uses three types of relationships

2.4.4 Degree:

The degree of relationship type is number of participating entity types in a particular relation

2.5 TYPES OF RELATIONSHIPS

One is to one:

One entity is associated with at most one other entity.
E.g., One department can have only one manager

One is to Many:

One entity is associated with any number of entities in other entity.
E.g., One teacher may teach to many students

Many is to Many:
One entity is associated with any number of entities in other entity.
E.g., Books in library issued by students

2.6 BUSINESS RULES

- **Definition:** Business rules are statements of a discrete operational business policy or practice within specific organisations that constrains the business. It is intended to control or influence the behaviour of the business.
- Database designer needs to take help from concepts such as entity, attributes and relationships to build a data model, but the above things are not sufficient to describe a system completely.
- Business rules may define actors and prescribe how they should behave by setting constraints and help to manage business change in the system.

2.6.1 Characteristics of Business Rules:

Atomicity:
Rule should define any one aspect of the system environment.
E.g.: - College should have students in it.

Business format:
Rule should be expressed in business terms understandable tobusiness people.

E.g.: ER diagram, object diagram etc

Business ownership:
Each rule is governed by a businessperson who is responsible for verifying it, enforcing it, and monitoring need for change.
E.g.: End user or customer is responsible for requirements submitted by him.

Classification:
Each rule can be classified by its data and constraints.

Business Formalism:
Each rule can be implemented in the related information system. Business rules should be consistent and non-redundant.

EXAMPLES:
A student may take admission to college
One subject is taught by only one professor
A class consists of minimum 60 and maximum 80 students

2.6.2 Types of Business Rules:
DEFINITIONS:
Define some business terms. Definitions are incorporated in systems data dictionary.
E.g., A professor is someone who teaches to students

FACTS:
Connect business terms in ways that make business sense. Facts are implemented as relationships between various data entities
E.g., A professor may have student

CONSTRAINTS:
Shows how business rules and how business terms are connected with each other. Constraints usually state how many of one data entity can be related to another data entity
E.g., Each professor may teach up to four subjects.

DERIVATIONS:
Enable new knowledge or actions. Derivations are often implemented as formulas and triggers.
E.g. A student pending fees is his fees paid minus total fees.

2.7 DEGREES OF 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. hide the complexity from users through

several levels of abstraction, to simplify users interactions with the system:

- **Physical level:** The lowest level of abstraction describes how the data are el data structures in detail.
- **Logical level:** 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 the user of the logical level does not need to be aware of this complexity. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.
- **View level:** 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.

Type customer = record
customer-id: string;
customer-name: string;
customer-street: string;
customer-city: string;
end;

This code defines a new record type called customer with four fields. Each field has a name and a type associated with it.

A banking enterprise may have several such record types, including account, with fields account number and balance employee, with fields employee name and salary

At the physical level, a customer, account, or employee record can be described as a block of consecutive storage locations (for example, words or bytes). The language compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest level storage details from database programmers.

Database administrators, on the other hand, may be aware of certain details of the physical organization of the data.

At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of

these record types is defined as well. Programmers using a programming language work at this level of abstraction.

Similarly, database administrators usually work at this level of abstraction

Finally, at the view level, computer users see a set of application programs that hide details of the data types. Similarly, at the view level, several views of the database are defined, and database users see these views. In addition to hiding details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing certain parts of the database.

2.8 LET US SUM UP

- A database-management system (DBMS) consists of a collection of interrelated data and a collection of programs to access that data. The data describe one particular enterprise.
- The primary goal of a DBMS is to provide an environment that is both convenient and efficient for people to use in retrieving and storing information
- A data-manipulation language (DML) is a language that enables users to access or manipulate data. Nonprocedural DMLs, which require a user to specify only what data are needed, without specifying exactly how to get those data, are widely used today.
- A data-definition language (DDL) is a language for specifying the database schema and as well as other properties of the data
- Transaction management ensures that the database remains in a consistent (correct) state despite system failures. The transaction manager ensures that concurrent transaction executions proceed without conflicting.

2.9 LIST OF REFERENCE

- A Silberschatz, H Korth, S Sudarshan, “Database System and Concepts”, fifth Edition McGraw- Hill
- Introduction to Database System,C.J.Date,Pearson
- Database Systems ,Rob Coronel,Cengage Learning.

2.10 BIBLIOGRAPHY

- Programming with PL/SQL for Beginners, H. Dand, R. Patil and T. Sambare,X-Team.

*****

3

DATABASE DESIGN

Unit Structure

3.0 OBJECTIVES

The objective of the chapter is as follow

- To get familiar with the design of the database schema
- To understand the influence of design choice on database schema
- To understand the concept of ER Model

3.1 INTRODUCTION

- The task of creating a database application is a complex one, involving design of the database schema, design of the programs that access and update the data, and design of a security scheme to control access to data.
- The needs of the users play a central role in the design process
- The design of a complete database application environment that meets the needs of the enterprise being modelled requires attention to a broad set of issues.
- These additional aspects of the expected use of the database influence a variety of design choices at the physical, logical, and view levels.

3.2 ERRELATIONSHIP MODEL

- 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
- The E-R data model
- Employs three basic concepts: entity sets, relationship sets, and attributes
- The E-R model also has an associated diagrammatic representation, the E-R diagram,

3.2.1 Entity:

An entity is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in an enterprise in an entity.

An entity has a set of properties, and the values for some set of properties may uniquely identity an entity. For instance, a person may have a person-id property whose value uniquely identifies that person

An entity may be concrete, such as a person or a book, or it may be abstract, such as a loan, or a holiday, or a concept.

3.2.2 Entity Set:

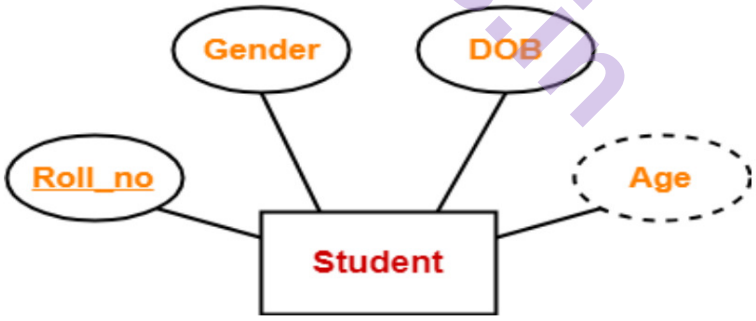
An entity set is a set of entities of the same type that share the same properties, or attributes. The set of all people who are instructors at a given university

3.2.3 Attributes:

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. Each entity has a value for each of its attributes



3.2.4 Relationship:

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.

3.2.5 Simple and Composite attributes:

Simple and composite attributes. In our examples thus far, the attributes have been simple; that is, they have not been divided into subparts. Composite attributes, on the other hand, can be divided into subparts (that is, other attributes). For example, an attribute name could be structured as a composite attribute consisting of first name, middle initial, and last name

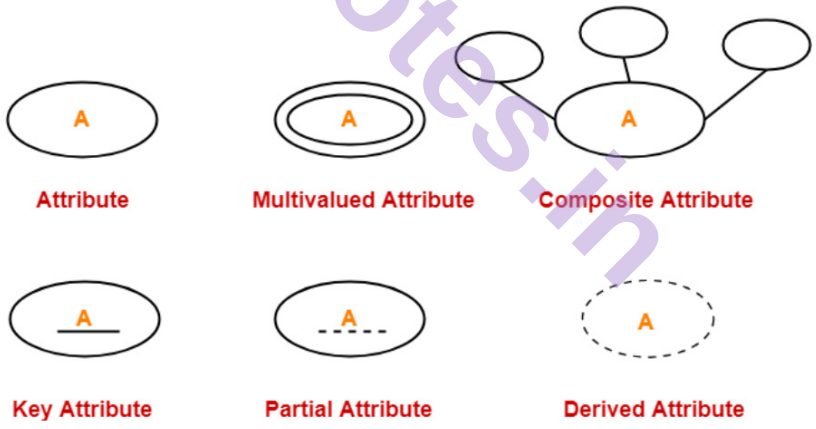
3.2.6 Single-valued and multivalued attributes:

The attributes in our examples all have a single value for a particular entity. For instance, the student ID attribute for a specific student entity refers to only one student ID. Such attributes are said to be single valued.

An instructor may have zero, one, or several phone numbers, and different instructors may have different numbers of phones. This type of attribute is said to be multivalued.

3.2.7 Derived attribute:

The value for this type of attribute can be derived from the values of other related attributes or entities. Eg age can be derived from the birthdate



3.3 CONSTRAINTS

An E-R enterprise schema may define certain constraints to which the contents of a database must conform. In this section, we examine mapping cardinalities and participation constraints, which are two of the most important types of constraints

3.3.1 Mapping Cardinalities:

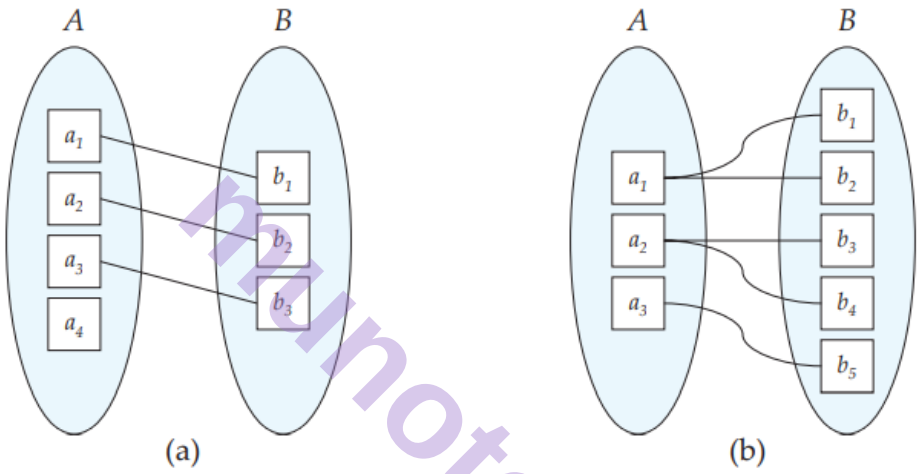
- Mapping cardinalities, express the number of entities to which another entity can be associated via a relationship set.

- Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets.

For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following:

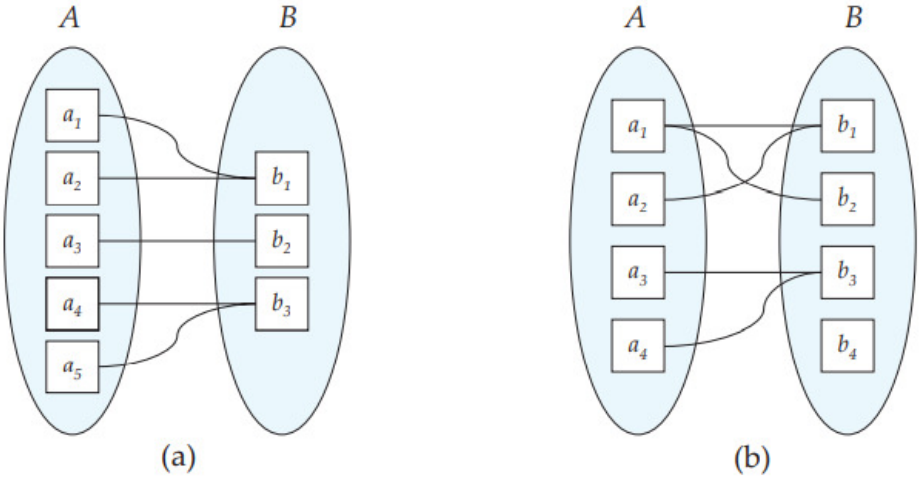
• **One to one:** An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A. (see figure (a).)

• **One to many:** An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, can be associated with at most one entity in A. (see figure (b))



• **Many to one:** An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A. (see figure a).

• **Any to many:** An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A. (see figure b).



3.3.2 Participation Constraints:

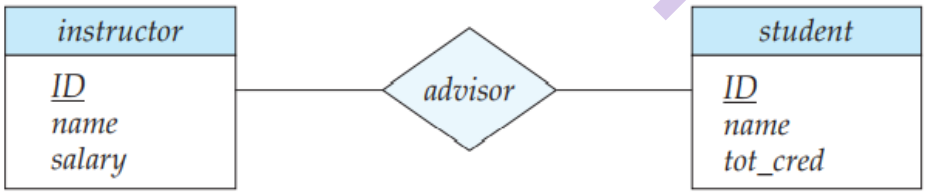
The participation of an entity set E in a relationship set R is said to be total if every entity in E participates in at least one relationship in R. If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be partial.

3.3.3 Keys:

- The column value that uniquely identifies a single record in the table is called as KEY of table.
- An attribute or set of attributes whose values uniquely identify each entity in an entity set is called as key for that entity set
- Any key consisting of single attribute is called a simple key while that consisting of a combination of attributes is called a composite key
- A super key is any combination of fields within a table that uniquely identifies each record within that table.
- A candidate key is a subset of super key. It is a single field or the least combination of fields that uniquely identifies each record in the table
- The least combination of fields distinguishes a candidate key from a super key
- A primary key is a candidate key that is most appropriate to be the main reference key for the table.

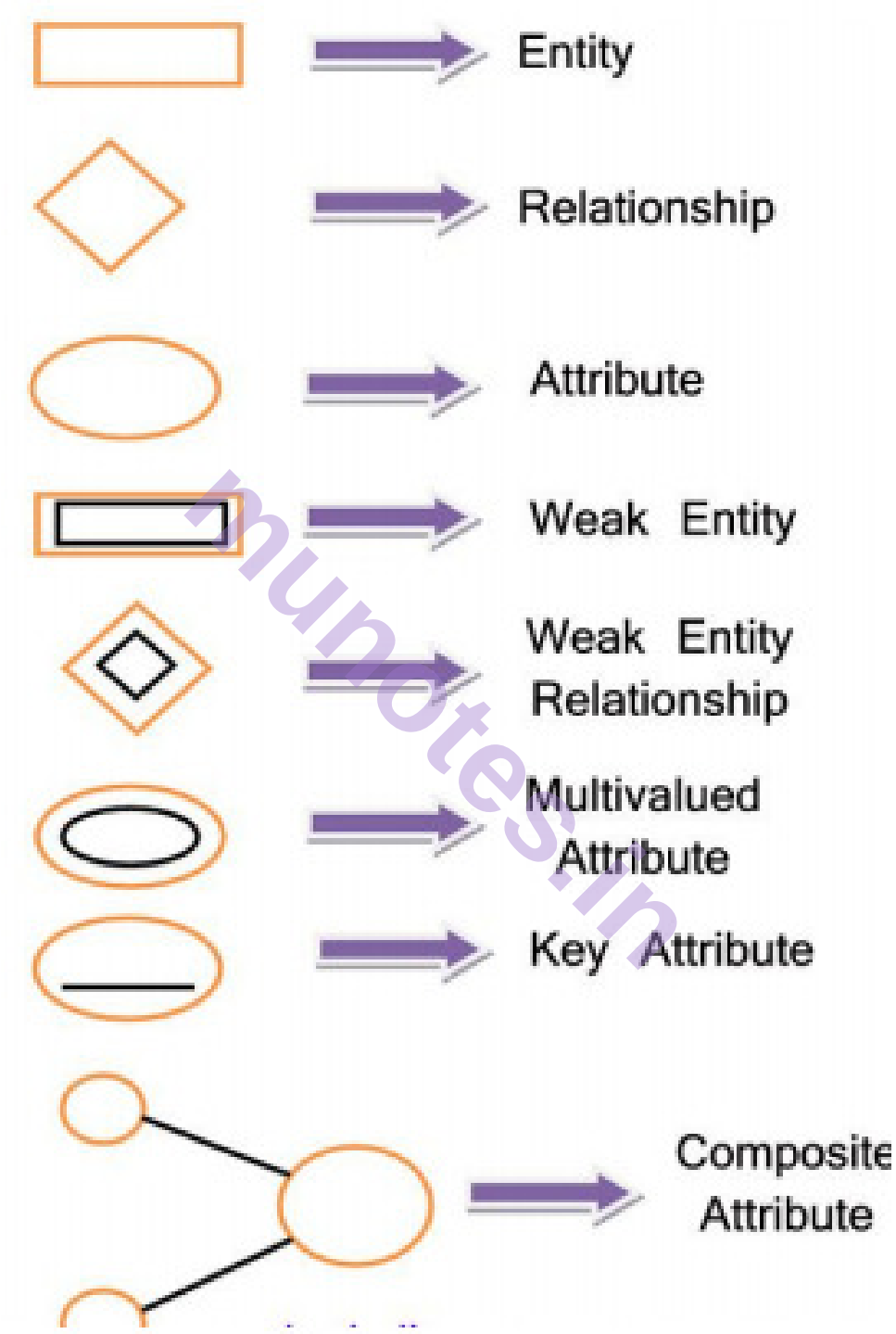
3.4 ER DIAGRAM

E-R diagram can express the overall logical structure of a database graphically. E-R diagrams are simple and clear—qualities that may well account in large part for the widespread use of the E-R model.



- **Rectangles divided into two parts represent entity sets**. The first part, which in this textbook is shaded blue, contains the name of the entity set. The second part contains the names of all the attributes of the entity set.
- **Diamonds** represent relationship sets.
- **Undivided rectangles** represent the attributes of a relationship set. Attributes that are part of the primary key are underlined.
- **Lines** link entity sets to relationship sets.
- **Dashed lines** link attributes of a relationship set to the relationship set.

- **Double lines** indicate total participation of an entity in a relationship set.
- **Double diamonds** represent identifying relationship sets linked to weak entity sets



3.4.1 Mapping Cardinality:

The relationship set advisor, between the instructor and student entity sets may be one-to-one, one-to-many, many-to-one, or many-to-many.

- **One-to-one:** A directed line from the relationship set advisor to both entity sets instructor and student. This indicates that an instructor may advise at most one student, and a student may have at most one advisor(figure a).
- **One-to-many:** A directed line from the relationship set advisor to the entity set instructor and an undirected line to the entity set student. This indicates that an instructor may advise many students, but a student may have at most one advisor(figure b).
- **Many-to-one:** An undirected line from the relationship set advisor to the entity set instructor and a directed line to the entity set student. This indicates that an instructor may advise at most one student, but a student may have many advisors.
- **Many-to-many:** An undirected line from the relationship set advisor an instructor may advise many students, and a student may have many advisors(figure c).

(a)

(b)

(c)

3.4.2 Strong Entity Set:

- A single rectangle is used for the representation of a strong entity set.
- It contains sufficient attributes to form its primary key
- A diamond symbol is used for the representation of the relationship that exists between the two strong entity sets.
- A single line is used for the representation of the connection between the strong entity set and the relationship.
- Total participation may or may not exist in the relationship.

3.4.3 Weak entity set:

- A double rectangle is used for the representation of a weak entity set
- It does not contain sufficient attributes to form its primary key.
- A double diamond symbol is used for the representation of the identifying relationship that exists between the strong and weak entity set
- A double line is used for the representation of the connection between the weak entity set and the relationship set
- Total participation always exists in the identifying relationship.

3.5 ERD ISSUES

- A common mistake is to use the primary key of an entity set as an attribute of another entity set, instead of using a relationship
- A common mistake is to use the primary key of an entity set as an attribute of another entity set, instead of using a relationship
- It is not always clear whether an object is best expressed by an entity set or a relationship
- Relationships in databases are often binary. Some relationships that appear to be non binary could actually be better represented by several binary relationships.

3.6 CODD'S RULES

1. **Information Rule:** All available data in a system should be represented as relations or tables.
2. **Guaranteed Access Rule:** Each data item must be accessible without ambiguity by providing table name and its primary key of the row also include its column name to be accessed
3. **Systematic Treatment of Null Values:** Null values are not equal to blank space or zero they are unknown unassigned values which should be treated properly

4. **Self-Describing Database:** There should be dynamic online catalog based dictionary on relational model which keep information about tables data in database
5. **Comprehensive Data Sublanguage**: The data access language (SQL) must be the only means of accessing data stored in the database and support DML, DDL etc.
6. **View Updating Rule:** All views of data are theoretically updateable can be updated using system also
7. **High Level Insert, Update And Delete:** This rule states that in a relational database, the query language must be capable of performing manipulations on sets of rows in a table
8. **Physical Data Independence:** Any changes made in the way is physically stored must not affect applications that access data
9. **Logical Data Independence:** This rule states that changes to the database to the database design should be done in a way without the users being aware of it
10. **Integrity Independence:** Data integrity constraints which are definable in the language must be stored in the database as data in table is, in the catalog and not in the application program
11. **Distribution Independence:** In a RDBMS data can be stored centrally that is on a system or distributed across multiple systems
12. **Non-Subversion Rule :** This rule states that there should be no bypass of constraints by any other languages

3.7 RELATIONAL SCHEMA

The E-R model and the relational database model are abstract, logical representations of real-world enterprises. Because the two models employ similar design principles, an E-R design can be converted into a relational design.

3.7.1 Representation of Strong Entity Sets with Simple Attributes:

Let E be a strong entity set with only simple descriptive attributes a1, a2,..., an. We represent this entity by a schema called E with n distinct attributes. Each tuple in a relation on this schema corresponds to one entity of the entity set E

An entity set student of the E-R diagram. This entity set has three attributes: ID, name, tot_ cred. We represent this entity set by a schema called student with three attributes

student (ID, name, tot cred)

3.7.2 Representation of Strong Entity Sets with Complex Attributes:

A strong entity set has non simple attributes. A composite attribute is handled by creating a separate attribute for each of the component

attributes; consider the version of the instructor entity set. For the composite attribute name, the schema generated for instructor contains the attributes first name, middle name, and last name is no separate attribute or schema for name

3.7.3 Representation of Weak Entity Sets:

Let A be a weak entity set with attributes a1, a2,..., am. Let B be the strong entity set on which A depends. Let the primary key of B consist of attributes b1, b2,..., bn. We represent the entity set A by a relation schema called A with one attribute for each member of the set {a1, a2,..., am} ∪ {b1, b2,..., bn}

3.8 INTRODUCTION TO UML

- UML is a standard language for specifying visualizing, constructing and documenting the artifacts of software systems.
- UML is specially proposed standard for creating specifications of various components of a complex software system.
- UML stands for Unified Modelling Language but is different from the other common programming languages like C++,Java etc.
- UML is a specification language that is used in the software engineering field.

3.8.1 Class diagram:

Most popular UML diagrams used for construction of software applications.

Like E-R diagram. It is a static diagram. Shown using class as its basic entity and lines between them represents relationship between them. Describes the attributes and operations of a class and the constraints imposed on the system.

3.8.2 Use case diagram:

Shows the interaction between users and the system in particular steps of tasks that users perform. Purpose is to capture the dynamic aspect of a system and to gather requirements of a system. It also identifies external and internal factors influencing the system.

3.8.3 Activity Diagram:

Is basically a flowchart to represent the flow from one activity to another. The activity can be described as an operation of the system. It is a particular operation of the system and not only used for visualizing

dynamic nature of a system but also used to construct the executable system by using forward and reverse engineering techniques.

3.8.4 Implementation diagram:

Implementation diagrams show the system components and their interconnections, both at the software component level and the hardware component level.

3.8.5 Advantages Of UML Diagrams:

- It is the most useful method of visualization and documenting software systems design .
- It is effective for modeling large, complex software systems.
- It is simple to learn, but provides advanced features for expert analysts, engineers, designers and architects.
- It can specify systems in an implementation - independent manner.
- It specifies a skeleton that can be refined and extended with additional features.
- It specifies the functional requirements of system in an object oriented manner

3.8.6 Disadvantages Of UML Diagrams:

- Still no specification for modelling of graphical user interface.
- Not suitable for distributed systems – no way to formally specify serialization and object persistence.

3.9 LET US SUM UP

- Database design mainly involves the design of the database schema. The entity-relationship (E-R) data model is a widely used data model for database design. It provides a convenient graphical representation to view data, relationships, and constraints.
- An entity is an object that exists in the real world and is distinguishable from other objects. We express the distinction by associating with each entity a set of attributes that describes the object.
- A relationship is an association among several entities. A relationship set is a collection of relationships of the same type, and an entity set is a collection of entities of the same type.
- The terms super key, candidate key, and primary key apply to entity and relationship sets as they do for relation schemas. Identifying the primary key of a relationship set requires some care, since it is composed of attributes from one or more of the related entity sets.

- Mapping cardinalities express the number of entities to which another entity can be associated via a relationship set.
- An entity set that does not have sufficient attributes to form a primary key is termed a weak entity set. An entity set that has a primary key is termed a strong entity set.

3.10 LIST OF REFERENCE

- A Silberschatz, H Korth, S Sudarshan, "Database System and Concepts", fifth Edition McGraw- Hill
- Database Systems ,Rob Coronel,Cengage Learning.
- Introduction to Database System,C.J.Date,Pearson

3.11 BIBLIOGRAPHY

Programming with PL/SQL for Beginners, H. Dand, R. Patil and T. Sambare,X-Team.

3.12 UNIT END EXERCISE

1. Explain the concept of weak entity set
2. Write various symbols and their meaning used to draw ER diagram
3. Explain the concept of mapping cardinalities in detail
4. Write a short note on Codd's rules
5. Write short note on UML

*****

Unit II

4

RELATIONAL DATABASE MODEL

Unit Structure

4.0 OBJECTIVES

1. Is to make students aware of different keys available in relational table.
2. Make them aware of different forms of Normalization.

4.1 LOGICAL VIEW OF DATA

- Tables / Relations are logical structure which is a collection of 2-dimensional tables consisting of horizontal rows and vertical columns.
- It is an abstract concept and do not represent how data is stored in physical memory of computer system.
- Each table in database has its own unique table name by which its contents can be referred.

4.1.1 Characteristics of Table/Relation:

- A table is perceived as 2-dimensional structure composed of rows and columns.

- Each table row(tuple) represents a single entity occurrence within the entity set.
- Each table column represents an attribute, and each column has a distinct name.
- Each row/column intersection represents a single data value.
- All values in a column must confirm to the same data format.
- Each column has specific range of values known as attribute domain.
- The order of the rows and columns is immaterial to DBMS.
- Each table must have an attribute or a combination of attributes that uniquely identifies each row.

4.1.2 Attributes:

- Each column in the table represents one data item stored in database for that table.
- Such column in database is called as attribute of a table.
- Tables must have at least one column in it and no two columns can have same name.
- The ANSI/ISO SQL standard does not specify a maximum number of columns in a table.

4.1.3 Tuple/Record

- A single row or tuple contains all the information about a single entity.
- Each horizontal row of the table represents a single entity.
- A table can have any number of rows from zero to thousand.
- If number of rows are zero, then it is called as empty table.

4.2KEY

- The column value that uniquely identifies a single record in the table is called as **KEY** of table.
- An attribute or set of attributes whose values uniquely identify each entity in an entity set is called as key for that entity set.
- Any key consisting of single attribute is called a simple key while that consisting of a combination of attributes is called a composite key.
- Keys are very important part of Relational database. They are used to establish and identify relation between tables.
- They also ensure that each record within a table can be uniquely identified by combination of one or more fields within a table.

4.2.1 Types of keys:

- Super Key
  - Super Key is defined as a set of attributes within a table that uniquely identifies each record within a table. Super Key is a superset of Candidate key.

- Candidate Key
  - Candidate keys are defined as the set of fields from which primary key can be selected.
  - It is an attribute or set of attributes, that can act as a primary key for a table to uniquely identify each record in that table.

- Primary Key
  - Primary key is a candidate key that is most appropriate to become main key of the table.
  - It is a key that uniquely identify each record in a table.
- **For example:**
  - Customer table

Cust_id	Order_id	Sales_details

    - So here, cust_id&order_id forms a composite key.

4.3 INTEGRITY RULES

- Integrity rules may sound very technical, but they are simple and straightforward rules that each table must follow.
- These are very important in database design, when tables break any of the integrity rules our database will contain errors when retrieving information.
- Hence the name "integrity" which describes reliability and consistency of values.
- There are two types of integrity rules that we will look at:
  - Entity Integrity Rule
  - Referential Integrity Rule
- Entity Integrity Rule:
  - The entity integrity rule refers to rules the primary key must follow.
  - The primary key value cannot be null.
  - The primary key value must be unique.

- If a table does not meet these two requirements, we say the table is violating the entity integrity rule.
- For example, does this table violate entity integrity rule? Where?

Student

Roll_no	Name
1	Ram
2	Shyam
	Pooja
3	Neha
5	Rani
5	Pankaj

- The team *Student* violates the entity integrity rules at two places.
  - Student Pooja - missing primary key
  - Student Rani & Pankaj have the same primary key
- Referential Integrity Rule:
  - The referential integrity rule refers to the foreign key.
  - The foreign key may be null and may have the same value but:
    - The foreign key value must match a record in the table it is referring to.
    - Tables that do not follow this are violating the referential integrity rule.

4.4 RELATIONAL DATABASE DESIGN:

- The relational database model was conceived by E.F. Codd in 1969, then a researcher at IBM.
- The basic idea behind the relational model is that a database consists of a sequence of relations (or tables) that can be manipulated using non-procedural operations that return tables.
- A relational DBMS must use its relational facilities exclusively to manage and interact with the database.
- When designing a database, we need to decide which tables to create, what columns they will contain, as well as the relationships between the tables.

4.4.1 Goals:

- Design should ensure that all database operations will be efficiently performed, and DBMS should not perform expensive consistency checks.
- No data redundancy should be there.

4.5 FEATURES OF GOOD RELATIONAL DATABASE DESIGN

- The primary feature of a relational database is its primary key, which is a unique identifier assigned to every record in a table.
- An example of a good primary key is a registration number. It makes every record unique, facilitating the storage of data in multiple tables, and every table in a relational database must have a primary key field.
- Another key feature of relational databases is their ability to hold data over multiple tables.
- This feature overcomes the limitations of simple flat file databases that can only have one table.
- The database records stored in a table are linked to records in other tables by theprimary key.
- The primary key can join the table in a one-to-one relationship, one-to-many relationship or many-to-many relationship.
- Relational databases enable users to delete, update, read and create data entries in the database tables.
- This is accomplished though structured query language, or SQL, which is based on relational algebraic principles.
- SQL also enable users to manipulate and query datain a relational database.
- Relational tables follow various integrity rules that ensure the data stored in them is always accessible and accurate.
- The rules coupled with SQL enable users to easily enforce transaction and concurrency controls, thus guaranteeing data integrity.
- The relational database concept was established by **Edgar F. Codd** in 1970.

4.6 NORMALIZATION (1NF, 2NF, 3NF, BCNF)

4.6.1 What is Normalization?:

- Database Normalisation is a technique of organizing the data in the database.

- Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.
- It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables.
- Normalization is used for mainly two purpose,
  - Eliminating redundant (useless) data.
  - Ensuring data dependencies make sense i.e data is logically stored.
- Problem without Normalization:
  - Without Normalization, it becomes difficult to handle and update the database, without facing data loss.
  - Insertion, Updation and Deletion Anomalies are very frequent if Database is not Normalized.

4.6.2 Normalization Rule:

- Normalization rule are divided into following normal form.

1. First Normal Form (1NF):

- As per First Normal Form, no two Rows of data must contain repeating group of information i.e. each set of columns must have a unique value, such that multiple columns cannot be used to fetch the same row.
- Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.
- The Primary key is usually a single column, but sometimes more than one column can be combined to create a single primary key.
- For example, consider a table which is not in First normal form

Student

Roll_no	Name	Subject
1	Ram	Biology, Maths
2	Shyam	English, Science
3	Pooja	Maths
4	Neha	Physics

  - In First Normal Form, any row must not have a column in which more than one value is saved, like separated with commas.
  - Rather than that, we must separate such data into multiple rows.

- *Student* Table following 1NF will be:

Name	Age	Subject
Ram	15	Biology
Ram	15	Maths
Shyam	14	English
Shyam	14	Science
Pooja	16	Maths
Neha	17	Physics

- Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows, but each row will be unique.

2. Second Normal Form (2NF):

- As per the Second Normal Form there must not be any partial dependency of any column on primary key.
- It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence.
- If any column depends only on one part of the concatenated key, then the table fails Second normal form.
- In example of First Normal Form there are two rows for Ram, to include multiple subjects that he has opted for. While this is searchable, and follows First normal form, it is an inefficient use of space.
- Also, in the above Table in First Normal Form, while the candidate key is {Student, Subject}, Age of Student only depends on Student column, which is incorrect as per Second Normal Form.
- To achieve second normal form, it would be helpful to split out the subjects into an independent table and match them up using the student names as foreign keys.

New *Student* table following 2NF will be:

Name	Age
Ram	15
Shyam	14
Pooja	16
Neha	17

- In *Student* Table the candidate key will be Student column because all other column i.e Age is dependent on it.
- *New Subject* Table introduced for 2NF will be:

Name	Subject
Ram	Biology

Ram	Maths
Shyam	English
Shyam	Science
Pooja	Maths
Neha	Physics

- In Subject Table the candidate key will be {Student, Subject} column. Now, both the above tables qualifies for Second Normal Form and will never suffer from Update Anomalies.
- Although there area few complex cases in which table in Second Normal Form suffers Update Anomalies, and to handle those scenarios Third Normal Form is there.

3. Third Normal Form:

- Third Normal form applies that every non-prime attribute of table must be dependent on primary key, or we can say that there should not be the case that a non-prime attribute is determined by another non-prime attribute.
- This transitive functional dependency should be removed from the table and the table must be in Second Normal form. For example, consider a table with following fields.

*Student_detail*Table

Stud_id	*Stud_name*	*DOB*	*Street*	*City*	*State*	*Zip*

- In this table Stud_id is Primary key, but street, city and state depend upon Zip.
- The dependency between zip and other fields is called transitive dependency. Hence to apply 3NF, we need to move the street, city and state to new table, with Zip as primary key.

New *Student_detail* table:

Stud_id	Stud_name	DOB	Zip

Address table:

Zip	Street	city	State

The advantage of removing transitive dependency is,

- Amount of data duplication is reduced.
- Data integrity achieved.

4. Boyce and Codd Normal Form (BCNF):

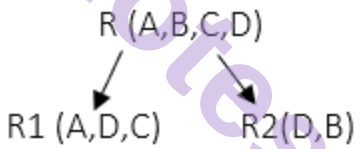
- Boyce and Codd Normal Form is a higher version of the Third Normal form.

- This form deals with certain type of anomoly that is not handled by 3NF.
- A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF.
- For a table to be in BCNF, following conditions must be satisfied:
  - R must be in 3rd Normal Form
  - **and** for each functional dependency (X -> Y),X should be a super Key.
- Consider the following relationship: **R (A,B,C,D)**
- And following dependencies:

  A -> BCD

  BC -> AD

  D -> B
- Above stated relationship is already in 3NF. Keys are A & BC.
- Hence, the functional dependency, A -> BCD, A is the Super key.
- In second relation, BC -> AD, BC is also a key but in D -> B, D is not a key.
- Hence, we can break our relationship R into two relationships **R1**&**R2**.



4.7 LET US SUM IT UP

1. Types of keys available - Primary, foreign, composite, candidate.
2. Characteristics of Good relational table.
3. Normalization with its types – 1NF, 2NF, 3Nf & BCNF

4.8 LIST OF REFERENCES

1. A Silberschatz, H Korth, S Sudarshan, “Database System and Concepts”, fifth EditionMcGraw- Hill

4.9 BIBLIOGRAPHY

1. Database Systems,Rob Coronel,Cengage Learning.
2. Programming with PL/SQL for Beginners,H.Dand, R. Patil and T. Sambare,XTeam.

*****

5

RELATIONAL ALGEBRA

Unit Structure

5.0 OBJECTIVES

1. Is to make students aware of relational algebra.
2. Develop the foundation for modelling data using algebraic structures.

5.1 RELATIONAL ALGEBRA

- The relational algebra defines a set of operations on relations, paralleling the usual algebraic operations such as addition, subtraction, or multiplication, which operate on numbers.
- Just as algebraic operations on numbers take one or more numbers as input and return a number as output, the relational algebra operations typically take one or two relations as input and return a relation as output.

5.1.1 Introduction:

- The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result.
- The fundamental operations in the relational algebra are select, project, union, set difference, Cartesian product, and rename.

- In addition to the fundamental operations, there are several other operations namely, set intersection, natural join, division, and assignment.
- The Tuple Relational Calculus:
  - When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query.
  - The tuple relational calculus, by contrast, is a non-procedural query language.
  - It describes the desired information without giving a specific procedure for obtaining that information.
  - A query in the tuple relational calculus is expressed as**{ t | P(t) }.**
  - That is, it is the set of all tuples t such that predicate P is true or t. following our earlier notation, we use t[A] to denote the value of tuple **t**on attribute A, and we use **t € r** to denote that tuple ***t*** *is inrelation* ***r***.

5.1.2 The Selection Operation:

- The select operation selects tuples that satisfy a given predicate. We use the lowercase Greek lettersigma (**σ**) to denote selection.
- The predicate appears as a subscript to **σ**.
- The argument relation is inparentheses after the σ.
- Thus, to select those tuples of the loan relation where the branch is "Perryridge", we write

$$\sigma_{branch\_name="Perryridge"}(loan)$$

- The relation that results from the preceding query is as shown in figure 1.
- We can find all tuples in which the amount lent is more than ₹1200 by writing,

$$\sigma_{amount>1200}(loan)$$

- In general, it allows comparisons using **=, ≠, ≤, <, ≥, >**in the selection predicate.
- Furthermore, we can combine several predicates into a larger predicate by using the connectives and (**∧**), or (**∨**), and not (**−**).
- Thus, to find those tuples pertaining to loans of more than $1200 made by the Perry ridge branch, we write

$$\sigma_{\text{branch-name="Perryridge"} \wedge \text{amount} > 1200}(\text{loan})$$

Loan-number	Branch-name	Amount
L-15	Perryridge	1500
L-16	Perryridge	1300

- Output of above query.
- The selection predicate may include comparisons between two attributes.
- To illustrate, consider the relation loan-officer that consists of three attributes: customer-name, banker-name, and loan-number, which specifies that a particular banker is the loan officer for a loan that belongs to some customer.
- To find all customers who have the same name as their loan officer, we can write

$$\sigma_{\text{customer-bane=banker-name}}(\text{loan-officer})$$

5.1.3 The Projection Operation:

- Suppose we want to list all loan numbers and the amount of the loans, but do not care about the branch name.
- The project operation allows us to produce this relation. The project operation is aunary operation that returns its argument relation, with certain attributes left out.
- Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter Pi (Π).
- We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses.
- Thus, we write the query to list all loan numbers and the amount of the loan as

$$\Pi_{\text{loan-number, amount}}(\text{loan})$$

- Following table shows the relation that results from this query.

Loan-number	Amount
L-11	900
L-14	1500

L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

5.2 SET OPERATIONS

- The SQL operations union, intersect, and except operate on relations and correspond to the relational algebra operations ∪, ∩ and –.
- Like union, intersection, and set difference in relational algebra, the relations participating in the operations must be compatible; that is, they must have the same set of attributes.
- Let us demonstrate how several of the example queries that we considered in Chapter 3 can be written in SQL.
- We shall now construct queries involving the union, intersect, and except operations of two sets: the set of all customers who have an account at the bank, which can be derived by,

```
Select customer-name
From depositor
```

and the set of customers who have a loan at the bank, which can be derived by,

```
Select customer-name
From browser
```

1) The Union Operation:

- To find all customers having a loan, an account, or both at the bank, we write,

```
(select customer-name
    from depositor)
    UNION
(select customer-name
    from borrower)
```

- The union operation automatically eliminates duplicates, unlike the select clause.
- Thus, in the preceding query, if a customer - say, Joneshas several accounts or loans (or both) at the bank, then Jones will appear only once in the result.

- If we want to retain all duplicates, we must write union all in place of union:

```
(select customer-name
   from depositor)
   UNION ALL
(select customer-name
   from borrower)
```

- The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both.
- Thus, if Jones has three accounts and two loans at the bank, then there will be five tuples with the name Jones in the result.

2) The intersect Operation:

- To find all customers who have both a loan and an account at the bank, we write,

```
(select distinct customer-name
 from depositor)
intersect
(select distinct customer-name
 from borrower)
```

- The intersect operation automatically eliminates duplicates. Thus, in the preceding query, if a customer-say, Jones-has several accounts and loans at the bank, then Jones will appear only once in the result.
- If we want to retain all duplicates, we must write intersect all in place of intersect:

```
(select customer-name
 from depositor)
intersect all
(select customer-name
 from borrower)
```

- The number of duplicate tuples that appear in the result is equal to the minimum number of duplicates in both d and b. Thus, if Jones has three accounts and two loans at the bank, then there will be two tuples with the name Jones in the result.

3) The Except Operation:

- To find all customers who have an account but no loan at the bank, we write,

```
(select distinct customer-name
 from depositor)
except
```

(select customer-name
from borrower)

- The except operation automatically eliminates duplicates. Thus, in the preceding query, a tuple with customer name Jones will appear (exactly once) in the result only if Jones has an account at the bank but has no loan at the bank.
- If we want to retain all duplicates, we must write except all in place of except:

(select customer-name
from depositor)
except all
(select customer-name
from borrower)

- The number of duplicate copies of a tuple in the result is equal to the number of duplicate copies of the tuple in d minus the number of duplicate copies of the tuple in b, provided that the difference is positive.
- Thus, if Jones has three accounts and one loan at the bank, then there will be two tuples with the name Jones in the result. If, instead, this customer has two accounts and three loans at the bank, there will be no tuple with the name Jones in the result.

5.3 RENAMING & JOINS

- Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them.
- It is useful to be able to give them names; the rename operator, denoted by the lowercase Greek letter rho (p), lets us do this, Given a relational algebra expression E the expression,

$$\rho_x(E)$$

returns the result of expression E under the name x.

- A relation r by itself is considered a (trivial) relational algebra expression. Thus, we can also apply the rename operation to a relation r to get the same relation under a new name.
- A second form of the rename operation is as follows. Assume that a relational-algebra expression Ehas arity n.
- Then, the expression, $\rho_x(A_1, A_2, \ldots, A_n)$ (E) returns the result of expression E under the namex' and with the attributes renamed to A_1, A_2,…A_n.

- To illustrate renaming a relation, we consider the query "Find the largest account balance in the bank."
- Strategy is to,
  (1) compute first a temporary relation consisting of those balances that are not the largest and
  (2) take the set difference between the relation $\Pi_{balance}$ **(account)** and the temporary relation just computed, to obtain the result.

Step 1: To compute the temporary relation, we need to compare the values of all account balances. We do this comparison by computing the Cartesian product account ×account and forming a selection to compare the value of any two balances appearing in one tuple.

- First, we need to devise a mechanism to distinguish between the two balance attributes.
- We shall use the rename operation to rename one reference to the account relation; thus we can reference the relation twice without ambiguity.

Balance
500
400
700
750
350

- Result of the sub-expression would be,

$$\Pi_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho d\ (account)))$$

Balance
900

- Largest account balance in the bank.
- We can now write the temporary relation that consists of the balances that are not the largest:

$$\Pi_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho d(account)))$$

- This expression gives those balances in the account relation for which a larger balance appears some where in the account relation (renamed as d).
- The result contains all balances except the larges one.

- Step 2: The query to find the largest account balance in the bank can be written as:

$$\Pi_{balance}(account) - \Pi_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho d(account)))$$

- As one more example of the rename operation, consider the query "Find the names of all customers who live on the same street and in the same city as Smith."
- We can obtain Smith's street and city by writing,

$$\Pi_{customer-street,\ customer-city}(\sigma_{customer-name = \text{"Smith"}}(customer))$$

- However, to find other customers with this street and city, we must reference the customer relation a second time.
- In the following query, we use the rename operation on the preceding expression to give its result the name smith-addr, and to rename its attributes to street and city, instead of customer, street and customer-city:

$$\Pi_{customer.customer-name}(\sigma_{customer.customer-street\ =\ Smith-addr.street\ \wedge\ customer.\ customer-city\ =\ smith-addr.city}(customer \times \rho_{smith-addr\,(street,city)}(\Pi_{customer-street,\ customer-city}(\sigma_{customer-name = \text{"Smith"}}(customer)))))$$

- The rename operation is not strictly required, since it is possible to use a positional notation for attributes.
- We can name attributes of a relation implicitly by using a positional notation, where \$1,\$2, … refer to the first attribute, the second attribute, and so on.
- The positional notation also applies to results of relational algebra operations.
- The result of above query looks like this,

Customer-name
Curry
Smith

5.4 DIVISION SYNTAX & SEMANTICS

- The division operation, denoted by ÷, is suited to queries that include the phrase "for all".
- Suppose that we wish to find all customers who have an account at all the branches located in Brooklyn. We can obtain all branches in Brooklyn by the expression,

$$r1 = \prod_{\text{branch-name}} (\sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch}))$$

- The result relation for this expression is shown below,

branch–name
Brighton
Downtown

- We can find all (customer-name, branch-name) pairs for which the customer has an account at abranch by writing,

$$r2 = \prod_{\text{customer-name, branch-name}} (\text{depositor} \bowtie \text{account})$$

- Now, we need to find customers who appear in r2 with every branch name in r1. The operation that
- provides exactly those customers is the divide operation. We formulate the query by writing,

$$\prod_{\text{customer-name, branch-name}} (\text{depositor} \bowtie \text{account}) \div \prod_{\text{branch-name}} (\sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch}))$$

- The result of this expression is a relation that has the schema (customer-name) and that contains the tuple (Johnson).
- Formally, let r(R) and s(S) be relations, and let S ⊆ R; that is, every attribute of schema S is also in schema R.
- The relation r÷s is a relation on schema R –S (that is, on the schema containing all attributes of schema R that are not is schema S). A tuple t is in r ÷ s if and only if both of two conditions hold:

  1. t is in ΠR–S(r)
  2. For every tuple ts in s, there is a tuple tr in r satisfying both of the following:

  (a) ts [S] = ts[S]

  (b) tr [R–S] = t

- It may surprise you to discover that, given a division operation and the schemas of the relations, we can, in fact, define the division operation in terms of the fundamental operations. Let r(R) and s(S)be given, with $S \subseteq R$:

$$r \div s = \prod R-S(r) - \prod R-S ((\prod R-S (r) \times s) - \prod R-S, S(r))$$

customer-name	branch–name
Heyes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

Result of Πcustomer-name, branch-name (depositor ∞account).

- To see that this expression is true, we observe that ΠR–S (r) gives us all tuples t that satisfy the first condition of the definition of the definition of division.
- The expression on the right side of the set difference operator,

$$\prod R-S ((\prod R-S (r) \times s) - \prod R-S, S(r))$$

- serves to eliminate those tuples that fail to satisfy the second condition of the definition of division.
- Let us see how it does so. Consider ΠR–S (r) × s. This relation is on schema R, and pairs every tuple in ΠR–S, (r) with every tuple in s.
- The expression ΠR–S, S(r) merely reorders the attributes of r.
- Thus, (ΠR–S (r) × s) –ΠR–S, S(r) gives us those pairs of tuples from ΠR–S (r) and s that do not appear in r.
- If tuple tj is in,

$$\prod R-S ((\prod R-S (r) \times s) - \prod R-S, S(r))$$

- Then there is some tuple ts in s that does not combine with tuple tj to form a tuple in r.
- Thus, tj holds a value for attributes R–S that does not appear in r ÷ s.
- It is these values that we eliminate fromΠR–S (r).

5.5 OPERATORS

Symbol (Name)	Example of Use
σ	$\sigma_{salary>=85000}(instructor)$
(Selection)	Return rows of the input relation that satisfy the predicate.
Π	$\Pi_{ID,salary}(instructor)$
(Projection)	Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
⋈	$instructor \bowtie department$
(Natural join)	Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.
×	$instructor \times department$
(Cartesian product)	Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes)
∪	$\Pi_{name}(instructor) \cup \Pi_{name}(student)$
(Union)	Output the union of tuples from the two input relations.

Ref. Database System Concepts by Korth

5.6 LET US SUM IT UP

1. There are majorly used 2 operations – selection & projection.
2. Using relational algebra we can perform basic set operations such as union, intersection & set difference.
3. Whereas we can also perform Division operation, rename etc.

5.7 LIST OF REFERENCES

1. A Silberschatz, H Korth, S Sudarshan, "Database System and Concepts", fifth EditionMcGraw- Hill

5.8 BIBLIOGRAPHY

1. Database Systems,Rob Coronel,Cengage Learning.
2. Programming with PL/SQL for Beginners,H.Dand, R. Patil and T. Sambare,XTeam.
3. Introduction to Database System,C.J.Date,Pearson.

5.9 UNIT END EXERCISE

1. Explain the concept of Relational Algebra.
2. Define Joins and its types.
3. Explain set operators with example.
4. Explain Selection and projection operation in relational algebra.

*****

6

CALCULUS

Unit Structure

6.0 OBJECTIVES

1. Is to make students aware of the difference between relational algebra & Calculus.
2. Develop the foundation for modelling data using algebraic structures.

6.1 TUPLE RELATIONAL CALCULUS

- When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query.
- The tuple relational calculus, by contrast, is a non-procedural query language.
- It describes the desired information without giving a specific procedure for obtaining that information.
- A query in the tuple relational calculus is expressed as,

  $\{ t \mid P(t) \}$
- That is, it is the set of all tuples t such that predicate P is true or t. following our earlier notation, we use t[A] to denote the value of tuple **t**on attribute A, and we use $t \in r$ to denote that tuple t is in relation r.
- Before we give a formal definition of the tuple relational calculus, we return to some of the queries for which we wrote relational algebra.
- Example Queries
  - Say that we want to find the branch-name, loan-number, and amount for loans of over ₹1200:

    $\{t \mid t \in loan \wedge t[amount] > 1200\}$

- Suppose that we want only the loan-number attribute, rather than all attributes of the loon relation.
- To write this query in the tuple relational calculus, we need to write an expression for a relation on the schema (loan-number).
- We need those tuples on (loan-number) such that there is a tuple in loan with the amount attribute > 1200. To express this request, we need the construct “there exists” from mathematical logic.
- The notation,

$$\exists\, t \in r\ (Q(t))$$

- means "there exists a tuple t in relation r such that predicate Q{t) is true."
- Using this notation, we can write the query "Find the loan number for each loan of an amount greater than ₹1200" as

$$\{t \mid \exists\, s \in \text{loan}\ (t[\text{loan–number}] = s[\text{loan–number}] \wedge s[\text{amount}] > 1200)\}$$

- In English, we read the preceding expression as "The set of all tuples t such that there exists a tuple sin relation loan for which the values of t and s for the loan-number attribute are equal, and the value of s for the amount attribute is greater than ₹1200."
- Tuple variable t is defined on only the loan-number attribute, since that is the only attribute having a condition specified for t. Thus, the result is a relation on (loan–number).
- Consider the query “Find the names of all customers who have a loan from the Perryridge branch”.
- This query is slightly more complex than the previous queries, since it involves two relations:borrower and loan.
- As we shall see, however, all it requires is that we have two "there exists" clausesin our tuple-relational-calculus expression, connected by and (∧).
- We write the query as follows:

$$\{t \mid \exists\, s \in \text{borrower}\ (t[\text{customer-name}] = s[\text{customer–name}]$$
$$\wedge \exists u \in \text{loan}\ (u[\text{loan–number}] = s[\text{loan–number}]$$
$$\wedge \exists\, u \in \text{loan}\ (u[\text{branch–name}] = \text{“Perryridge”}))\}$$

- In English, this expression is “The set of all (customer-name) tuples for which the customer has aloan that is at the Perryridge branch”.
- Tuple variable if ensures that the customer is a borrower at the Perryridge branch. Tuple variable s is restricted to pertain to the same loan number as s.
- To find all customers who have a loan, an account, or both at the bank, we used the union operation in the relational algebra. In the tuple

relational calculus, we shall need two "there exists" clauses, connected by or (V):

{t | ∃ s ∈ borrower (t[customer-name] = s[customer–name])

∧∃u ∈ depositor (t [customer–name] = u[customer–name])}

- This expression gives us the set of all customer-name tuples for which at least one of the following holds:
- The customer name appears in some tuple of the borrower relation as a borrower from the bank.
- The customer-name appears in some tuple of the depositor relation as a depositor of the bank.
- If some customer has both a loan and an account at the bank, that customer appears only once in the result, because the mathematical definition of a set does not allow duplicate members.
- If we now want only those customers who have both an account and a loan at the bank, all we need to do is to change the or (v) to and (∧) in the preceding expression.

  {t | ∃ s ∈ borrower (t[customer-name] = s[customer–name])

  ∧∃u ∈ depositor (t [customer–name] = u[customer–name])}

- Now consider the query "Find all customers who have an account at the bank but do not have a loanfrom the bank." The tuple-relational–calculus expression for this query is like the expressionsthat we have just seen, except for the use of the not (¬) symbol:

  {t | ∃ u ∈ depositor (t [customer-name] = u[customer–name])

  ∧¬∃ s ∈ borrower (t [customer–name] = u[customer–name])}

Customer-name
Adams
Hayes

Fig.: Names of all customers who have a loan at the Perryridge branch

- This tuple-relational-calculus expression uses the ∃ u ∈ depositor (...) clause to require that the customer have an account at the bank, and it uses the ¬∃ s ∈ borrow r (...) clause to eliminate those customers who appear in some tuple of the borrower relation as having a loan from the bank.
- The query that we shall consider next uses implication, denoted, by ⇒.
- The formula P ⇒ Q means "P implies Q"; that is, "if P is true, then must be true."
- Note that P ⇒ Q is logically equivalent to ¬P v Q. The use of implication rather than not and or often suggests, a more intuitive interpretation of a query in English.

- "Find all customers who have an account at all branches located in Brooklyn." To write this query in the tuple relational calculus, we introduce the "for all" construct, denoted by ∀.
- The notation,

  ∀ t ∈ r (Q(t))

  means "Q is true for all tuples t in relation r."
- We write the expression for our query as follows:

  {t | ∃ r ∈ customer (r [customer–name] = t [customer–name])∧

  (∀ u ∈ branch (u [branch–city] = "Brooklyn" ⇒

  ∃ s ∈ depositor (t[customer–name] = s[customer–name]

  ∧∃ w ∈ account (w[account–number] = s[account–number]

  ∧ w [branch–name] = u[branch–name])))}
- In English, we interpret this expression as "The set of all customers (that is, (customer-name) tuples t) such that, for all tuples u in the branch relation, if the value of u on attribute branch-City is Brooklyn, then the customer has an account at the branch whose name appears in the branch–name attribute of u."
- Note that there is a subtlety in the above query: If there is no branch in Brooklyn, all customer names satisfy the condition.
- The first line of the query expression is critical in this case–without the condition.

  ∃ r ∈ customer (r[customer–name] = t [customer–name])
- If there is no branch in Brooklyn, any value of t (including values that are not customer names in the depositor relation) would qualify.
- Formal Definition:
  - We are now ready for a formal definition. A tuple–relational–calculus expression is of the form **{t | P (t)}** Where P is a formula.
- Several tuple variables may appear in a formula. A tuple variable is said to be a free variable unless it is quantified by a ∃ or ∀. thus, in

  t ∈ loan ∧∃ s∈ customer (t[branch–name]

  = s [branch–name])
- T is a free variable. tuple variable s is said to be a bound variable.
- A tuple-relational-calculus formula is built up out of atoms. An atom has one of the following forms:
  - s ∈ r, where s is a tuple variable and r is a relation (we do not allow use of the ∉ operator)
  - s[x] Θ u[y], where s and u are tuple variables, x is an attribute on which s is defined, y is an attribute on which u is defined, and Θ is

a comparison operator (<, ≤, >, ≥, =, ≠); we require thatattributes x and y have domains whose members can be compared by Θ.

- s[x] Θ c, where s is a tuple variable, x is an attribute on which s is defined, Θ is a comparison operator, and c is a constant in the domain of attribute x We build up formulae from atoms by using the following rules:
  - An atom is a formula.
  - If P1 is a formula, then so are ¬P1 and (P1).
  - If P1 and P2 are formulae, then so are P1 ∨ P2, P1 ∧ P2, and P1 ⇒ P2.
  - If P1(s) is a formula containing a free tuple variable s, and r is a relation, then ∃ s ∈ r (P1 (s)) and ∀ s ∈ r (P1 (s)) are also formulae.

- As we could for the relational algebra, we can write equivalent expressions that are not identical in appearance. In the tuple relational calculus, these equivalences include the following three rules:

  i) P1 ∧ P2 is equivalent to ¬(¬(P1)∨¬(P2)).

  ii) ∀t∈r(P1(t)) is equivalent to ¬∃∈ r (¬P1(t)).

  iii) P1 ⇒ P2 is equivalent to ¬ (P1) ∨ P2.

6.2 DOMAIN RELATIONAL CALCULUS

- A second form of relational calculus, called domain relational calculus, uses domain variables that take on values from an attribute domain, rather than values for an entire tuple.
- The domain relational calculus, however, is closely related to the tuple relational calculus.
- Domain relational calculus serves as the theoretical basis of the widely used QBE language, just as relational algebra serves as the basis for the SQL language.
- Formal Definition An expression in the domain relational calculus is of the form { | P (x1, x2, ….., xn)} where x1, x2, …., xn represent domain variables.
- P represents a formula composed of atoms, as was the case in the tuple relational calculus.
- An atom in the domain relational calculus has one of the following forms:
  - < x1 x2,……., xn>∈ r, where r is a relation on n attributes and x1 x2,…, xn are domain variables or domain constants.
  - x Θ y, where x and y are domain variables and Θ is a comparison operator (<, ≤, =, ≠, >, ≥).

- We require that attributes x and y have domains that can be compared by Θ.
- x Θ c, where x is a domain variable, Θ is a comparison operator, and c is a constant in the domain of the attribute for which x is a domain variable.
- We build up formulae from atoms by using the following rules:
  - An atom is a formula.
  - If P1 is a formulae, then so are ¬P1 and (P1).
  - If P1 and P2 are formulae, then so are P1 ∨ P2, P1 ∧ P2, and P1 ⇒P2.
  - If P1 (x) is a formula in x, where x is a domain variable, then ∃x (P1 (x)) and ∀ x (P1(x)) are also formulae.
  - As a notational shorthand, we write ∃ a, b, c (P(a, b, c)) for ∃ a (∃b (∃c (P (a, b, c))))
- Example Queries
  - We now give domain-relational-calculus queries for the examples that we considered earlier.
  - Notethe similarity of these expressions and the corresponding tuple-relational-calculus expressions.
- Find the loan number, branch name, and amount for loans of over ₹1200:

  {<l, b, a> | <l, b, a>∈ loan ∧ a > 1200}
- Find all loan numbers for loans with an amount greater than ₹1200:

  {< l > | ∃ b, a (< l, b, a >∈ loan ∧ a > 1200)}
- Although the second query appears similar to the one that we wrote for the tuple relational calculus, there is an important difference.
- In the tuple calculus, when we write ∃ s for some tuple variables, we bind it immediately to a relation by writing ∃ s ∈ r.
- However, when we write ∃ b in the domain calculus, b refers not to a tuple, but rather to a domain value.
- Thus, the domain of variable b is unconstrained until the sub-formula <l, b, a>∈ loan constrains b to branch names that appear in the loan relation.
- For example,
  - Find the names of all customers who have a loan from the Perryridge branch and find the loan amount

    {<c, a> | ∃ l (< c, l >∈ borrower

    ∧∃ b (<l, b, a>∈ loan ∧ b = "Perryridge"))}
  - Find the names of all customers who have a loan, an account, or both at the Perryridge branch:

{<c> | ∃ l (<c, l>∈ borrower

∧∃ b, a (<l, b, a>∈ loan ∧ b = "Perryridge"))

∨∃ a (<c, a>∈ depositor

∧∃ b, n (<a, b, n>∈ account ∧ b = "Perryridge"))}

- Find the names of all customers who have an account at all the branches located in Brooklyn:

{<c> | ∃ n (<c, n>∈ customer) ∧

∀ x, y, z (<x, y, z>∈ branch ∧ y = "Brooklyn" ⇒

∃ a, b (< a, x, b>∈ account ∧<c, a>∈ depositor))}

- In English, we interpret this expression as "The set of all {customer-name) tuples c such that, for all(branch-name, branch-city, assets) tuples, x, y, z, if the branch city is Brooklyn, then the following is true":
  - There exists a tuple in the relation account with account number **a** and branch name x.
  - There exists a tuple in the relation depositor with customer c and account numbers. a"

6.3 CALCULUS VS ALGEBRA

Calculus	**Relational Algebra**
While Relational Calculus is Declarative language.	It is a Procedural language.
While Relational Calculus means what result we must obtain.	Relational Algebra means how to obtain the result.
While in Relational Calculus, the order is not specified.	In Relational Algebra, the order is specified in which the operations have to be performed.
While Relation Calculus can be a domain dependent.	Relational Algebra is independent on domain
While Relational Calculus is not nearer to programming language.	Relational Algebra is nearer to a programming language.
It is denoted as below, {t | P(t)} where, t: the set of tuples p: is the condition which is true for the given set of tuples.	Basic operations used are, 1. Select (σ) 2. Project (Π) 3. Union (U) 4. Set Difference (-) 5. Cartesian product (X) 6. Rename (ρ)

Ref: Geeksforgeeks

6.4 LET US SUM IT UP

1. There are majorly used 2 operations – selection & projection.

2. Using relational algebra, we can perform basic set operations such as union, intersection & set difference.
3. Whereas we can also perform Division operation, rename etc.
4. Comes in two flavours: Tuple relational calculus (TRC) and
5. Domain relational calculus (DRC)
    a. TRC: Variables range over (i.e., get bound to) tuples.
    b. DRC: Variables range over domain elements (= attributevalues)

6. Both TRC and DRC are subsets of first-order logic

6.5 LIST OF REFERENCES

1. A Silberschatz, H Korth, S Sudarshan, "Database System and Concepts", fifth EditionMcGraw- Hill
2. https://www.ccs.neu.edu/home/kathleen/classes/cs3200/4-RAAndRC.pdf

6.6 BIBLIOGRAPHY

1. Database Systems,Rob Coronel,Cengage Learning.
2. Programming with PL/SQL for Beginners,H.Dand, R. Patil and T. Sambare,XTeam.
3. Introduction to Database System,C.J.Date,Pearson.

6.7 UNIT END EXERCISE

1) Explain the concept of Relational Calculus.
2) Define Domain Relational Calculus with example.
3) Define Tuple Relational Calculus with example.
4) Differentiate between Calculus & relational algebra.

*****

UNIT III

7

CONSTRAINT

Unit Structure

7.0 OBJECTIVES

- To understand Integrity, Domain constraint and Referential Integrity.
- To understand and apply primary key, unique key, Default, Foreign, Null and Check Integrity constraint.
- To understand and apply Assertions Triggers in SQL.
- To Understand and apply Security and Authorization in SQL.

7.1 INTRODUCTION

Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.

Example of integrity constrains are:

- An account balance cannot be null.
- No two students can have same roll no.

In general, an integrity constraint can be any arbitrary predicate pertaining to the database. However, arbitrary predicate can be costly to test. Hence, the most database systems allow one to specify integrity constraint that can be tested with minimal overhead.

7.2 DOMAIN CONSTRAINTS

- Domain constraints are the most elementary form of integrity constraint.
- They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domain can be created from existing data types.

Example:

Create domain Dollars **numeric (12, 2)**

Create domain pounds **numeric (12, 2)**

- We cannot assign or compare a value of type Dollars to a value of type Pounds.

  However, we can convert type as below:

  (Cast r. A **as** Pounds)
- (Should also multiply by the dollar-to-pound conversion-rate)

Different types of constraints are:

7.3 REFERENTIAL INTEGRITY

A value that appears in one relation for given set of attributes also appears for a certain set of attributes in another relation. This called referential integrity.

Referential integrity in the E-R model:

Referential integrity constraints arise frequently. If we derive our relational database scheme by constructing tables from E-R diagrams then every relation arising from a relationship set has referential integrity constraints.

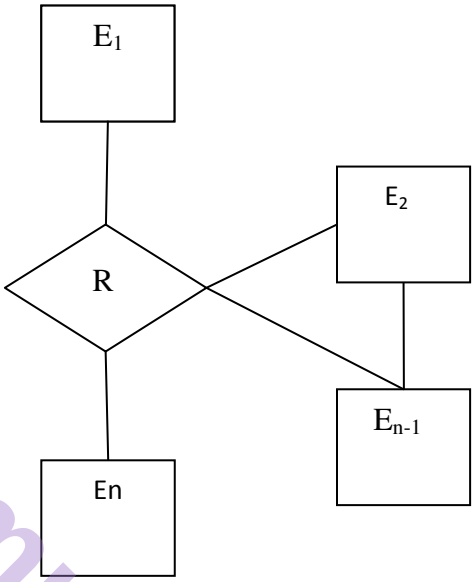


Figure : 7.3.1 An n-ary relationship set

As shown in Fig. 7.3.1, an n-ary relationship set R, relating entity sets E_1, E_2, ……E_n.

Let K_i denote the primary key of E_i. The attributes of the relation scheme for relationship set R include K_1 ∪K_2 ∪ …..∪ K_n. Each K_i in the scheme for R is a foreign key that leads to a referential integrity constraint.

Another source of referential integrity constraints are weak entity sets. The relation scheme for a weak entity set entity set must include the primary key of the entity set on which it depends. Thus, the relation scheme for each weak entity set includes a foreign key that leads to a referential integrity constraint.

- **Referential integrity in SQL :**

Using SQL, primary key, candidate key, and foreign key are defined as part of the create table statement as given below:

Example:

```
Create table Deposit
(Branch_name char (15),
Acc_no char(10),
Cust_name char (20) not null,
Balance integer,
Primary key (Acc_no, Cust_name),
foreign key (Branch_name) references Branch,
foreign key (Cust_name) references Customer) ;
```

- **Cascading Actions in SQL**

```
create table Account
foreign key (Branch-name) references Branch
on delete cascade
                on update cascade
. . .)
```

- Due to the **on delete cascade** clause, if a delete of a tuple in Branch results in referential-integrity constraint violation, the delete_casades_ to the Account relation, deleting the tuple that refers to the branch that was deleted.
- Cascading updates are similar.
- If there is a chain of foreign-key dependencies across multiple relations, with **on delete cascade** specified for each dependency, a deletion or update at one end of the chain can propagate across the entire chain.
- If a cascading update to delete causes a constraint violation that caanot be handled by a further cascading operation, the system aborts the transaction. As a result, all the changes caused by the transaction and its cascading actions are undone.
- Referential integrity is only checked at the end of a transaction. Intermediate steps are allowed to violate referential integrity provided later steps remove the violation. Otherwise it would be impossible to create some database states.

Example:

Insert two tuples whose foreign keys point to each other:

```
# Example: spouse attribute of relation
Marriedperson (Name, Address, Spouse)
```

```
Alternative to cascading:
" on delete set null
                " on delete set default
```

- Null values in foreign key attributes complicate SQL referential integrity semantics, and are best prevented using **not null.** If any attribute of a foreign key is null, the constraint.

7.4 DATA CONSTRAINTS

Besides the column name, datatype and length, there are other parameters that can be passed to the DBA at cell creation time.

These data constraints are checked by DBA when data is assigned to columns. If the data being loaded fails any of the data constraint checks fired by the DBA, the DBA will not load the data into the column, reject the entered record and will display error message to the user.

These constraints are given a constraint name and the DBA stores the constraints with it's name and instructions internally along with the column itself.

The constraints can either be placed at the column level or at the table level.

1) Column level constraints:

If the constraints are defined along with the column definition, it is called a column level constraint. Column level constraint can be applied to any column. If the constraint spans across multiple columns, the user has to use table level constraint.

2) Table level constraint:

If the data constraints attached to a specific column in a table references the contents of another column in the table then the user will have to use table level constraints.

Examples of different constraints that can be applied on the table are as follows:

- Null Value Concept
- Primary Key Concept
- Unique Key Concept
- Default Value Concepts
- Foreign Key Concepts
- Check Integrity Constraints

7.4.1 NULL Value Concept:

While creating tables, if a row lacks a data value for a particular column, that value is said to be null. Columns of any data types may

contain null values unless the column was defined as not null when the table was created.

Principles of NULL Values:

- Setting a null value is appropriate when the actual value is unknown, or when a value would not be meaningful.
- A null value is not equivalent to a value of zero.
- A null value will evaluate to null in any expression. Example : null multiplied by 10 is null.
- When a column name is defined as not null, then that column becomes a mandatory column. It implies that that the user is forced to enter data into that column.

7.4.2 Primary Key Concept:

A primary key is one more column in a table used to uniquely identify each row in the table. primary key values must not be null and must not unique across the column.
A multicolumn primary key is called a composite primary key.

Examples:

1) Column level primary key constraint

```
SQL > CREATE TABLE student
          (roll_no numbers (5) PRIMARY KEY,
           name varchar(25) NOT NULL,
           address varchar(25) NOT NULL,
           ph_no varchar (15);
```

Table created.

2) Table level primary key constraint.

```
SQL >     CREATE TABLE student1
                    (roll_no number(5),
                    name varchar(25) NOT NULL,
                    address varchar(25) NOT NULL,
                   ph_no varchar(15),
                   PRIMARY KEY(roll_no));
```

Table created.

7.4.3 Unique Key Concept:

Unique key is used to ensure that the information in the column for each record is unique, as with license number. A table may have many unique keys.

Example:

```
CREATE TABLE special_customer
(customer_code number(5) PRIMARY KEY,
customer_name varchar(25) NOT NULL,
customer_address varcher (30) NOT NULL,
license_no varchar (15) constraint uk_license_no_UNIQUE);
```

7.4.4 Default Value Concept:

At the time of column creation a 'default value' can be assigned to it. When the user is loading a record with values and leaves this column empty, the DBA will automatically load this column with the default value specified. The data type of the default value should match the data type of the column. You can use the default clause to specify any default value you want.

Example

```
SQL > CREATE TABLE employee
      (emp_code number(5),
       emp_name varchar(25) NOT NULL,
       ph_no varchar(15),
       married char(1) DEFAULT 'M' ,
      PRIMARY KEY (emp_coder));
```

Table created.

7.4.5 Foreign Key Concept:

Foreign key represents relationship between tables. The existence of a foreign key implies that the table with the foreign key is related to the primary key table table from which the foreign key is derived.

The foreign key/ references constraint

- Rejects an INSERT or UPDATE of a value, if a corresponding value does not currently exist in the primary key table.
- Rejects a DELETE, if it would invalidate a REFERENCES constraint.
- Must refer a PRIMARY KEY or UNIQUE column in primary key table.
- Will refer the PRIMARY KEY of primary key table if no column or group of columns is specified in the constraint.
- Must refer a table, not a view or cluster.
- Requires that the FOREIGN KEY column(s) and the CONSTRAINT column (s) have matching data types.

Example

```
SQL > CREATE TABLE book
(ISBN varchar(25) PRIMARY KEY,
 title varchar(25) , pub_year varchar(4),
 unit_price number(4),
 author_name varchar(25) references author,
 publisher_name varchar(25) references publisher);
```

Table created.

7.4.6 Check Integrity Constraint:

The CHECK constraint defines a condition that every row must satisfy. There can be more than one CHECK constraint on a column and the CHECK constraint can be defined at column as well as table level.

At the column level , the constraint is defined by,

```
DeptID Number (2) CONSTRAINT ck-deptID
CHECK (( DeptID >= 10) and (DeptID <= 99));
```

And at the table level by

```
CONSTINT ck-deptId
CHECK (DeptID > = 10) and (DeptID < = 99));
```


Following are few examples of appropriate CHECK constraints.

- Add CHECK constraint on the Emp_Code column of the Employee so that every Emp_Code should start with 'E'.
- Add CHECK constraint on the City column of the Employee so that only the cities 'Mumbai', 'Pune', 'Nashik', 'Solapur' are allowed.

Example

```
SQL > CREATE TABLE employee
  (emp_code number(5) CONSTRAINT ck_epcode CHECK
(emp_code like 'E%),
          emp_name varchar(25) NOT NULL,
          city varchar (30) CONSTRAINT ck_city
          CHECK (city IN ('Mumbai', 'Pune', 'Nashik', 'Solapur' )),
           salary numbers(5),
PRIMARY KEY (emp_code));
```

Table created.

Restrictions on CHECK constraints:

- The condition must be a Boolean expression that can evaluated using the values in the row being inserted or updated.
- The condition cannot contain subqueries or sequences.
- The condition cannot include the SYS DATE, UID, USER OR USER ENV SQL functions.

Defining integrity constraints in the ALTER TABLE command:
You can also define integrity constraints using constraint clause in the ALTER TABLE command.

Consider following existing tables:

1) Student with definition:

```
SQL > CREATE TABLE student
        (roll_no number(3),
         name varchar(25);
```

2) Test with definition:

```
SQL > CREATE TABLE test
        (roll_no number(3),
         Subject_ID number(2),
         Marks number(2));
```

3) Subject_info with definition:

```
SQL > CREATE TABLE subject_info
(subject_ID number(2) PRIMARY KEY,
        subject_name varchar(15));
```

The following examples show the definitions of several integrity constraints.

1) ADD PRIMARY KEY constraints on column roll_no in student table.

```
SQL > ALTER TABLE student
        ADD PRIMARY KEY(roll_no)
```

Table altered.

2) Modify column marks to include NOT NULL constraint.

```
SQL > ALTER TABLE test
        MODIFY (marks number(3) NOT NULL);
```

Dropping integrity constraints in the ALTER TABLE command:

You can drop an integrity constraint if the rule that it enforces is no longer true or if the constraint is no longer needed. Drop the constraint using the ALTER TABLE command with the DROP clause.

The following examples illustrate the dropping of the integrity constraints.
1) Drop following primary key of student table.

```
SQL > ALTER TABLE student
      DROP PRIMARY KEY;
```

Table altered.

2) Drop unique key constraint on column license-no in table customer.

```
SQL > ALTER TABLE student
      DROP CONSTRAINT license_ukey;
```

7.5 ASSERTIONS

An assertion is a predicate expressing a condition that we wish the database always to satisfy.

- An assertion in SQL takes the form:

```
Create assertion < assertion-name >check <predicate>
```

- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion. This testing may introduce a significant amount of overhead; hence assertions be used with great care.

Assertion Examples:

1) The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
Create assertion sum-constraint check
(not exists (select *  from branch
where (select sum (amount) from loan
where loan. Branch-name =
        branch.branch-name)
> = (select sum (amount) from account
where loan.branch-name =
branch.branch-name)))
```

2) Every loan has atleast one borrower who maintains an account with a minimum balance or $1000.00

```
        create assertion balance-constraint check
    (not exists (
select *  from loan
where not exists(
select*
from borrower, depositor, account
where loan.loan-number = borrower.loan-number
and borrower.customer-name = depositor. customer-name
and depositor.account-number = account.account-number
and account.balance > = 1000)))
```

7.6 TRIGGER

A trigger is a statement that the system executes automatically as a side effect of a modification to the database. To design a trigger mechanism, we must meet two requirements:

1. Specify when a trigger is to be executed. This is broken up into event that causes the trigger to be checked and as condition that must be satisfied for trigger execution to proceed.
2. Specify the actions to be taken when the trigger executes.

The above model of triggers is referred to as the **event-condition-action** model for trigger.

The database stores triggers just as if they were regular data, so that they are persistent and are accessible to all database operations. Once we enter a trigger into the database, the database system takes on the responsibility of executing it whenever the specified event occurs and the corresponding condition is satisfied.

7.6.1 Need for Triggers:

Triggers are useful mechanism for alerting humans for starting certain tasks automatically when certain conditions are met. For example, suppose that, instead of allowing negative account balances, the bank deals with overdrafts by setting the account balance to zero and creating a loan in the amount of the overdraft. The bank gives this loan a loan number identical to the account number of the overdraft. The bank gives this loan a loan number identical to the account number of the overdrawn account. For this example, the condition for executing the trigger is an update to the account relation that results in a negative balance value. Suppose that Jones withdrawal of some money from an account made the account balance negative. Let t denote the account tuple with a negative balance value. The actions to be taken are:

- Insert a new tuple s in the loan relation with

```
s[loan_no] = t [account_not]
```

```
s[branch_name] = t [branch_name],
 s[amount] = - t [balance]
```

- Insert a new tuple u in the borrower relation with

```
u [customer_name] = "Jones"
 u [loan_no] = t [account_no]
```

- Set t [balance] to 0.

As another example of the use of triggers, suppose a warehouse wishes to maintain a minimum inventory of each item; when the inventory level of an item falls below the minimum level, an order should be placed automatically. This is how the business rule can be implemented by triggers: on an update of the inventory level of an item, the trigger should compare the level with the minimum inventory level for the item, and if the level is at or below the minimum, a new order is added is added to an orders relation.

7.6.2 Triggers in SQL:

SQL based database systems use triggers widely. Example of SQL trigger is given below:

```
Create trigger overdraft_trigger after update on account
new row as nrow
for each row
when nrow.balance < 0
begin atomic
insert into borrower
(select customer_name, account_no
from depositor
where nrow. account_no = depositor. account_no);
insert into loan values
(nrow.account_no, nrow.branch_name, -nrow.balance) ;
update account set balance = 0
where account. account_no = nrow.account_no;
end
```

This trigger definition specifies that the trigger is initiated after any update of the relation account is executed. The referencing new as clause creates a variable nrow, which stores the value of an update row after the update. Then, for each row, when statement checks the value of balance whether it is less than zero or not. If it is, then customer_name and account_no of depositor for given account_no inserted into borrower relation. A new tuple with values nrow.account_no, nrow.branch_name and _nrow.balance is inserted into loan relation, and finally balance of that account_no is set to zero in account relation.

7.6.3 When not to use Triggers:

Triggers should be written with great care, since a trigger error detected at run time causes the failure of the insert/delete /update statement that set off the trigger. The action of one trigger can set off another trigger. In worst case, this could lead to an infinite chain of triggering. For example, suppose an insert trigger on a relation has an action that causes another (new) insert on the same relation.

The insert action then triggers yet another insert action, and so on. Database system typically limits the length of such chains of triggers, and considers longer chains of triggering an error.

7.7 SECURITY AND AUTHORIZATION IN SQL

7.7.1 Security:

Security is protection from malicious attempts mechanisms to steal or modify the data. The security should be provided at following levels:

1. Database system level:

- Use Authentication and authorization mechanisms to allow specific users access only to required data.

2. Operating system level:

- Operating system super-users can do anything they want to the database. Good operating system level security is required.

3. Network level:

- Use encryption to prevent
- Eavesdropping (unauthorized reading of messages).
- Masquerading (pretending to be an authorized user or sending messages supposedly from authorized users).

4. Physical level:

- Here, physical access to computers allows destruction of data by intruders; traditional lock-and-key security is needed.
- Computers must also be protected from floods, fire, etc.

5. Human level:

- Users must be screened to ensure that authorized users do not give access to intruders.
- Users should trained on password selection and secrecy.

7.7.2 Authorization:

We should assign a user several forms of authorizations on parts of the database.

Different forms of authorization on parts of the database are :

- **Read authorization** – allows reading, but not modification of data.
- **Insert authorization-** allows insertion of new data, but not modification of existing data.
- **Update authorization-** allows modification, but not deletion of data.
- **Delete authorization-** allows deletion of data.

Different forms of authorization to modify the database schema are :

- **Index authorization –**allows creation and deletion of indices.
- **Resource authorization-** allows creation of new relations.
- **Alteration authorization –**allows addition or deletion of attributed in a relation.
- **Drop authorization-** allows deletion of relations.

Each of these types of authorizations is called a privilege. We may authorize the use all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

Privileges in SQL:

The SQL includes the privileges select, insert, update and delete. The select privilege authorizes a user to read data. In addition, to these forms of privileges, SQL supports several other privileges, such as the privilege to create, delete or modify relations, and privilege to execute the procedures.

7.7.3 Authorization in SQL:

The SQL DDL includes commands to grant and revoke privileges.

1. Grant command:

The grant statement is used to confer authorization. The form of grant is :

Grant < privilege list > on < relation name or view name > to < user / role list>

The privilege list allows the granting of several privileges in one command.

Example:

The following grant statement grants database users Shyam and Om select authorization on the Student relation:

```
Grant select on Student to Shyam, Om
```

2. Revoke command:

The revoke command is used to cancel the authorization. The form of revoke statement is :

```
Revoke < privilege list > on <relation name or view name >
from < user / role list >
```

To revoke the above privileges write following statement,

```
Revoke select on Student from Shyam , Om
```

Limitation of SQL Authorization:

- SQL does not support authorization at a tuple level.
  - Example: We cannot restrict students to see only (the tuples storing) their own grades.
- With the growth in Web access to databases, database accesses come primarily from application servers.
  - End users don't have database user ids, they are all mapped to the same database user id.
- All end-users of an application (such as a web application) may be mapped to a single database user.
- The task of authorization in above cases falls on the application program, with no support from SQL :
  - Benefit is: Fine-granted authorizations, such as to individual tuples, can be implemented by the application.
  - Drawback is: Authorization is required to be done in application code. Checking for absence of authorization loopholes becomes very difficult since it requires reading large amounts of application code.

7.8 IMPORTANT QUESTIONS AND ANSWERS

Q1) Explain the integrity constraints: Not Null, Unique, Primary Key with an example each. Is the combination 'Not Null, primary Key' a valid combination. Justify.

Answer:

- **Not Null:** Should contain valid values and cannot ne NULL.

- **Unique:** An attribute or a combination of two or more attributes must have a unique value in each row. The unique key can have NULL values.
- **Primary Key:** It is same as unique key but cannot have NULL values. A table can have at most one primary key in it.

For Example:
Consider following '**Student'** table:

Roll_No	Name	City	Mobile No
01	Om Patil	Pune	9822398776
02	Shanti Desai	Mumbai	9019198234

In 'Student' table:

- Roll_No is a primary key.
- Name is defined with NOT NULL, means each student must have a name.
- Mobile_No is unique.

'Not Null' constraint in it but we can also add 'Not Null' constraint with it. The use of

'Not Null' with 'Primary Key' will not have any effect. It is same as if we are using just

'Primary Key'.

Q.2 What do you mean by integrity constrains? Explain the two constraints, check and foreign key in SQL with an example for each. Give the syntax.

Answer :

- **Integrity Constraints:** An integrity constraint is a condition specified on a database schema and restricts and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constraints specified on the database schema, it is a legal instance. A DBMS enforces integrity constraints, in that it permits only legal instances to be stored in the database.

- **CHECK Constraint:** CHECK constraint specifies an expression that must always be true for every row in the table. it the table. it can't refer to values in other rows.

Syntax:

```
ALTER TABLE < table_name >
ADD CONSTRAINT < constraint_name > CHECK (< expression>);
```

- **FOREIGN KEY constraint:** A foreign key is combination of columns with values based on the primary key values from another table. A foreign key constraint, also known as referential integrity constraint, specifies that the values of the foreign key correspond to actual values of the primary or unique key in other table. One can refer to a primary or unique key in the same table also.

Syntax :

```
ALTER TABLE < table_name >
ADD CONSTRAINT < constraint_name > FOREIGN KEY ( < column_name(s) > )
REFRENCES < base_table > (column_name > ) ON { DELETE | UPDATE}
CASCADE;
```

Q3) Discuss the types of integrity constraints that must be checked for the update operations- Insert and Delete.

Answer: Insert operation can violet any of the following four constraints:

1. Domain constraints can be violated if given attribute value does not appear in corresponding domain.
2. Key constraint can be violated if given attribute value does not appear in corresponding domain.
3. Entity integrity can be violated if the primary key of the new tuple t is NULL.
4. Referential integrity can be violated if value of any foreign key in t refers to a tuple that does not exist in referenced relation.

Delete operation can violate only referential integrity constraints , if the tuple being deleted is referenced by the foreign keys from other tuples in the database.

7.9 SUMMARY

A value that appears in one relation for given set of attributes also appears for a certain set of attributes in another relation. The relation scheme for each weak entity set includes a foreign key that leads to a referential integrity constraint. The relation scheme for each weak entity set includes a foreign key that leads to a referential integrity constraint.

Data constraints are checked by DBA when data is assigned to columns. If the data being loaded fails any of the data constraint checks fired by the DBA, the DBA will not load the data into the column, reject the entered record and will display error message to the user.

If the constraints are defined along with the column definition, it is called a column level constraint. If the data constraints attached to a specific column in a table references the contents of another column in the table then the user will have to use table level constraints.

If a row lacks a data value for a particular column, that value is said to be null. Columns of any data types may contain null values unless the column was defined as not null when the table was created.

A primary key is one more column in a table used to uniquely identify each row in the table. primary key values must not be null and must not unique across the column.

Unique key is used to ensure that the information in the column for each record is unique.

At the time of column creation a 'default value' can be assigned to it. When the user is loading a record with values and leaves this column empty, the DBA will automatically load this column with the default value specified. The data type of the default value should match the data type of the column.

The existence of a foreign key implies that the table with the foreign key is related to the primary key table table from which the foreign key is derived.

The CHECK constraint defines a condition that every row must satisfy. There can be more than one CHECK constraint on a column and the CHECK constraint can be defined at column as well as table level.

An assertion is a predicate expressing a condition that we wish the database always to satisfy.

A trigger is a statement that the system executes automatically as a side effect of a modification to the database.

Security is protection from malicious attempts mechanisms to steal or modify the data.

Use Authentication and authorization mechanisms to allow specific users access only to required data.

Read authorization – allows reading, but not modification of data,Insert authorization- allows insertion of new data , but not modification of existing data, Update authorization- allows modification , but not deletion of data, Delete authorization- allows deletion of data.

QUESTIONS

Q1) What is integrity? Explain constraints with example.

Q2) Explain referential integrity constraints with example.

Q3) What is need of trigger ? Explain when not to use trigger.

Q4) What is assertion ? Explain with example.

Q5) Write short note on :

a. Assertion b. Trigger

Q6) Write short note on security mechanism in database.

Q7) What do you mean by authorization and authentication in DBMS? Explain how it is implemented in SQL with suitable example.

Q8) Discuss different security and authorization mechanism in database.

Q9) What is meant by authorization in DBMS? What is granting of privileges? Explain security mechanisms in database.

REFERENCES

1. Peter Rob and Carlos Coronel, ― Database Systems Design, Implementation and

   Management‖, Thomson Learning, 9thEdition.

2. G. K. Gupta : "Database Management Systems", McGraw – Hill.

Text Books:

1. Korth, Slberchatz, Sudarshan, Database System Concepts, 6th Edition, McGraw Hill
2. Elmasri and Navathe, Fundamentals of Database Systems, 6thEdition, Pearson education
3. Raghu Ramkrishnan and Johannes Gehrke, Database Management Systems, TMH

*****

8

STRUCTURED QUERY LANGUAGE PART-I

Unit Structure

8.0 OBJECTIVE

- To understand characteristics of SQL, Literals, Operators.
- To understand and apply different types of SQL commands to real world problems.
- To Understand Data Definition Commands and solve real world problems.

8.1 INTRODUCTION

Structured Query Language (SQL) is the standard command set used to communicate with the relational database management systems. All tasks related to relational data management creating tables, querying the database for information, modifying the data in the database, deleting them, granting access to users and so on can be done using SQL.

8.1.1 Characteristics of SQL:

SQL usage by its very nature is extremely flexible. It uses a free from syntax that gives the user the ability to structure SQL statements in a way best suited to him. Each SQL request is parsed by the RDMS before execution, to check for proper syntax and to optimize the request. Unlike certain programming languages, there is no need to start SQL statements in a particular column or be finished in a single line. The same SQL request can be written in a variety of ways.

8.1.2 Advantages of SQL:

The various advantages of SQL are:

- SQL is a high level language that provides a greater degree of abstraction than procedural languages.
- SQL enables the end-users and systems personnel to deal with a number of database management systems where it is available. Increased acceptance and availability of SQL are also in its favor.
- Applications written in SQL can be easily ported across systems. Such porting could be required when the underlying DBMS needs to be upgraded or changed.
- SQL specifies what is required and not how it should be done.
- The language while being simple and easy to learn can handle complex situations.
- All SQL operations are performed at a set level. One select statement can retrieve multiple rows, one modify statement can modify multiple rows. This set at a time feature of the SQL makes it increasingly powerful than the record at a time processing techniques employed in language like COBOL.

8.2 SQL LITERALS

There are four kinds of literal values supported in SQL. They are :

- Character string
- Bit string
- Exact numeric
- Approximate numeric

1. Character string:

Character strings are written as a sequence of characters enclosed in single quotes. The single quote character is represented within a character string by two single quotes. Some example of character strings are:

- Computer Engg'

- Structured Query Language'

2. Bit string:

A bit string is written either as a sequence of 0 s and 1s enclosed in single quotes and preceded by the letter 'B' or as a sequence of hexadecimal digits enclosed in single quotes and preceded by the letter 'X' some examples are given below:

- B'1011011'
- B'1'
- B'0'
- X'A 5'
- X'1'

3. Exact numeric:

These literals are written as a signed or unsigned decimal number possibly with a decimal point. Examples of exact numeric literals are given below:

- 9
- 90
- 90.00
- 0.9
- + 99.99
- -99.99

4. Approximate numeric:

Approximate numeric literals are written as written as exact numeric literals followed by the letter R 'E', followed by a signed or unsigned integer. Some example are:

- 5E5
- 55.5E5
- +55E-5
- 0.55E
- -5.55E-9

8.3 TYPES OF SQL COMMANDS

SQL provides set of commands for a variety of tasks including the following:

- Querying data
- Updating, inserting and deleting data
- Creating, modifying and deleting database objects

- Controlling access to the database
- Providing for data integrity and consistency.

For example, using SQL statements you can create tables, modify them, delete the tables, query the data in the tables, insert data into the tables, modify and delete the data, decide who can see data and so on.

The SQL statement is a set of instructions to the RDMS to perform an action. SQL statements are divided into the following categories:

- Data definition language (DDL)
- Data Manipulation Language (DML)
- Data Query Language(DQL)
- Data Control Language (DCL)

1. Data Definition Language (DDL):

- Data definition language is used to create, alter and delete database objects.
- The commands used are create, alter and drop.
- The principal data definition statements are :
  - Create table, create view, create index
  - Alter table
  - Drop table, drop view, drop index

2) Data Manipulation Language (DML):

Data Manipulation Language commands let users insert, modify and delete the data in the database. SQL provides three data manipulation statements insert, update and delete.

3) Data Query Language(DQL):

This is one of the most commonly used SQL statements. This SQL statement enables the users to query one or more tables to get the information they want. SQL has only one data query statement 'select'.

4) Data Control Language (DCL):

The data control language consists of commands that control the user access to the database objects. Various DCL commands are: Commit, Rollback, Save point, Grant, Revoke.

8.4 SQL OPERATORS

Operators and conditions are used to perform operations such as addition, subtraction or comparison on the data items in an SQL statement.

Different types of SQL operators are:

Arithmetic operators:

Arithmetic operators are used in SQL expressions to add, subtract, multiply, divide and negate data values. The result of this expression is a number value.

Unary operators (B)	
+, -	Denotes a positive or negative expression
Binary operator (B)	
*	Multiplication
/	Division
+	Addition
-	Subtraction

Fig.8.1 Arithmetic operators

Example:

```
Update Emp_Salary
Set salary=salary* 1.05;
```

- **Comparison Operators**

These are used to compare one expression with another. The comparison operators are given below:

Operator	**Definition**
=	Equality
!=, <>, -, =	Inequality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
IN	Equal to any member of set
NOT IN	Not equal to any member of set
Is NULL	Test for nulls
Is NOT NULL	Test for anything other than nulls
LIKE	Returns true when the first expression matches the pattern of the second expression
ALL	Compares a value to every value in a list
ANY SOME	Compares a value to each value in a list
EXISTS	True if sub query returns at least one row
BETWEEN x and y	>=x and <=y

Fig.8.2 Comparison operators

Examples:

1) Get the name of students who have secured first class.

```
→ select student_name
  from student
  where percentage > = 60 and percentage < 67;
```

2) Get the out of state students name
→ select student_name
from student
where state <> 'Maharashtra';

3) Get the names of students living in 'Pune'
→ select student_name
from student
where city = 'Pune' ;

4) Display the names of students with no contact phone number.
→ select student_name
from student
where Ph_No is NULL;

5) Display the names of students who have secured second class in exam.
→ select student_name
from student
where percentage BETWEEN 50 AND 55;

- **Logical Operators:**

A logical operator is used to produce a single result from combining the two separate conditions. Following figure shows logical operators and their definitions.

Operator	Definition
AND	Returns true if both component conditions are true; otherwise return false.
OR	Return true if either component condition is true; otherwise returns false.
NOT	Returns true if the condition is false; otherwise returns false.

Fig 8.3 Logical operators

Examples:
1) Display the names of students living in Pune and Bombay.
→ select student_name
from student
where city = 'Pune' or city = 'Bombay';

2) Display the names of students who have secured higher second class in exam.
→ select student_name
from student
where percentage > = 55 and percentage < 60 ;

- **Set Operators:**

Set operators combine the results of two separate queries into a single result. Following table shows different set operators with definition.

Operator	Definition
UNION	Returns all distinct rows from both queries
INTERSECT	Returns common rows selected by both queries
MINUS	Returns all distinct rows that are in the first query, but not in second one.

Fig. 8.4 Set Operators

Examples:
Consider following two relations-
Permanent_Emp (Emp_Code, Name, Salary}
Temporary_emp = { Emp_Code, Name, Daily_wages}

1) Display name of all employees.
→ select Name
from Permanent_Emp
Union
select Name
from Temporay_Emp;

- **Operator Precedence:**

Precedence defines the order that the DBMS uses when evaluating the different operators in the same expression. The DBMS evaluates operators with the highest precedence first before evaluating the operators of lower precedence. Operators of equal precedence are evaluated from the left to right.

Fig. 8.5 shows the order of precedence.

Operator	Definition
:	Prefix for host variable
,	Variable separator
( )	Surrounds subqueries
“	Surrounds a literal
“ “	Surrounds a table or column alias or literal text
( )	Overrides the normal operator precedence
+ , -	Unary operators
* , /	Multiplication and division
+ , -	Addition and subtraction
||	Character concatenation
NOT	Reverses the result of an expression
AND	True if both conditions are true
OR	True if either conditions are true
UNION	Returns all data from both queries

INTERSECT	Returns only rows that match both queries
MINUS	Returns only row that do not match both queries

Fig. 8.5 Operator precedence

8.5 DATA DEFINITION COMMANDS

The standard query language for relational database is SQL (Structured Query Language). It is standardized and accepted by ANSI (American National Standards Institute). SQL, is a fourth-generation high-level nonprocedural language, using a nonprocedural language query, a user requests data from the DBMS. The SQL language uses English – like commands such as CREATE, INSERT, DELETE, UPDATE, and DROP. The SQL language is standardized and its syntax is same across most DBMS packages.

➢ **Different types of SQL commands are:**

- Data retrieval retrives data from the database, for example SELECT.
- Data manipulation Language (DML) inserts new rows, changes existing rows, and removes unwanted rows, for example INSERT, UPDATE, and DELETE.
- Data Definition Language (DDL) creates, changes, and removes a table structure, for example, CREATE, ALTER, DROP, RENAME, and TRUNCTATE.
- Transaction Control manages and changes logical transactions. Transactions are changes made to the data by DML statements grouped together, for example, COMMIT, SAVE POINT, and ROLLBACK.
- Data control Language (DCL) gives and removes rights to database objects, for example GRANT, and REVOKE.

➢ **SQL DDL**

SQL DDL is used to define relational database of a system. The general syntax of SQL sentence is:

VERB (parameter 1, parameter 2,…., parameter n);

In above syntax, parameters are separated by commas and the end of the verb is indicated by a semicolon. The relations are created using CREATE verb.

The different DDL commands are as follows:

- CREATE TABLE
- CREATE TABLE …AS SELECT
- ALTER TABLE …….ADD
- ALTER TABLE ……MODEIFY
- DROP TABLE

1) Create Table:

This command is used to create a new relation and the corresponding syntax is:

```
CREATE TABLE relation _name
(field 1 data type (size), field 2 data type (size)….., fieldn data type (size);
```

➢ **Example:**

The modern Book House mostly supplies books to institutions which frequently buy books from them. Various relations used are Customer, Sales Book Author, and Publisher. Design database scheme for the same.

1) The customer table definition is as follows:

```
SQL > create table Customer
            (Cust_no varchar (4) primary key,
             Cust_ name varchar (25), Cust_add varchar(30),
             Cust_ ph varchar(15);
```

Table created.

2) The sales table definition is as follows:

```
SQL > create table Sales
          (Cust_n0 varchar (4) , ISBN varchar (15),
           Qty number (3),
           Primary key (Cust_no, ISBN);
```

Table created.

3) The book table definition is as follows:

```
SQL > create table book
          (ISBN varchar (15) primary key,
          Title varchar (25), Pub_year varchar (4),
           Unit_price number (4),
           Author_name varchar (25);
```

Table created.

4) The author table definition is as follows:

```
SQL > create table Author
        (Author_name varchar (25) primary key,
        Country varchar (15);
```

2) CREATE TABLE …. AS SELECT

This type of create command is used to create the structure of new table from the structure of existing table.

The generalized syntax of this form is shown in below:

```
CREATE TABLE relation_name 1
( field1, field2,….,fieldn)
AS SELECT field1, field2,…..fieldn
FROM relation_name2;
```

Example:
Create the structure for special customer from the structure of Customer table.

The required command is shown below.

```
SQL > create table Special_customer
        (Cust_no, Cust_name, Cust_add )
        As select Cust_no, Cust_name, Cust_add
        From Customer;
```

Table created.

3) ALTER TABLE …..ADD……..

This is used to add some extra columns into existing table. The generalized format is given below.

```
ALTER TABLE relation_name
ADD (new field1 datatype (size),
New field 2 datatype (size) ,…..,
New field n datatype (size)) ;
```

Example:
ADD customer phone number and fax number in the customer relation.

```
SQL > ALTER TABLE Customer
        ADD (Cus_ph_no varchar (15),
        Cust_fax_no varchar (15);
```

Table created.

4) ALTER TABLE …… MODIFY:

This form is used to change the width as well as data type of existing relations. The generalized syntax of this shown below.

```
ALTER TABLE relation_name
MODIFY (field1 new data type (size),
            field2 new data type (size),
             ----------
            fieldn new data type (size);
```

Example:
Modify the data type of the publication year as numeric data type.

```
SQL > ALTER TABLE Book
        MODIFY (Pub-year number (4);
```

Table created.

Restrictions of the Alter Table:

Using the alter table clause you cannot perform the following tasks:

- Change the name of the table
- Change the name of the column
- Drop a column
- Decrease the size of a column if table data exists.

5) DROP TABLE:

This command is used to delete a table. The generalized syntax of this form is given below:

DROP TABLE relation-name

Example: Write the command for deleting special-customer relation.

SQL > DROP TABLE Special_customer

Table dropped**.**

6) Renaming a Table:

You can rename a table provided you are the owner of the table. The general syntax is:

RENAME old table name TO new table name;

Example:

SQL > RENAME Test To Test_info;

Table renamed.

7) Truncating a Table:

Truncating a table is removing all records from the table. The structure of the table stays intact. The SQL language has a DELETE statement which can be used to remove one or more (or all) rows from a table. Truncation releases storage space occupied by the table, but deletion does not. The syntax is :

TRUNCATE TABLE table name;

Example:

SQL > TRUNCATE TABLE Student;

8) Indexes:

Indexes are optional structures associated with tables. We can create indexes explicitly to speed up SQL statement execution on a table.

Similar to the indexes in books that help us to locate information faster, an index provides faster access path to table data.

The index points directly to the location of the rows containing the value. Indexes are the primary means of reducing disk I/O when properly used. We create an index on a column or combinations of column.

Types of index:
These are two types of index:

1) **Simple index:** It is created on multiple columns of a table.

Example:

```
Create index Item_index on order_details (Item_code);
```

2) Composite index: It is created on multiple columns of a table.

Example:

```
Create index OrdernoItem_index on Order_detail (Order_no, Item-code);
```

The indexes in the above examples do not enforce uniquencess i.e. the column included in the index can have duplicate values. To create a unique index, the key word 'unique' should be included in 'create index' command.

Example:

```
Create unique index client_index on Client_master (Client_no);
```

When the user defines a primary key or a unique key constraint, Oracle automatically creates unique indexes on the primary key column or unique key.

Dropping indexes:
An index can be dropped by using the 'drop index' command.

Example:

```
Drop index client_index;
```

When the user drops the primary key, unique key constraint or the table, Oracle automatically drops the indexes on the primary key column, unique key or the table itself.

8.6 SET OPERATIONS

The SQL operations **union, intersect,** and **except** operate on relations and correspond to the relational-algebra operations ∪,∩, and.

Consider two tables:

i) Depositor (Customer_name, Account_no)

ii) Borrower (Customer_name, Loan_no)

```
SQL > select * from Depositor;
```

Output:

Customer_name	Account_no
Johan	1001
Sita	1002
Vishal	1003
Ram	1004

```
SQL> select * from Borrower;
```

Output:

Customer_name	Loan_no
Johan	2001
Tonny	2003
Rohit	2004
Vishal	2002

8.6.1 The Union Operation:

Union clause merges the output of two or more queries into a single set of rows and column.

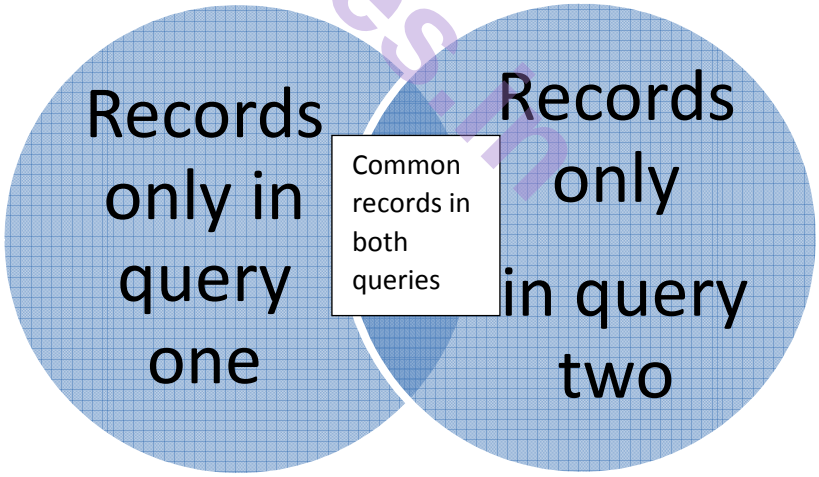


Fig 8.6.1 Output of union clause

Output = Records only in query one + records only in query in query two + A single set of records which is common in both queries.

Example: Find all customers having a loan, an account, or both at the bank.

```
SQL> select customer_name
```

```
from Borrower
union
select Custmer_name
from Depositor;
```

Output:

Customer_name
John
Ram
Rohit
Sita
Tonny
Vishal

The union operation automatically eliminates duplicates.
If we want to retain all duplicates, we must write union all in place of union.

```
SQL > select Customer_name
      from Borrower
      union all
      select Customer_name
      from Depositor;
```

Output:

CUSTOMER_NAME
John
Tonny
Rohit
Vishal
John
Sita
Vishal
Ram

The restrictions on using a union are as follows:

- Number of columns in all the queries should be same.
- The data type of the column in each query must be same.
- Union cannot be used in sub-queries.
- You cannot use aggregate functions in union.

8.6.2 The Intersect Operation:

The Intersect clause output only rows produced by both the queries intersected the intersect operation returns common records from the output of both queries.

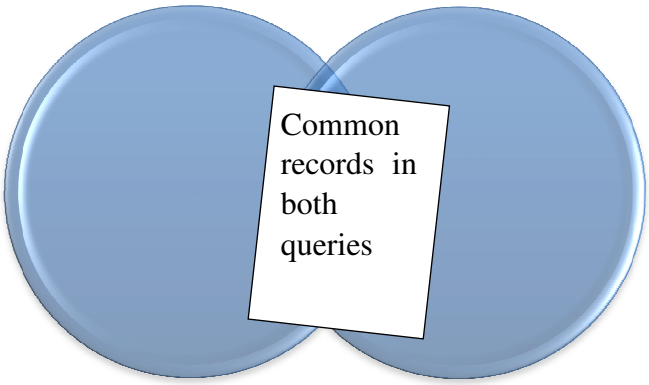


Fig.8.6.2 Output of intersect clause

Output = A single set of records which are common in both queries.

Example: Find all customers who have an account and loan at the bank.

```
SQL > select Customer_name
        from Depositor
        intersect
        select Customer_name
        from Borrower;
```

Output:

Customer_name
John
Vishal

The **intersect** operation automatically eliminates duplicates. If we want retain all duplicates we must use **intersect** all in place of **intersect.**

```
SQL > select Customer_name
        from Depositor
        intersect all
       select Customer_name
       from Borrow;
```

8.6.3 The Except Operation:

The Except also called as **Minus** outputs rows that are in first table but not in second table.

Output = Records only in query one.

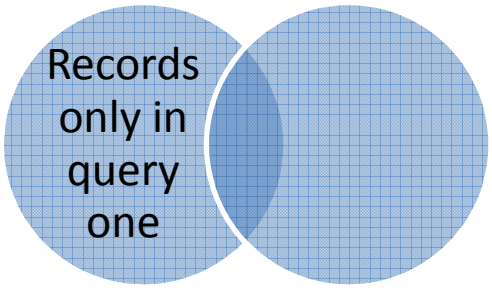


Fig. 8.6.3 Output of the except (Minus) clause

Example: Find all customers who have an account but no loan at the bank.

```
SQL > select Customer_name
      from Depositor
      minus
      select Customer_name
      from Borrower;
```

Output:

Customer_name
Ram
Sita

8.7 IMPORTANT QUESTIONS AND ANSWERS

Q.1 Consider the following relational schemas:
EMPLOYEE_NAME STREET, CITY)
WORKS (EMPLOYEE_NAME , COMPANYNAME, SALARY)
COMPANY (COMPANY_NAME, CITY)
Specify the table definitions in SQL.

Ans. ;

1) CREATE TABLE EMPLOYEE
(EMPLOYEE_NAME VARCHAR2(20) PRIMARY KEY,
STREET VARCHAR2(20),
CITY VARCHAR2(15);

2) CREATE TABLE COMPANY
(COMPANY_NAMEVARCHAR2(50) PRIMARY KEY,
CITY VARCHAR2(15);

3) CREATE TABLE WORK
(EMPLOYEE_NAME VARCHAR2(20)
REFERENCES EMPLOYEE EMPLOYEE_NAME,
(COMPANY_NAMEVARCHAR2(50)
REFERENCES COMPANY COMPANY_NAME,
SALARY NUMBER(6),
CONSTRAINT WORK_PK PRIMARY KEY (EMPLOYEE_NAME,
COMPANY_NAME));

Q.2 Consider the relations defined below:

PHYSICIAN (regno, name, telno, city)

PATIENT (pname, street, city)

VISIT (pname, regno, date_of_visit, fee)

Where the regno and pname identify the physician and the patient uniquely
Respectively. Express queries (i) to (iii) in SQL.

i) Get the name and regno of physicians who are in Mumbai.

ii) Find the name and city of patient(s) who visited a physician on 01 August 2012.

iii) Get the name of the physician and the total number of patients who have visited her.

iv) What does the following SQL query answer

```
SELECT DISTINCT name
FROM PHYSICAN P
WHERE NOT EXISTS
(SELECT *
FROM VISIT
WHERE regno = p.regno)
```

Ans.:

i) Select name, regno from PHYSICIAN where city = 'Mumbai',

ii) Select pname, city from PATIENT, VISIT where PATIENT.pname = VISIT .pname and date_of_visit = '01-Aug-12';

iii) select name, count(*) from PHSICAN, VISIT

where PHYSICIAN. Regno = VISIT.regno

group by Physician. Regno;

iv) This will give the name of physicians who have not visited any patient.

Q.3 Consider the following relations:

BRANCH (bno, street, area, city, pcode, Tel_no, Fax_no)

STAFF(Sno, Fname, Lname, address, position, salary, bno)

Express the following queries in SQL :

i) List the staff who work in the branch at 'Main Bazar'

ii) Find staff whose salary is larger than the salary of every member of staff at branch B1.

Ans.:

i) Select Fname, Lname
from STAFF, BRANCH

ii) Select Fname, Lname
from STAFF
where salary > (select max (salary) from Staff where bno ='B1');

Q.4 Consider the relations given below

Borrower (id_no, name)

Book (accno, title, author borrower_idno)

a) Define the above relations as tables in SQL making real world assumptions about the type of the fields. Define the primary keys and the foreign keys.

b) For the above relations answer the following queries in SQL

i. What are the titles of the books borrowed by the borrower whose id-no in 123.

ii. Find the numbers and names of borrowers who have borrowed books on DBMS in ascending order in id_no.

iii. List the names of borrowers who have borrowed at least two books.

Ans.:

a)

1) Create table Book
(Accno int Primary Key,
Title char(30), author char(30),
Borrow-idno int references Borrower_id.no);
2) Create table borrower
(id-no int Primary Key,
Name char(30));

b)

i) Select title from Book
where borrower.id=123

ii) Select id_no, name
from borrower, Book
where Borrower.id_no = Book.borrower_idno
and title ='DBMS'
order by id_no asc;

iii) Select name
from Borrower, Book
where Borrower.id_no = book.Borrower_id_no
having count (*) > 2;

Q.5 Describe substring comparison in SQL. For the relation Person (name, address), write a SQL query which retrieves the names of people whose name begins with 'A' and address contains 'Pune'.

Ans.:

i) **SUBSTR** is used to extract a set of charters from a string by specifying the character starting position and end position and length of characters to be fetched.

Example:

Substr(hello', 2,3);
will return 'ell.

ii) Select name
from Person
where name like 'A%'and address 'Pune';

Q.6 Consider the following relations:
S(S#, SNAME, STATUS, CITY)
SP(S#, P#, QTY)
P(P#, PNAME, COLOR, WEIGHT, CITY)

Give an expression in SQL for each of queries below:

i) Get supplier names for supplier who supply at least one red part

ii) Get supplier names for supplier who do not supply part P2.

Ans.:

i)

```
SELECT SNAME FROM S
WHERE S# IN (SELECT S# FROM SP
WHERE P# IN (SELECT P# FROM P
WHERE COLOR = RED'));
```

ii)

```
SELECT SNAME FROM S
WHERE S# NOT IN (SELECT S# FROM SP WHERE P# = 'P2')
```

8.8 SUMMARY

All tasks related to relational data management creating tables, querying the database for information, modifying the data in the database, deleting them, granting access to users and so on can be done using SQL.SQL specifies what is required and not how it should be done.

There are four kinds of literal values supported in SQL. They are Character string, Bit string, Exact numeric and Approximate numeric. Data definition language is used to create, alter and delete database objects.

Arithmetic operators are used in SQL expressions to add, subtract, multiply, divide and negate data values. The result of this expression is a number value.

A logical operator is used to produce a single result from combining the two separate conditions. Set operators combine the results of two separate queries into a single result. Precedence defines the order that the DBMS uses when evaluating the different operators in the same expression.

Data retrieval retrives data from the database, for example SELECT.

Data Definition Language (DDL) creates, changes, and removes a table structure, for example, CREATE, ALTER, DROP, RENAME, and TRUNCTATE.

ALTER TABLE …..ADD……..is used to add some extra columns into existing table. **ALTER TABLE …… MODIFY** is used to change the width as well as data type of existing relations **.DROP TABLE** command is used to delete a table. Truncating a table is removing

all records from the table. The structure of the table stays intact. We can create indexes explicitly to speed up SQL statement execution on a table.

8.9 UNIT END QUESTIONS

1) Write short note on : SQL
2) Explain different characteristics of SQL.
3) Explain various advantages of SQL.
4) Explain basic structure of SQL.
5) Explain SQL data types.
6) Explain different SQL DDL commands.
7) Explain different types of SQL commands.
8) Explain various set operations in SQL.

Text Books:

1. Korth, Slberchatz, Sudarshan, Database System Concepts, 6th Edition, McGraw Hill
2. Elmasri and Navathe, Fundamentals of Database Systems, 6thEdition, Pearson education
3. Raghu Ramkrishnan and Johannes Gehrke, Database Management Systems, TMH

References:

1. Peter Rob and Carlos Coronel, ―Database Systems Design, Implementation and Management‖, Thomson Learning, 9thEdition.

2. G. K. Gupta : "Database Management Systems", McGraw – Hill.

*****

UNIT III

9

STRUCTURED QUERY LANGUAGE
Part-II

Unit Structure

9.0 OBJECTIVE

- To apply Aggregate functions to different SQL functions.
- To apply data manipulation commands to solve the problem.
- To understand and apply data control commands.

9.1 INTRODUCTION

All tasks related to relational data management creating tables, querying the database for information, modifying the data in the database, deleting them, granting access to users and so on can be done using SQL.

9.2 AGGREGATE FUNCTIONS

Aggregate functions are functions that take a collection of values as input and return a single value. SQL offers five built-in aggregate functions:

- Average: **avg**
- Minimum : **min**
- Maximum : **max**
- Total : **sum**
- Count : **count**

1. avg:

Syntax : avg ([Distinct | All] n)
Purpose: Returns average value of n, ignoring null values.

Example:

```
SQL > select avg (Unit- price) "Average Price"
          From Book;
```

Output:

Average Price
359.8

2. min
Syntax : min ([Distinct | All] expr)
Purpose: Return minimum value of expression

Example :

```
SQL >  select min(Unit_price ) " Minimum Price"
       From Book;
```

Output :

Minimum Price
250

3. max
Syntax : max ([Distinct | All] expr)
Purpose : Return maximum value of expression

Example:

```
SQL >  select max(Unit_price ) " Maximum Price"
       From Book;
```

Output:

Maximum Price
450

4. sum
Syntax : sum ([Distinct | All] n)
Purpose : Return sum of values of n.

Example:

```
SQL >  select sum(Unit_price ) " Total"
       From Book;
```

Output :

Total
1799

5. count
Syntax : count ([Distinct | All] expr)
Purpose : Returns the number of rows where expression is not null.

Example :

```
SQL >  select count(Title) " No. of Books"
       From Book;
```

Output:

No. of Books
5

9.3 NULL VALUES

SQL allows the use of null values to indicate absence of information about the value of an attribute.

We can use the special keyword null in a predicate to test for null value.

Let no. of records of customer relation are :

```
SQL > select * from Cust;
```

Output:

Cust-no	Cust_name	Cust_ph_no
C101	Telco	5346772
C102	Bajaj	5429810
C103	Mahindra	

Example: Final all customers from relation with null values for cust_ph_no field.

```
SQL > select Cust_name
      from Cust
      where Cust_ph_no is null;
```

Output:

Cust_name
Mahindra

Not Null:
The predicate **is not null** tests for the absence of a null value.

Example: Find all customers from customer relation where phone is not null.

```
SQL > select cust_name
from cust
where cust_ph_no is not null;
```

Output:

Cust_name
Telco
Bajaj

9.4 DATA MANIPULATION COMMANDS

9.4.1 Basic Structure:

The basic structure of an SQL expression consists of three clauses: Select, from, and where.

- The **select** clause corresponds to projection operation of the relational algebra. It is used to list the attributes desired in the result of query.
- The **from** clause corresponds to the Cartesian product operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.
- The **where** clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the from clause.

A typical SQL query has the form:

```
Select A1, A2,……, An
                    from r1,r2,……..,rn
                     where p;
```

```
Where,
        Ai -  respresents an attribute
```

r_i - representing relation p - is a predicate

SQL forms the Cartesian product of the relations named in the from clause, performs a relational algebra selection using the **where** clause predicate, and then projects the result onto the attributes of the **select** clause.

9.4.1.1 The select Clause:

This command is used to display all fields/or set of selected fields for all/ selected records from a relation.

Different forms of select clause are given below:

- **Form 1: Use of select clause for displaying selected fields**

Example: Find the names of all publishers in the book relation.

```
SQL > Select publisher_name from book;
```

Output:

```
PUBLISHER_NAME
PHI
Technical
Nirali
Technical
SciTech
```

Above query displays all publisher_name, from Book relation. Therefore, some publishers name will get displayed repeatedly. SQL allows duplicates in relations as well as in the results of SQL expressions.

Form 2: Use of select for displaying distinct values

For elimination of duplicates the keyword **distinct** is used. The above query is rewritten as,

```
SQL > select distinct publisher_name from book;
```

Output:

```
PUBLISHER_NAME
Nirali
PHI
SciTech
Technical
```

SQL allows us to the keyword **all** to specify explicitly that duplicates are not removed.

```
SQL > select all publisher_name from book;
```

Output:

PUBLISHER_NAME
PHI
Technical
Nirali
Technical
SciTech

Form 3: Use of select for displaying all fields:

The asterisk symbol "*" can be used to denote "all attributes." A select clause of the form select * indicates that all attributes of all relations appearing in the from clause are selected.

Example:

```
SQL > select * from book;
```

Output:

ISBN	TITLE	PUB_ YEAR	UNIT_ PRICE	AUTHOR _NAME	PUBLISHER_ NAME
1001	Oracle	2004	399	Arora	PHI
1002	DBMS	2004	400	Basu	Technical
2001	DOS	2003	250	SInha	Nirali
2002	ADBMS	2004	450	Basu	Technical
2003	Unix	2000	300	Kapoor	SciTech

```
SQL > select * from author;
```

Output:

AUTHOR_NAME	COUNTRY
Arora	U.S.
Kapoor	Canada
Basu	India
Sinha	India

```
SQL > select * from publisher;
```

PUBLISHER_NAME	PUB_ADD
PHI	Delhi
Technical	Pune MainBazar
Nirali	Mumbai
SciTech	Chennai

Form 4 : Select clause with arithmetic expression:

The select clause may also contain arithmetic expressions involving the operators +, -, *, and / operating on constants or attributes of tuples.

Example:

```
SQL > select title, unit_price* 10 from book;
```

Output:

TITLE	UNIT_PRICE *10
Oracle	3990
DBMS	4000
DOS	2500
ADBMS	4500
Unix	3000

The above query returns a relation that is the same as the book relation with attributes title as it is and unit_price will get multiplied by 10.

9.4.1.2 The where Clause:

The where clause is used to select specific rows satisfying given predicate.

Example: "Find the titles of books published in year 2004."

This query can be written in SQL as:

```
SQL > select title from book where pub_year = '2004' ;
```

Output:

TITLE
Oracle
DBMS
ADBMS

SQL uses the logical connective **and, or** and **not** in the where clause. The operands of logical connectives can be expressions involving the comparison operators <, <=, >,>=, =, and <>. SQL allows us to use the comparison operators to compare strings and arithmetic expressions, as well as special types, such as date types.

Between:

SQL includes a between comparison operator to specify that a value be less than or equal to some value and greater than or equal to some other value.

Example: "Find the titles of book having price between 300 to 400".

```
SQL> select Title from Book
        where Unit_price between 300 and 400;
```

Output:

TITLE
Oracle
DBMS
Unix

Or

```
SQL > select Title from Book
        where Unit_price >= 300 and Unit_price < = 400;
```

Output:

TITLE
Oracle
DBMS
Unix

Similarly, we can use the **not between** comparison operator.

9.4.1.3 The from Clause:

The from clause defines a Cartesian product of the relations in the clause.

Example: "Find the titles of books with author name and country published in year2004".

```
SQL > select Title, Book. author_name, Country
  from Book, Author
  where Book.author_name = Author.author_name
  and pub_year = '2004'.
```

Output:

TITLE	AUTHOR_NAME	COUNTRY
Oracle	Arora	U.S.
DBMS	Basu	India
ADBMS	Basu	India

Notice that SQL uses the notation relation_name to avoid ambiguity in cases where an attribute appears in the schema of more than one relation.

9.4.2 Rename Operation:

SQL provides a mechanism for renaming both relations and attributes. It uses the as clauses taking the form:
Old-name as new-name

The **as** clause can appear in both the select and from clause.

Example:

```
SQL > select Title, Unit_price * 1.15 as New_price
      from Book;
```

Output:

TITLE	NEW_PRICE
Oracle	458.85
DBMS	460
DOS	287.5
ADBMS	517.5
Unix	345

The above query calculates the new price as Unit_price * 1.15 and display the result under the column name NEW_PRICE.

9.4.3 Tuple Variable:

Tuple variables are defined in the **from** clause by the way of **as** clause.

Example: "Find the titles of Books with author name and author country." The above query is written in SQL using tuple variable **as:**

```
SQL > select Title, B.Author A_name, Country
 from Book B, Author A
 where B.Author_name = A.Author_name;
```

Output:

TITLE	AUTHOR_NAME	COUNTRY
Oracle	Arora	U.S.
DBMS	Basu	India
DOS	SInha	India
ADBMS	Basu	India
Unix	Kapoor	Canada

Tuple variables are most useful for comparing two tuples in the same relation. Suppose we want to display the book titles that have price greater than at least one book published in year 2004.

```
SQL > select distinct B1. Title
 from Book B1, Book B2
  where B1. Unit_price > B2. Unit_price
  and B2. Pub_year = '2004';
```

Output:

TITLE
ADBMS
DBMS

9.4.4 String Operations:

SQL specifies strings by enclosing them in single quotes, for example: 'DBMS'. A single quote character that is part of a string can be specified by using two single quote characters; for example the string : "It's right" can be specified by "Its right".

The most commonly used operation on string is pattern matching using the operator **like.** We describe patterns by using two special characters.

- Percent (%): The % character matches any substrings.
- Underscore: (_): The _ character matches any character.

Patterns are case sensitive, that is, upper case characters do not match lowercase character or vice-versa.

Example:

- 'Computer%' – matches any string beginning with 'computer'.
- '%Engg' – matches any string containing "Engg" as a substring, for example, "Computer Engg department", Mechanical Engg".
- '-s%' – matches any string with second character's'.
- '_ _ _ ' – matches any string of exactly three characters.
- '_ _ _%' – matches any string of at least three characters.

Example of SQL queries:

1) Find the names of author's from author table where the first two characters of name are 'Ba';

```
SQL > select Author_name
      from Author
      where Author_name like 'Ba%';
```

Output:

AUTHOR_NAME
Basu

2) Select author_name from author where the second character of name is 'r' or 'a';

```
SQL > select Author_name
      from Author
       Where Author_name like '_r%' or Author_name like '_a% ' ;
```

Output:

AUTHOR_NAME
Arora
Kapoor
Basu

3) Display the names of all publishers whose address includes the substring 'Main';

```
SQL > Select  Publisher_name
      from Publisher
      where Pub-add like '%Main%';
```

Output:

PUBLISHER_NAME
Technical

9.4.5 Ordering the Display of Tuples:

SQL uses order by clause to display the tuples in the result of the query to appear in sorted order.

Example:

1) Display all titles of books with price in ascending order of titles

```
SQL > select Title, Unit_price
      from Book
      order by Title;
```

Output:

TITLE	UNIT_PRICE
ADBMS	450
DBMS	400
DOS	250
Oracle	399
Unix	300

2) Display all titles of books with price, and year in decreasing order of year.

```
SQL > select Title, Unit_price, Pub_year
      from Book
      order by Pub_year desc;
```

Output:

TITLE	UNIT_PRICE	PUB_YEAR
Oracle	399	2004

DBMS	400	2004
ADBMS	450	2004
DOS	250	2003
Unix	300	2000

9.4.6 Group by:

Group by clause is used to group the rows based on certain criteria. For example, you can group the rows in the Book table by the publisher name, or in an exmployee table, you can group the employees by the department and so on. **Group by** is usually used in conjuction with aggregate functions like **sum, avg, min, max,** etc.

Examples:

1) Display total price of all books (publisher wise).

```
SQL > select Publisher_name, sum (Unit_price) "Total Book Amount"
      from Book
      group by Publisher_name;
```

Output:

Publisher name	Total Book Amount
Nirali	250
PHI	399
SciTech	300
Technical	850

2) "On every book author receives 15% display royalty. Display total royalty amount received by each author till date".

```
SQL > select Author_name, sum (Unit_price * 0.15) "Royalty Amount"
 from Book
 group by Author_name;
```

Output:

Author_name	Royalty Amount
Arora	59.85
Basu	127.5
Kapoor	45
Sinha	37.5

9.4.7 Having:

The **having** clause tells SQL to include only certain groups produced by the **group by** clause in the query result set. **Having** is

equivalent to the **where** clause and is used to specify the search criteria or search condition when **group by** clause is specified.

Examples:

1) Display publisher wise total price of books published, except for publisher 'PHI'.

```
SQL > select Publisher_name, sum(Unit_price) "Total Book Amount"
      from Book
      group by Publisher_name
      having Publisher_name <> 'PHI';
```

Output:

Publisher_name	Total Book Amount
Nirali	250
SciTech	300
Technical	850

2) Display royalty amount received by those authors whose second character name contains 'a'.

```
SQL > select Author_name, sum (Unit_price * 0.15) "Royalty Amount"
 from Book
 group by Authour_name
 having Author_name like '_a%';
```

Output:

Author_name	Royalty Amount
Basu	127.5
Kapoor	45

9.5 DATABASE MODIFICATION

Different operations that modify the contents of the database are:

- Delete
- Insert
- Update

9.5.1 Delete:

A delete request is expressed in same way as a query. The delete operation is used to delete all or specific rows from database. We cannot delete values of particular attributes.

Syntax:

```
delete from r
 where p
```

Where,
r- relation
p- predicate

The **delete** statement first finds all tuples in r for which P (t) is true, and then deletes them from r. The **where** clause is omitted if all tuples in r are to be deleted.

A **delete** command operates only on one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation.

Examples:

1) Delete all books from Book relation where publishing year is less than 1997.

```
SQL > delete from Book
  Where Pub_year < 1997;
```

2) Delete all books from Book relation where publishing year is between 1997 to 1999.

```
SQL > delete from Book
  Where Pub_year between 1997 and 1999.
```

3) Delete all books of authors living in country 'U.K.' from Book relation

```
SQL > delete from Book
    where Author_name in
    (select Author_name
     from Author
          where Country = 'U.K.');
```

4) Delete all books having price less than average price of books.

```
SQL > delete from Book
          where Unit_price <
        (select avg (Unit_price)
         from Book);
```

5) Delete all books from Book relation.

```
SQL > delete from Book;
```

9.5.2 Insertion:

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. The

attribute values for inserted tuples must be members of the attribute's domain.

Consider following customer relation.

Customer= { Cust_no, Cust_name, Cust_add, Cust_ph}

Example: Insert a new customer record with customer record with customer no. as 05, customer_Name as 'Pragati', Address- 'ABC' and ph. no.as 9822398910.

```
SQL > insert into Customer
 values (05, 'Pragati', ABC',  9822398910);
```

In above example, the values are specified in the order in which the corresponding attributes are listed in the relation schema. SQL allows the attributes to be specified as part of the insert statement.

Above query is rewritten as:

```
SQL > insert into customer (cust_no, cust_name, cust_address,
cust_ph)
Values (05,'Pragati', ABC',  9822398910);
```

We can insert tuples in a relation on the basis of the result of a query.

Consider two relations:

- Student = {Roll_No, Name, Flag}
- Defaulter_student = {Roll_No, Name}

Supposed we want to inster tuples from Student relation to Defaulter_student where flag is 'D'.

The query for above statement is:

```
SQL > insert into Defaulter_student
        select Roll_no, Name
        from Student
        where falg = 'D',
```

It is also possible to assign values only to particular attributes while inserting a tuple

Example:

```
SQL > insert into Customer
      Values (05, 'Pragati', 'ABC', null);
```

9.5.3 Updates:

Using **update** statement, we can change value in a tuple or all the tuples of a relation. Consider following relation;

Employee = { Emp_code, Name, Salary};
No. of records of employee relation are:

```
SQL > select * from Employee;
```

Output:

Emp_code	Name	Salary
1	Ram	10000
2	Jim	7000
3	John	9000
4	Sita	11000

Examples:

1) Increase salary of all employees by 15%

```
SQL > update Employee
Set Salary =Salary * 1.15;
```

Rows updated.

The updated salary values are:

```
SQL > select * from Employee;
```

Output:

Emp_code	Name	Salary
1	Ram	11500
2	Jim	8050
3	John	10350
4	Sita	12650

2) Increase salary of employees who earn less than 9,000 by 15%

```
SQL > update Employee
 Set Salary = Salary* 1.15;
 Where Salary < 9000;
```

1 row updated.

The updated salary values are:

```
SQL > select * from Employee;
```

Output:

Emp_code	Name	Salary
1	Ram	10000
2	Jim	8050
3	John	9000
4	Sita	11000

3) Increase salary of employees by 15% who earn less than average salary of employees.

```
SQL > update Employee
        set Salary = Salary * 1.15
        where Salary < (Select avg( Salary)
        from Employee);
```

2. rows updated.

The updated salary values are:

```
SQL > select * from Employee;
```

Output:

Emp_code	Name	Salary
1	Ram	10000
2	Jim	8050
3	John	10350
4	Sita	11000

4) Increase salary of employees by 15% who earn less than 9000 and for remaining employees give 5% salary raise.

For above example, we require two update statements:

```
i)  update Employee
    set Salary = Salary * 1.15
     where Salary < 9000;
ii) update Employee
    set Salary = Salary * 1.05
      where Salary >= 9000;
```

SQL provides a **case** construct, which we can use to perform both updates with a single update statement.

```
update employee
set Salary = case
when Salary < 9000 than
  Salary * 1.15
      else  Salary * 1.05
end;
```

The general form of case statement is:

```
Case
     when pred 1 then result 1
      when pred 2 then result 2
      - - - - - - -
      when pred n then result
end else result
```

The operation returns result i; where I is the first of pred 1, pred 2, …., pred n, that is satisfied ; if none of the predicate is satisfied the operation returns result 0.

9.6 DATA CONTROL COMMANDS

Transaction control commands manage changes made by DML commands.

Collection of operation that forms a single logical unit of work are called Transactions.

A transaction can either be one DML statement or a group of statements. When managing groups of transactions, each designated group of transactions must be successful as one entity or none of them will be successful.

The Following list describes the nature of transactions:

- All transactions have a beginning and an end.
- A transaction can be saved or undone.
- If a transaction fails in the middle, no part of the transaction can be saved to the database.

Transactional control is the ability to manage various transactions that may occur within a relational database management system. Different types of transaction control commands are:

1) COMMIT Command:

The commit command saves all transactions to the database since the last COMMIT or ROLLBACK command was issued.

2) ROLLBACK Command:

The rollback command is the transactional control command used to undo transactions that have not already been saved to the database. The rollback command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

3) SAVEPOINT Command:

Establishes a point back to which you may roll.

4) SET TRANSACTION Command:

Establish properties for the current transaction.

9.7 IMPORTANT QUESTIONS AND ANSWERS

Q.1 Give an expression in SQL for each of queries below:

1) Find the names of all employees who work for City Bank Corporation.

2) Find the names and company names of all employees sorted in ascending order of company name and descending order of employee names of that company.

3) Change the city Bank Corporation to 'Delhi'

Ans.:

1) SELECT EMPLOYEE_NAME
FROM WORKS
WHERE COMPANYNAME = 'City Bank Corporation';

2) SELECT EMPLOYEE_NAME, COMPANYNAME
FROM WORKS
ORDER BY COMPANYNAME, EMPLOYEE_NAME DESC;

3) UPDATE COMPANY
SET CITY = 'Delhi'
WHERE COMPANY_NAME = 'City Bank Corporation';

Q.2 Consider the following relational database:

STUDENT (name, student#, class, major)
COURSE (course name, course#, credit hours, department)
SECTION (section identifier, course#, semester, year, instructor)
GRADE_REPORT (student#, section identifier, grade)
PRESEQUISITE (course#, presequisite#)

Specify the following queries in SQL on the above database schema.

i) Retrieve the names of all students majoring in 'CS'.

ii) Retrieve the names of all courses taught by Professor Kulkarni in 1999.

iii) Delete the record for the student whose name is 'Om' and whose student number is 12.

iv) Insert a new course <'DBMS', 'CS1390', 23, 'CS' >

Ans.:

i) SELECT NAME FROM STUDENT WHERE MAJOR = 'CS';

ii) SELECT COURSE_NAME
FROM COURSE C, SECTION S
WHERE C. COURSE# = S.COURSE#
AND INSTRUCTOR = 'Kulkarni' AND YEAR = 1999;
OR
SELECT COURSE_NAME FROM COURSE

WHERE COURSE# IN (SELECT COURSE# FROM SECTION
WHERE INSTRUCTOR = 'Kulkarni' AND YEAR = 1999);

iii) DELETE FROM STUDENT WHERE NAME = 'Om' AND STUDENT# = 12;

iv) INSERT INTO COURSE
VALUES ('DBMS', 'CS1390', 23, 'CS');

9.8 SUMMARY

SQL provides three data manipulation statements insert, update and delete. Data retrieval retrives data from the database, for example SELECT.

Data manipulation Language (DML) inserts new rows, changes existing rows, and removes unwanted rows, for example INSERT, UPDATE, and DELETE.

Transaction Control manages and changes logical transactions. Transactions are changes made to the data by DML statements grouped together, for example, COMMIT, SAVE POINT, and ROLLBACK.

Data control Language (DCL) gives and removes rights to database objects, for example GRANT, and REVOKE.

Aggregate functions are functions that take a collection of values as input and return a single value. SQL offers five built-in aggregate functions: Average, Minimum, Maximum, sum, count.SQL allows the use of null values to indicate absence of information about the value of an attribute.

The **select** clause corresponds to projection operation of the relational algebra. The **from** clause corresponds to the Cartesian product operation of the relational algebra. The **where** clause corresponds to the selection predicate of the relational algebra.

SQL provides a mechanism for renaming both relations and attributes. Group by clause is used to group the rows based on certain criteria

Having is equivalent to the **where** clause and is used to specify the search criteria or search condition when **group by** clause is specified.
Collection of operation that forms a single logical unit of work are called Transactions.

9.9 UNIT END QUESTIONS

1. Explain different SQL DML commands.

2. Explain different SQL DCL commands.
3. Explain different types of SQL commands.
4. Explain various aggregate functions available in SQL.
5. Explain the use of NULL value with example.

TEXT BOOKS

1. Korth, Slberchatz, Sudarshan, Database System Concepts, 6th Edition, McGraw Hill
2. Elmasri and Navathe, Fundamentals of Database Systems, 6thEdition, Pearson education
3. Raghu Ramkrishnan and Johannes Gehrke, Database Management Systems, TMH

REFERENCES

1. Peter Rob and Carlos Coronel, ― Database Systems Design, Implementation and Management‖, Thomson Learning, 9th Edition.
2. G. K. Gupta : "Database Management Systems", McGraw – Hill.

*****

10

VIEWS, NESTED QUERIES AND JOINS

Unit structure

10.0 OBJECTIVE

- To understand and apply different operation such as Creation, selecting and destroying View in SQL.
- To write nested and complex queries to solve the real-world problems in efficient manner.
- To understand and apply Join operations to retrieve the data from multiple tables.

10.1 INTRODUCTION

A subquery is a SELECT statement that is embedded in a clause of another SELECT statement. It is also used in complex type of query. Subqueries are written in the WHERE clause of the another SQL statement to obtain values based on the unknown conditional value. Use of a subquery is equivalent to performing two or more sequential queries. The result of inner query is fed to the outer query to display the values. A join condition is used when data from more than one table in the database is required to be displayed. Rows in one table can be joined to rows of another table according to common values existing in the corresponding columns, those usually primary key and foreign key columns.

An SQL view is a virtual table, dynamically constructed for a user by extracting data from actual base tables. A view is a tailored presentation of the data contained in one or more tables (or other views). A view takes takes the output of a query and treats it as a table ; therefore a view can be thought of as stored query or a virtual table .It provides an additional level of table security by restricting access to a predetermined set of rows and / or columns of a table. It hides data complexity and it simplifies commands for the user because they allow to select information from multiple tables without actually knowing how to perform a join.

10.2 VIEWS IN SQL

A view object that gives the user a logical view of data from an underlying table or tables. You can restrict what uses can view by allowing them to see only a few attributes columns from a table.

View may be created for the following reasons:

- Simplifies queries
- Can be queried as a base table
- Provides data security

10.2.1 Creation of Views:

Syntax:

```
CREATE VIEW viewname as
SELECT columnname, columnname
FROM tablename
WHERE columnname = expression list;
```

Example: Create view on Book table which contains two fields title, and author name.

```
SQL > create view V_Book as
        Select Title, Author_name
```

```
        from Book;
View created.
SQL > select * from V_Book;
```

Output:

TITLE	AUTHOR_NAME
Oracle	Arora
DBMS	Basu
DOS	SInha
ADBMS	Basu
Unix	Kapoor

10.2.2 Selecting Data from a View:

Example: Display all the titles of books written by author 'Basu'.

```
SQL > select Title
      from V_Book
      Where Author_name = 'Basu';
```

Output:

TITLE
DBMS
ADBMS

10.2.3 Updatable Views:

Views can also be used for data manipulation i.e the user can perform Insert, Update, and the Delete operations on the view. The views on which data manipulation can be done are called Updatable Views

Views that do not allow data manipulation are called Ready only Views. When you give a view name in the Update, Insert, or Delete statement, the modifications to the data will be passed to the underlying table.

For the view to be updatable, it should meet following criteria:

- The view must be created on a single table.
- The primary key column of the table should be included in the view.
- Aggregate functions cannot be used in the select statement.
- The select statement used for creating a view should not include Distinct, Group by, or Having clause.
- The select statement used creating a view should not include subqueries.
- It must not use constants, strings or value expressions like total / 5.

10.2.4 Destroying a View:

A view can be dropped by using the DROP VIEW command.

Syntax:

```
DROP VIEW viewname;
```

Example:

```
DROP VIEW V_Book;
```

10.3 NESTED AND COMPLEX QUERIES

10.3.1 Nested Queries:

SQL provides a mechanism for nesting subqueries. A subquery is a select from where expression that is nested within another query. Mostly subqueries are used to perform tests for set membership to make set comparisons and determine set cardinality.

10.3.1.1 Set Memberships:

SQL uses in and not in constructs for set membership tests.

i) IN:

The in connective tests for set membership , where the set is a collection of values produced by a **select** clause.

Examples:

1) Display the title, author, and publisher name of all books published in 2000, 2002 and 2004.

```
SQL > select Title, Author_name, Publisher_name, Pub_year
      from Book
      where Pub_year in ('2000', '2004') ;
```

Output:

TITLE	AUTHOR _NAME	PUBLISHER _NAME	PUB_YEAR
Oracle	Arora	PHI	2004
DBMS	Basu	Technical	2004
ADBMS	Basu	Technical	2004
Unix	Kapoor	SciTech	2000

2) Get the details of author who have published book in year 2004.

```
SQL> select * from Author
     where Author_name in (select Author_name
     from Book
     where Pub_year = '2004');
```

Output:

AUTHOR_NAME	Country
Arora	U.S
Basu	India

ii) Not IN:

The **not in** connective tests for the absence of set membership.

Examples

1) Display title, author, and publisher name of all books except those which are published in year 2002, 2004 and 2005.

```
SQL > select Title, Author_name, Publisher_name, Pub_year
        from Book
        where Pub_year not in ('2002', '2004', '2005')
```

Output:

TITLE	AUTHOR_NAME	PUBLISHER_NAME	PUB_YEAR
DOS	SInha	Nirali	2003
Unix	Kapoor	SciTech	2000

2) Get the titles of all books written by authors not living in India.

```
SQL > select Title
        from Book
        where Author_name not in (select Author_name from
from author
                                  where country = 'India');
```

Output:

TITLE
Oracle
Unix

10.3.1.2 Set Comparison:

Nested subqueries are used to compare sets. SQL uses various comparison operators such as <, < = , > , > = , =, <>, any, all, and some, etc to compare sets.

Examples:

1) Display the titles of books that have price greater than at least one book published in year 2004.

For given example, we write the SQL query as:

```
SQL > select distinct B1. Title
```

```
from Book B1, Book B2
where B1. Unit_price > B2. Unit_price
and B2. Pub_year = '2004';
```

Output:

TITLE
ADBMS
DBMS

SQL offers alternative style for writing preceding query.

The phrase 'greater than at least one' is represented in SQL by > some.

Using > some, we can rewrite the query as:

```
SQL > select distinct Title
                from Book
      where Unit_price > some (select Unit_price
                               from Book
                               where Pub_year = '2004');
```

Output:

TITLE
ADBMS
DBMS

The subquery generates the set of all unit price values of books published in year 2004. The **> some** comparison in the **where** clause of the outer select is true if the unit price value of the tuple is greater than at least one member of the set of all unit price values of books published in year 2004.

- SQL allows **< some, >some, <=some, >=some, =some, and <> some** comparisons.
- **= some** is identical to **in** and **<> some** is identical to **not in.**
- The keyword **any** is similar to **some.**

All

- The "**>all"** corresponds to the phrase 'greater than all'.
- SQL allows <all, <=all, >all, <>all, =all comparisons.
- all is identical to **not in** construct.

1) Display the titles of books that have price greater than all the books publisher in year 2004.

```
SQL > select distinct title
from Book
where Unit_price > all (select Unit_price
```

```
from Book
where Pub_year = '2004');
```

Output: no rows selected

2) Display the name of author who have received highest royalty amount.

```
SQL > select Author_name
      from Book
      group by Author_name
having sum (Unit_price * 0.15) > = all (select sum (Unit_price * 0.15)
                                        from Book
                                       group by Author_name);
```

Output:

AUTHOR_NAME
Basu

10.3.1.3 Test for Empty Relations:

Exists is a test for non empty set. It is represented by an expression of the form Exists (select ….. from ……). Such an expression evaluates to true only if the result of evaluating the subquery represented by the (select ….. from …..) is nonempty.

Consider two relations:

i) Book_Info = { Book_ID, Title, Author_name, Publisher_name, Pub_year}

ii) Order_Info = { Order_no, Book_ID, Order, Date, Qty, Price}

Records of Book _Info and Order_Info relations are:

```
SQL > select * from Book _Info;
```

Book_id	TITLE	AUTHOR_NAME	PUBLISHER_NAME	PUB_YEAR
1001	Oracle	Arora	PHI	2004
1002	DBMS	Basu	Technical	2004
2001	DOS	Sinha	Nirali	2003
2002	ADBMS	Basu	Technical	2004
2003	Unix	Kapoor	SciTech	2000

```
SQL > select * from Order_Info;
```

Output:

Order_no	Book_id	Order_date	Qty	Price
1	1001	10-10-2004	100	399
2	1002	11-01-2002	50	400

Consider the following example:
"Get the names of all books for which order is placed".

```
SQL > select Title
                    from Book_Info
                    where exists (select * from Order_Info
where Book_Info.book_id = Order_Info. Book_id);
```

Output:

TITLE
Oracle
DBMS

In above SQL, first the subquery 'select * from Order_Info where Book_Info.Book_Id = Order_Info.Book_Id' is evaluated. Then the outer query is evaluated which displays the titles of books returned by inner query.

Similar to the **exists** we can use **not exists** also.

Example: Display the titles of books for which order is not placed.

```
SQL> select Title
        from Book_info
        where not exists (select * from Order_ Info
        where Book_Info. Book_id = Order_Info.Book_Id)
```

Output:

Title
DOS
ADBMS
Unix

10.3.2 Complex Queries:

Complex queries are often hard or impossible to write as a single SQL block. There are two ways for composting multiple SQL blocks to express a complex query.

i) Derived relations

ii) With clause

10.3.2.1 Derived Relations:

SQL allows a sub query expression to be used in the from clause. If we use such as expression, then we must give the results relation a name, and we can rename the attributes. For renaming as clause is used.

Example:

```
(select Branch_name, avg (Balance)
```

```
        from Account
        group by Branch_name)
as result (Branch_name, Avg_balance);
```

This subquery generates a relation consisting of the names of all branches and their corresponding average account balances. The subquery is named as result and the attributes as **Branch_name**, and **Avg_balance**.

The use of a subquery expression in the from clause is given below:

1) "Find the average account balance of those branches where the average account balance is greater than $1200".

```
SQL > select Branch_name, avg (Balance)
        from (select Branches_name, avg(Balance)
         from Account
          group by Branch_name)
        as Branch_avg(Branch_name,avg(Balance))
        where Avg_Balance > 1200;
```

2) 'Find the maximum across all branches of the total balance at each branch.

```
SQL > select max (Tot_balance)
      from (select Branch_name, sum (Balance)
        from Account
       group by Branch_name)
    as branch_total (Branch_name, Tot_balance);
```

10.3.2.2 The with Clause:

The with clause provides a way of defining a temporary view whose definition is available only to the query in which the with clause occurs. Consider the following query, which selects accounts with the maximum balance; if there are many accounts with the same maximum balance, all of them are selected.

```
with Max_balance (value) as
select max (Balance)
from Account
select Account_number
from Account, Max_balance
where Account. Balance =
Max_balance. Value;
```

10.4 JOIN

Join is a query in which data is retrieved from two or more table. a join matches data from two or more tables, based on the values of one or more columns in each table.

Need for joins:

In a database where the tables are normalized, one table may not give you all the information about a particular entity. For example, the Employee table gives only the department ID, so if you want to know the department name and the manager name for each employee, than you will have to get the information from the Employee and Department table.

In other words, you will have to join two tables. So for comprehensive data analysis, you must assemble data from several tables. The relational model having made you to partition your data and put them in different tables for reducing data redundancy and improving data independence – relies on the join operation to enable you to perform adhoc queries that will combine the related data which resides in more than one table.

Different types of Joins are:

- Inner Join
- Outer Join
- Natural Join

10.4.1 Inner Join:

Inner Join returns the matching rows from the tables that are being joined. Consider following two relations:

i) Employee (Emp-name,Department, Salary)

ii) Employee_salary (Emp_name, Department, Salary)

These two relations are shown in Figure 10.4.1 and 10.4.2

Employee	
Emp_name	**City**
Hari	Pune
OM	Mumbai
Smith	Nashik
Jay	Solapur

Fig .10.4.1 The Employee relation

Employee_salary		
Emp_name	**Department**	**Salary**
Hari	Computer	10000
Om	IT	7000
Bill	Computer	8000
Jay	IT	5000

Fig. 10.4.2 The Employee_Salary relation

Example 1:

```
SQL > Select Employee.Emp_name, Employee_salary. Salary
from Employee inner join Employee_salary on
Employee. Emp_name = Employee_salary. Emp_name;
```

Fig. 10.4.3 shows the result of above query

Emp_name	Salary
Hari	10000
Om	7000
Jay	5000

Fig.10.4.3 The result of Employee inner join employee_salary operation with selected fields from employee and employee and employee_salary relation

Example 2:

```
Select  *
from Employee inner join Employee_salary on
Employee. Emp_name = Employee_salary. Emp_name;
```

The result of above query is shown in Fig. 10.4.4

Emp_name	City	Emp_name	Department	Salary
Hari	Pune	Hari	Computer	10000
Om	Mumbai	Om	IT	7000
Jay	Solapur	Jay	IT	5000

Fig. 10.4.4 The result of Employee inner join employee_salary operation with all fields from employee and employee_salary relation

As shown in Fig. 10.4.4 the result consists of the attributes of the left-hand-side relation followed by the attributes of the right-hand-side relation. Thus the Emp_name attribute appears twice in result first is form Employee relation and second is from Employee_salary relation.

10.4.2 Outer Join:

When tables are joined using inner join, rows which contain matching values in the join predicate are returned. Sometimes, you may want both matching and non_matching rows returned for the tables that are being joined. This kind of an operation is known as an **outer join.**

An outer join is an extended from of the inner join. In this, the rows in one table having no matching rows in the other table will also appear in the result table with nulls.

Types of outer join:

The outer join can be any one of the following:

- Left outer

- Right outer
- Full outer

Join types and conditions:

Join operations take two relations and return another relation as the result. Each of the variants of the join operation in SQL consists of a join type and join condition.

Join type: It defines how tuples in each relation that do not match with any tuple in the other relation, are treated. Following Fig. 10.4.5 shows various allowed join types.

Join types
Inner join
Left outer join
Right outer join
Full outer join

Fig. 10.4.5 Join types

Join conditions: The join condition defines which tuples in the two relations match and what attributes are present in the result of the join.

Following Fig. 10.4.6 shows allowed join conditions.

Join conditions
natural
on <predicate>
using (A1, A2, …..An)

Fig. 10.4.6 Join conditions

The use of join condition is mandatory for outer joins, but is optional for inner join (if it is omitted, a Cartesian product results). Syntactically, the keyword **natural** appears before the join type, whereas the **on** and **using** conditions appear at the end of the join expression.

The join condition **using** (A1, A2, ….., An) is similar to the natural join condition, except that the join attributes are the attributes A1, A2, ……, An, rather than all attributes that are common to both relations. The attributes A1, A2, ……, An must consist of only attributes that are common to both relations, and they appear only once in the result of the join.

Example: Employee **full outer join** Emp_salary using (Emp_name)

10.4.2.1 Left Outer Join:

The left outer join returns matching rows from the tables being joined, and also non-matching rows from the left table in the result and places null values in the attributes that come from the right table.

Example:

```
Select Employee. Emp_name, Salary
from Employee left outer Employee_salary
on Employee. Emp_name = Employee_salary. Emp_name;
```

The result of above query is shown in Fig. 10.4.7.

Emp_name	Salary
Hari	10000
Om	7000
Jay	5000
Smith	Null

Fig. 10.4.7 The result of Employee left outer join Employee_salary with selected fields from Employee_salary relations.

Left outer join operation is computed as follows:

First, compute the result of inner join as before. Then, for every tuple t in the left hand side relation Employee that does not match any tuple in the right-hand-side relation Employee_salary in the inner join, add a tuple r to the result of the join : The attributes of tuple r that are derived from the left-hand-side relation are filled with the from tuple t, remaining attributes of r are filled with null values as shown in Fig. 10.4.7

10.4.2.2 Right Outer Join:

The right outer join operation returns matching rows from the tables being joined, and also non-matching rows from the right table in the result and places null values in the attributes that comes from the left table.

Example:

```
Select Employee. Emp_name, City, Salary from Employee right outer join
Employee_salary on Employee. Emp_name = Employee_salary. Emp_name;
```

The result of preceding query is shown in Fig. 10.4.8.

Emp_name	City	Salary
Hari	Pune	10000
OM	Mumbai	7000
Bill	Null	5000
Jay	Solapur	8000

Fig. 10.4.8 The result of outer join operation with selected fields from Employee and Employee_salary relations

10.5 IMPORTANT QUESTIONS AND ANSWERS

Q.1 For given database, write SQL queries.

Employee (EID, Name, Street, City)

Works(BID, CID, Salary)

Manager (CID, Company_name, City)

i) Modify the database so that PRATHAM now lives in "USA

ii) Find all employees in the database who live in the same cities as the company for which they work.

iii) Give all employees of SHARAYU Steel' a 10% raise in salary.

Ans.:

i) update Employee
set city = 'USA'
where EID = 'PRATHAM';

ii) select Name
from Employee, Works, company where Employee. EID = Works.EID and
Works.CID = company. CID and Employee. City = comapy. City;

iii) update Works
set Salary = Salary * 1.10
where CID = (Select CID from company where company-name = 'SHARAYU
Steel');

Q.2 For the given employee database give an expression in SQL for the following:

Employee (empname,street, city)

Works (empname, company-name, salary)

Company (Company-name, city)

Managers (empname, manager-name)

i) Modify database so that 'swapnil' now lives in 'Navi Mumbai'.

ii) Give all employees of 'IBM' a 40% noise.

iii) List all the employees who lives in the same cities as their managers.

Ans.:

i) update Employee
set city = Navi Mumbai
where empname = Swapnil;

ii) update Employee
set salary = Salary * 1.40
where company-name = IBM;

iii) select E. empname
from Empolyee E, Employee T, Managers M
where E.empname = M. empname and
e.city = T. city and T. empname = M. manager – name;

Q.3 Consider the employee database where the primary keys are underlined. Give an Expression in SQL for the following gueries:

Employee (employee-name, street, city)

Works (employee-name, company-name, salary)

Company (company-name, city)

manages (employee-name, manager-name)

i) Find all employees in the database who earn more than each employee of small Bank Corporation.

ii) Find all employees in the database who do not work for first Bank Corporation.

iii) Find all employees who earn more that the average salary of all employees of their company.

iv) Find the names of all employees who work for First Bank Corporation.

Ans.:

i) select employee name
from works
where salary > all
(select salary
from works
where company name = 'small Bank Corporation');

ii) select employee name
from works
where company name <> 'First bank Corporation';

iii) create view Avg-salary (salary) as
select avg (Salary)
from works;
select employee-name
from works
where salary > Avg-salary. Salary ;

iv) select employee name
from works
where company name = 'First Bank Corporation';

Q.4 For the following given database, write SQL queries:-
Person (driver_id #, name, address)
Car (license, model, year)
accident (report_no, date, location)

owns (driver_id #, license)
participated (drive cid, car, report_number, damage_amount)

i) Find the total number of people who of people who owned cars that involved in an accident in 2007.

ii) Find the number of accidents in which the cars belonging to "Ajay". Were involved.

iii) Find the number of accidents that were reported in Mumbai region in the year 2004.

Ans.:

i) select count (distinct name)
from accident, participated, person
where accident. report number = participated.report number
and participated. Driver id = person.driver id
and date between '01-01-2007' and '31-12-2007';

ii) select count (distinct report_number)
from accident natural join participated natural join person
where name = 'Ajay';

iii) select count (distinct report_number)
from accident
where location = 'Mumbai' and date between '01-01-2004' and '31-12-2004';

Q.5 For the following given database, write SQL queries:-
person (driver_id#, name, address)
car (license, model, year)
accident (report_no, date, location)
owns (driver_id #, license)
participated (driver_id, car, report_number, damage_amount)

i) Find the total number of people who owned cars that were involved in accident in 1995.

ii) Find the number of accidents in which the cars belonging to Sunil K", were involved.

iii) Update the damage amount for car with licence number "Mum2022" in the accident with report number "AR2197" to rs5000.

Ans.:

i) select count (distinct name)
from accident, participated, person
where accident.report number = participated. Report number
and participated. Driver id = person. Driver id
and date between '01-01-1995' and '31-12-1995';

ii) select count (distinct report_number)
from accident natural join participated natural join person
where name = 'Sunil K';

iii) update participated
set damage_amount = 5000
where license = "MUM2022" and report_number = "AR2197";

Q.6 Given the following relational schema.
Division (div #, div-name, director)
Department (dept #, dept-name, location, div #)
Employee (emp #, emp name, salary, address, dept #)
State the following queries in SQL.

i) Get tmployee name, dept-name and division name for all employees whose salary is above 20,000/-

ii) List the name of all employees who work in "Marketing" division.

iii) List the dept-names and employee name in that dept, for all department whose location is "Mumbai".

Ans.:

i) Select emp-name, dept-name, div-name From Employee, Department, Division
where Employee. Dept # = Department. Dept # and Department. Div # = Division
. div # and salary > 20,000;

ii) Select emp-name FROM Employee, Department, Division
where Employee. Dept # and Department. Dept # and Department div # = Division .
div # and div-name = 'Marketing' ;

iii) Select dept-name, emp-name
from Employee, Department
where Employee. Dept # = Department. Dept # and location = 'Mumbai' ;

Q. 7 Consider the relations
EMP (ENO,ENAME, AGE, BASIC_SALARY)
WORK_IN(ENO,DNO)
DEPT (DNO,DNAME,CITY)
Express the following queries in SQL

i) Find names of employees who work in a department in Pune.

ii) Get the dept. Number in which more than one employee is working.

iii) Find name of employee who earns highest salary in 'HR' department.

Ans.:

i)
```
select ENAME
from EMP, WORK_IN, DEPT
where EMP.ENO = WORK_IN.ENO and WORK_IN.DNO =
DEPT. DNO and CITY =
'Pune' ;
```

ii)
```
select DNO
from WORK_IN
group by DNO
HAVING COUNT (*) >1;
```

iii)
```
select ENAME
from EMP e
where BASIC_SALARY > = (select max(BASIC_SALARY)
from DEPT, WORK_IN
where DNAME = 'HR' and e. ENO =
WORK_IN. ENO and
WORK_IN. DNO = DEPT. DNO)
```

10.6 SUMMARY

Subqueries are written in the WHERE clause of the another SQL statement to obtain values based on the unknown conditional value. Use of a subquery is equivalent to performing two or more sequential queries.

The result of inner query is fed to the outer query to display the values.

A join condition is used when data from more than one table in the database is required to be displayed. Rows in one table can be joined to rows of another table according to common values existing in the corresponding columns, those usually primary key and foreign key columns.

A view takes takes the output of a query and treats it as a table ; therefore a view can be thought of as stored query or a virtual table .It provides an additional level of table security by restricting access to a predetermined set of rows and / or columns of a table. It hides data complexity and it simplifies commands for the user because they allow to select information from multiple tables without actually knowing how to perform a join.

A view object that gives the user a logical view of data from an underlying table or tables. The user can perform Insert, Update, and the Delete operations on the view. The views on which data manipulation can be done are called Updatable Views.

A subquery is a select from where expression that is nested within another query. Mostly subqueries are used to perform tests for set membership to make set comparisons and determine set cardinality.

SQL uses in and not in constructs for set membership tests. Nested subqueries are used to compare sets. SQL uses various comparison operators such as <, < = , > , > = , =, <>, any, all, and some, etc to compare sets.

Join is a query in which data is retrieved from two or more table. a join matches data from two or more tables, based on the values of one or more columns in each table.

Inner Join returns the matching rows from the tables that are being joined. You may want both matching and non_matching rows returned for the tables that are being joined. This kind of an operation is known as an **outer join.**

10.7 UNIT END QUESTIONS

1) What are views? Why they can't be used for updates?
2) Explain following operations w.r.t views : create, drop, update.
3) Explain join operation in SQL.
4) Explain different types of outer join operations with example.
5) Explain database modification using SQL.
6) Explain need for following :
   i) View
   ii) Null values
10) Compare:
   a. Outer join and full outer join
   b. Left outer join and right outer join

Text Books:

1. Korth, Slberchatz, Sudarshan, Database System Concepts, 6th Edition, McGraw Hill
2. Elmasri and Navathe, Fundamentals of Database Systems, 6th Edition, Pearson education
3. Raghu Ramkrishnan and Johannes Gehrke, Database Management Systems, TMH

REFERENCES

1. Peter Rob and Carlos Coronel, ― Database Systems Design, Implementation and Management‖, Thomson Learning, 9thEdition.
2. G. K. Gupta : "Database Management Systems", McGraw – Hill

*****

Unit IV

11

TRANSACTION MANAGEMENT & CONCURRENCY CONTROL

Unit Structure

11.1 ACID PROPERTIES

ACID Properties:

A transaction is a very small unit of a program and it may contain several lowlevel tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability – commonly known as ACID properties – in order to ensure accuracy, completeness, and data integrity.

- **Atomicity:** This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency:** The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability:** The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction

updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

- **Isolation:** In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

To gain a better understanding of ACID properties and the need for them, consider a simplified banking system consisting of several accounts and a set of transactions that access and update those accounts. For the time being, we assume that the database permanently resides on disk, but that some portion of it is temporarily residing in main memory.

11.2 SERIALIZABILITY & CONCURRENCY CONTROL

The database system must control concurrent execution of transactions, to ensure that the database state remains consistent. Before we examine how the database system can carry out this task, we must first understand which schedules will ensure consistency, and which schedules will not.

T_1	T_2
read(A)	
A := A − 50	
	read(A)
	temp := A * 0.1
	A := A − temp
	write(A)
	read(B)
write(A)	
read(B)	
B := B + 50	
write(B)	
	B := B + temp
	write(B)

Fig. 1 : Schedule 4-n a concurrent schedule.

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. For this reason, we shall not interpret the type of operations that a transaction can perform on a data item. Instead, we consider only two operations: read and write. We thus assume that, between a read(Q) instruction and a write (Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction.

Thus, the only significant operations of a transaction, from a scheduling point of view, are instructions in schedules, as we do in schedule 3 in Figure 1.

In this section, we discuss different forms of schedule equivalence; they lead to the notions of conflict serializability and view serializability.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Fig. 2 : Schedule 3—showing only the read and write instructions

11.3 LOCK BASED CONCURRENCY CONTROL

A DBMS must be able to ensure that only serializable, recoverable schedules are allowed, and that no actions of committed transactions are lost while undoing aborted transactions. A DBMS typically uses a locking protocol to achieve this. A locking protocolis a set of rules to be followed by each transaction (and enforced by the DBMS), in order to ensure that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order.

Strict Two-Phase Locking (Strict 2PL):

The most widely used locking protocol, called Strict Two-Phase Locking, or Strict 2PL, has two rules. The first rule is

(i) If a transaction T wants to read (respectively, modify) an object, it first requests a shared (respectively, exclusive) lock on the object.

Of course, a transaction that has an exclusive lock can also read the object; an additional shared lock is not required. A transaction that requests a lock is suspended until the DBMS is able to grant it the requested lock. The DBMS keeps track of the locks it has granted and ensures that if a transaction holds an exclusive lock on an object, no other transaction holds a shared or exclusive lock on the same object.

The second rule in Strict 2PL is:

(ii) All locks held by a transaction are released when the transaction is completed.

Requests to acquire and release locks can be automatically inserted into transactions by the DBMS; users need not worry about these details.

In effect the locking protocol allows only 'safe' interleavings of transactions. If two transactions access completely independent parts of the database, they will be able to concurrently obtain the locks that they need and proceed merrily on their ways. On the other hand, if two transactions access the same object, and one of them wants to modify it, their actions are effectively ordered serially—all actions of one of these transactions (the one that gets the lock on the common object first) are completed before (this lock is released and) the other transaction can proceed.

We denote the action of a transaction T requesting a shared (respectively, exclusive) lock on object O as S_T (O) (respectively, X(O)), and omit the subscript denoting the transaction when it is clear from the context. As an example, consider the schedule shown in figure 2. This interleaving could result in a state that cannot result from any serial execution of the three transactions. For instance, T1 could change A from 10 to 20, then T2 (which reads the value 20 for A) could change B from 100 to 200, and then T1 would read the value 200 for B. If run serially, either T1 or T2 would execute first, and read the values 10 for A and 100 for B: Clearly, the interleaved execution is not equivalent to either serial execution.

If the Strict 2PL protocol is used, the above interleaving is disallowed. Let us see why. Assuming that the transactions proceed at the same relative speed as before, T1 would obtain an exclusive lock on A first and then read and write A (Figure However, this request cannot be granted until

T1	T2
X(A)	
R(A)	
W(A)	

Fig. 3 : Schedule Illustrating Strict 2PL

T1 releases its exclusive lock on A, and the DBMS therefore suspends T2. T1now proceeds to obtain an exclusive lock on B, reads and writes B, then finally commits, at which time its locks are released. T2's lock request is now granted, and it proceeds. In this example the locking protocol results in a serial execution of the two transactions. In general, however, the actions of different transactions could be interleaved. As an example, consider the interleaving of two transactions shown in Figure 18.6, which is permitted by the Strict 2PL protocol.

Deadlock:

In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.

For example, assume a set of transactions $\{T_0, T_1, T_2, ...,T_n\}$. T_0 needs a resource X to complete its task. Resource X is held by T_1, and T_1 is waiting for a resource Y, which is held by T_2. T_2 is waiting for resource Z, which is held by T_0. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

Deadlock Prevention:

For large database, deadlock prevention method is suitable. A deadlock can be prevented if the resources are allocated in such a way that deadlock never occur. The DBMS analyzes the operations whether they can create deadlock situation or not, If they do, that transaction is never allowed to be executed.

Deadlock prevention mechanism proposes two schemes :

Wait-Die Scheme:

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur –

- If $TS(T_i) < TS(T_j)$ – that is T_i, which is requesting a conflicting lock, is older than T_j – then T_i is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ – that is T_i is younger than T_j – then T_i dies. T_i is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

Wound-Wait Scheme:

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur –

- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back – that is T_i wounds T_j. T_j is restarted later with a random delay but with the same timestamp.

- If $TS(T_i) > TS(T_j)$, then T_i is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.
In both the cases, the transaction that enters the system at a later stage is aborted.

Deadlock Detection:

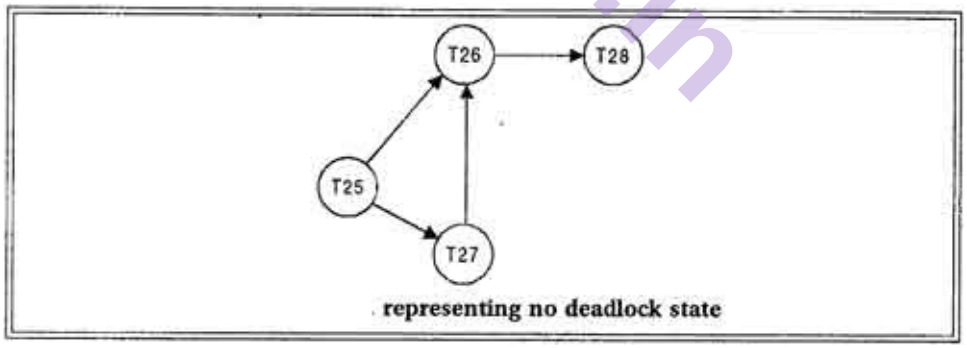
A simple way to detect a state of deadlock is with the help of wait-for graph. This graph is constructed and maintained by the system. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction Ti is waiting to lock an item X that is currently locked by a transaction Tj, a directed edge (Ti->Tj).is created in the wait-for graph. When Tj releases the lock(s) on the items that Ti was waiting for, the directed edge is dropped from the wait-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle. Then each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

To illustrate these concepts, consider the following wait-for graph in figure. Here:

Transaction T25 is waiting for transactions T26 and T27.

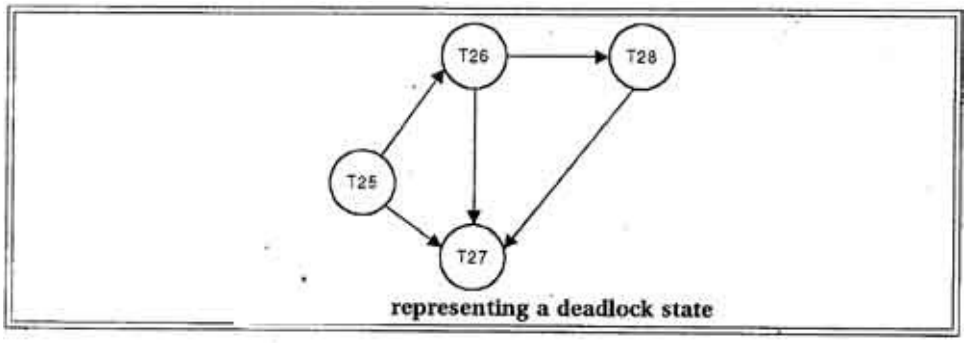
Transactions T27 is waiting for transaction T26.

Transaction T26 is waiting for transaction T28.



representing no deadlock state

This wait-for graph has no cycle, so there is no deadlock state.
Suppose now that transaction T28 is requesting an item held by T27. Then the edge T28 ----------->T27 is added to the wait -for graph, resulting in a new system state as shown in figure.



representing a deadlock state

This time the graph contains the cycle.
T26------>T28------->T27---------->T26

It means that transactions T26, T27 and T28 are all deadlocked.

Invoking the deadlock detection algorithm

The invoking of deadlock detection algorithm depends on two factors:

- How often does a deadlock occur?
- How many transactions will be affected by the deadlock?

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently than usual. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

Deadlock Recovery:

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Choosing which transaction to abort is known as Victim Selection.

Choice of Deadlock victim:

In below wait-for graph transactions T26, T28 and T27 are deadlocked. In order to remove deadlock one of the transaction out of these three transactions must be roll backed.

We should roll back those transactions that will incur the minimum cost. When a deadlock is detected, the choice of which transaction to abort can be made using following criteria:

- The transaction which have the fewest locks
- The transaction that has done the least work
- The transaction that is farthest from completion

Rollback:

Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back. The simplest solution is a total rollback; Abort the transaction and then restart it. However it is more effective to roll back the transaction only as far as necessary to break the deadlock. But this method requires the system to maintain additional information about the state of all the running system.

Problem of Starvation:

In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result this transaction never completes can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

11.4 TIME STAMPING METHODS

Timestamp-Based Concurrency Control:

In lock-based concurrency control, conflicting actions of different transactions are ordered by the order in which locks are obtained, and the lock protocol extends this ordering on actions to transactions, thereby ensuring serializability. In optimistic concurrency control, a timestamp ordering is imposed on transactions, and validation checks that all conflicting actions occurred in the same order.

Timestamps can also be used in another way: each transaction can be assigned a time stamp at startup, and we can ensure, at execution time, that if action aj of transaction Tj conflicts with action aj of transaction Tj, aj occurs before aj if TS(Ti)<TS(Tj). If an action violates this ordering, the transaction is aborted and restarted.

To implement this concurrency control scheme, every database object O, is given a read timestamp RTS(O) and a write timestampWTS(O). If transaction T wants to read object O, and TS (T) <WTS(O), the order of this read with respect to the most recent write on O would violate the timestamp order between this transaction and the writer. Therefore, T is aborted and restarted with a new, larger timestamp. If TS(T) >WTS(O), T reads O, and RTS(O) is set to the larger of RTS(O)and TS(T). (Note that there is a physical change—the change to RTS(O)—to be written to disk and to be recorded in the log for recovery purposes, even on reads. This write operation is a significant overhead.)

Observe that if T is restarted with the same timestamp, it is guaranteed to be aborted again, due to the same conflict. Contrast this behavior with the use of timestamps in 2PL for deadlock prevention: there, transactions were restarted with the same timestamp as before in order to avoid repeated restarts. This shows that the two uses of timestamps are quite different and should not be confused.

Next, let us consider what happens when transaction T wants to write object O:

i) If TS(T)<RTS(O), the write action conflicts with the most recent read action of O, and T is therefore aborted and restarted.

ii) If TS(T) <WTS(O), a naive approach would be to abort T because its write action conflicts with the most recent write of O and is out of timestamp order. It turns out that we can safely ignore such writes and continue. Ignoring outdated writes is called the Thomas Write Rule.

iii) Otherwise, T writes O and WTS (O) is set to TS(T).

11.5 OPTIMISTIC METHODS

Optimistic concurrency control (**OCC**) is a concurrency control method applied to transactional systems such as relational database management systems and software transactional memory. OCC assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.[1] Optimistic concurrency control was first proposed by H.T. Kung.[2]

OCC is generally used in environments with low data contention. When conflicts are rare, transactions can complete without the expense of managing locks and without having transactions wait for other transactions' locks to clear, leading to higher throughput than other concurrency control methods. However, if contention for data resources is frequent, the cost of repeatedly restarting transactions hurts performance significantly; it is commonly thought[*who?*] that other concurrency control methods have better performance under these conditions.[*citation needed*] However, locking-based ("pessimistic") methods also can deliver poor performance because locking can drastically limit effective concurrency even when deadlocks are avoided.

More specifically, OCC transactions involve these phases:

- **Begin**: Record a timestamp marking the transaction's beginning.
- **Modify**: Read database values, and tentatively write changes.

- **Validate**: Check whether other transactions have modified data that this transaction has used (read or written). This includes transactions that completed after this transaction's start time, and optionally, transactions that are still active at validation time.
- **Commit/Rollback**: If there is no conflict, make all changes take effect. If there is a conflict, resolve it, typically by aborting the transaction, although other resolution schemes are possible. Care must be taken to avoid a TOCTTOU bug, particularly if this phase and the previous one are not performed as a single atomic operation

11.6 DATABASE RECOVERYMANAGEMENT

A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage. In any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions. An integral part of a database system is a recovery scheme that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide high availability; that is, it must minimize the time for which the database is not usable after a crash.

Failure Classification:

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. The simplest type of failure is one that does not result in the loss of information in the system. The failures that are more difficult to deal with are those that result in loss of information. In this chapter, we shall consider only the following types of failure :

- **Transaction failure.** There are two types of errors that may cause a transaction to fail :
  - **Logical error.** The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - **System error.** The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.
- **System crash:** There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.

  The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage

contents, is known as the fail-stop assumption. Well-designed systems have numerous internal checks, at the hardware and the software level, that bring the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.

- **Disk failure:** A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as tapes, are used to recover from the failure.

To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data. Next, we must consider how these failure modes affect the contents of the database. We can then propose algorithms to ensure database consistency and transaction atomicity despite failures. These algorithms, known as recovery algorithms, have two parts:

i) Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.

ii) Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

STORAGE STRUCTURE:

The various data items in the database may be stored and accessed in a number of different storage media. To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of these storage media and their access methods.

Storage Types:

We saw that storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as volatile storage or nonvolatile storage. We review these terms, and introduce another class of storage, called stable storage.

- **Volatile storage:** Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.

- **Nonvolatile storage:** Information residing in nonvolatile storage survives system crashes. Examples of such storage are disk and magnetic tapes. Disks are used for online storage, whereas tapes are used for archival storage. Both, however, are subject to failure (for example, head crash), which may result in loss of information. At the current state of technology, nonvolatile storage is slower than volatile

storage by several orders of magnitude. This is because disk and tape devices are electromechanical, rather than based entirely on chips, as is volatile storage. In database systems, disks are used for most nonvolatile storage. Other nonvolatile media are normally used only for backup data. Flash storage, though nonvolatile, has insufficient capacity for most database systems.

- **Stable storage:** Information residing in stable storage is never lost (never should be taken with a grain of salt, since theoretically never cannot be guaranteed — for example, it is possible, although extremely unlikely, that a black hole may envelop the earth and permanently destroy all data!). Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely.

The distinctions among the various storage types are often less clear in practice than in our presentation. Certain systems provide battery backup, so that some main memory can survive system crashes and power failures. Alternative forms of nonvolatile storage, such as optical media, provide an even higher degree of reliability than do disks.

Stable-Storage Implementation:

To implement stable storage, we need to replicate the needed information in several nonvolatile storage media (usually disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

Recall that RAID systems guarantee that the failure of a single disk (even during data transfer) will not result in loss of data. The simplest and fastest form of RAID is the mirrored disk, which keeps two copies of each block, on separate disks. Other forms of RAID offer lower costs, but at the expense of lower performance.

RAID systems, however, cannot guard against data loss due to disasters such as fires or flooding. Many systems store archival backups of tapes off-site to guard against such disasters. However, since tapes cannot be carried off-site continually, updates since the most recent time that tapes were carried off-site could be lost in such a disaster. More secure systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system. Since the blocks are output to a remote system as and when they are output to local storage, once an output operation is complete, the output is not lost, even in the event of a disaster such as a fire or flood.

In the remainder of this section, we discuss how storage media can be protected from failure during data transfer. Block transfer between memory and disk storage can result in

- **Successful completion.** The transferred information arrived safely at its destination.
- **Partial failure.** A failure occurred in the midst of transfer, and the destination block has incorrect information.
- **Total failure.** The failure occurred sufficiently early during the transfer that the destination block remains intact.

We can extend this procedure easily to allow the use of an arbitrarily large number of copies of each block of stable storage. Although a large number of copies reduces the probability of a failure to even lower than two copies do, it is usually reasonable to simulate stable storage with only two copies.

Data Access:

The database system resides permanently on nonvolatile storage (usually disks), and is partitioned into fixed-length storage units called blocks. Blocks are the units of data transfer to and from disk, and may contain several data items. We shall assume that no data item spans two or more blocks. This assumption is realistic for most data-processing applications, such as our banking example.

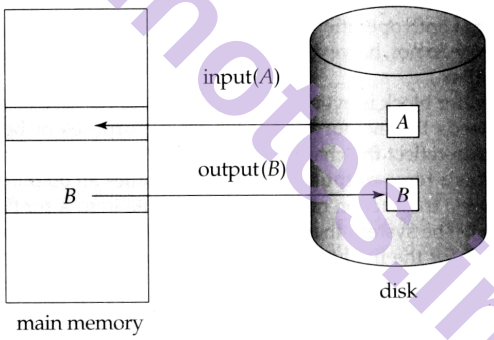


Fig. 4 : Block storage operations.

Transactions input information from the disk to main memory, and then output the information back onto the disk. The input and output operations are done in block units. The blocks residing on the disk are referred to as physical blocks; the blocks residing temporarily in main memory are referred to as buffer blocks. The area of memory where blocks reside temporarily is called the disk buffer.

Block movements between disk and main memory are initiated through the following two operations:

1) Input (B) transfers the physical block B to main memory.
2) Output (B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.

Each transaction $T_i$has a private work area in which copies of all the data items accessed and updated by $T_i$are kept. The system creates this work area when the transaction is initiated; the system removes it when the transaction either commits or aborts. Each data item X kept in the work area of transaction T_i is denoted by x_i. Transaction $T_i$interacts with the database system by transferring data to and from its work area to the system buffer. We transfer data by these two operations:

i) read(X) assigns the value of data item X to the local variable x_i. It executes this operation as follows:
   (a) If block $B_X$on which X resides is not in main memory, it issues input(B_X).
   (b) It assigns to x_i, the value of X from the buffer block.

ii) write(X) assigns the value of local variable x_i to data item X in the buffer block. It executes this operation as follows:
   (a) If block $B_X$on which X resides is not in main memory, it issues input (B_X).
   (b) It assigns the value of $x_i$to X in buffer B_X.

Note that both operations may require the transfer of a block from disk to main memory. They do not, however, specifically require the transfer of a block from main memory to disk.

A buffer block is eventually written out to the disk either because the buffer manager needs the memory space for other purposes or because the database system wishes to reflect the change to B on the disk. We shall say that the database system performs a force-output of buffer B if it issues an output (B).

When a transaction needs to access a data item X for the first time, it must execute read (X). The system then performs all updates to X on x_i. After the transaction accesses X for the final time, it must execute write(X) to reflect the change to X in the database itself.

The output (B_X) operation for the buffer block $B_X$on which X resides does not need to take effect immediately after write (X) is executed, since the block $B_X$may contain other data items that are still being accessed. Thus, the actual output may take place later. Notice that, if the system crashes after the write(X) operation was executed but before output(B_X) was executed, the new value of X is never written to disk and, thus, is lost.

Recovery and Atomicity:

Consider again our simplified banking system and transaction $T_i$that transfers $50 from account A to account B, with initial values of A and B being $1000 and $2000, respectively. Suppose that a system crash has occurred during the execution of T_i,after output(B_A) has taken place,

but before output(B_B) was executed, where $B_A$and denote the buffer blocks on which A and B reside. Since the memory contents were lost, we do not know the fate of the transaction; thus, we could invoke one of two possible recovery procedures:

- Re-execute T_i. This procedure will result in the value of Abecoming $900, rather than $950. Thus, the system enters an inconsistent state.
- Do not re-execute T_i. The current system state has values of $950 and $2000 for A and B, respectively. Thus, the system enters an inconsistent state.

In either case, the database is left in an inconsistent state, and thus this simple recovery scheme does not work. The reason for this difficulty is that we have modified the database without having assurance that the transaction will indeed commit. Our goal is to perform either all or no database modifications made by T_i. However, if T_i performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

To achieve our goal of atomicity, we must first output information describing the modifications to stable storage, without modifying the database itself. As we shall see, this procedure will allow us to output all the modifications made by a committed transaction, despite failures.

11.7 SUMMARY

This chapter defines a transaction and describes how the database processes transactions. It also gives an idea about Serializability and concurrency control, Lock based concurrency control (2PL, Deadlocks),Time stamping methods and Optimistic methods.

11.8 REFERENCES

1. Database System and Concepts A Silberschatz, H Korth, S Sudarshan McGraw-Hill Fifth Edition
2. “Fundamentals of Database Systems” by Elmsari, Navathe, 5th Edition, Pearson Education (2008).
3. “Database Management Systems” by Raghu Ramakrishnan, Johannes Gehrke, McGraw Hill Publication.
4. “Database Systems, Concepts, Design and Applications” by S.K.Singh, Pearson Education.

11.9 UNIT END QUESTIONS

Multiple Choice Questions:

1. Identify the characteristics of transactions
   a) Atomicity
   b) Durability
   c) Isolation
   d) All of the mentioned

2. Which of the following has “all-or-none” property ?
   a) Atomicity
   b) Durability
   c) Isolation
   d) All of the mentioned

3. The database system must take special actions to ensure that transactions operate properly without interference from concurrently executing database statements. This property is referred to as
   a) Atomicity
   b) Durability
   c) Isolation
   d) All of the mentioned

4. Deadlocks are possible only when one of the transactions wants to obtain a(n) ____ lock on a data item.
   a) binary
   b) exclusive
   c) shared
   d) complete

5. The ____ statement is used to end a successful transaction.
   a) COMMIT
   b) DONE
   c) END
   d) QUIT

6. If a transaction acquires a shared lock, then it can perform ..... operation.
   a) read
   b) write
   c) read and write
   d) update

7. A system is in a _____ state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.
   a) Idle
   b) Waiting
   c) Deadlock
   d) Ready

8. The deadlock state can be changed back to stable state by using _____ statement.
   a) Commit
   b) Rollback
   c) Savepoint
   d) Deadlock

9. What are the ways of dealing with deadlock ?
   a) Deadlock prevention
   b) Deadlock recovery
   c) Deadlock detection
   d) All of the mentioned

10. The situation where the lock waits only for a specified amount of time for another lock to be released is
   a) Lock timeout
   b) Wait-wound
   c) Timeout
   d) Wait

Answer the following

1. Explain the concept of transaction
2. Describe ACID properties of transaction
3. Explain difference between the terms serial serial schedule and serializable schedule with suitable examples.
4. Explain View and conflict serializability with suitable example
5. Explain the need of concurrency control in transaction management
6. Write a short note on Two phase locking protocol
7. Explain Timestamp based protocol
8. Show that two phase locking protocol ensures conflict serializability
9. What is Deadlock. Explain Deadlock detection
10. Explain Deadlock Recovery

*****

Unit V

12

BEGINNING WITH PL / SQL,

Unit Structure

12.1 BEGINNING WITH PL / SQL

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are certain notable facts about PL/SQL:

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line **SQL*Plus interface**.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- Apart from Oracle, PL/SQL is available in **TimesTen in-memory database** and **IBM DB2**.

Features of PL/SQL:
PL/SQL has the following features:

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

Advantages of PL/SQL:

PL/SQL has the following advantages:

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.

- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.

- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.

- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.

- Applications written in PL/SQL are fully portable.

- PL/SQL provides high security level.

- PL/SQL provides access to predefined SQL packages.

- PL/SQL provides support for Object-Oriented Programming.
  \
- PL/SQL provides support for developing Web Applications and Server Pages.

PL SQL blocks can be divide into two broad categories:

1. Anonymous Block:

2. Named Block:

Structure of PL SQL block:

1. Anonymous Block:

PL/SQL program units organize the code into blocks. A block without a name is known as an anonymous block. The anonymous block is the simplest unit in PL/SQL. It is called anonymous block because it is not saved in the Oracle database.

An anonymous block is an only one-time use and useful in certain situations such as creating test units. The following illustrates anonymous block syntax:

```
DECLARE]
  Declaration statements;
BEGIN
  Execution statements;
     [EXCEPTION]
      Exception handling statements;
END;
/
```

The anonymous block has three basic sections that are the declaration, execution, and exception handling. Only the execution section is mandatory and the others are optional.

- The declaration section allows you to define data types, structures, and variables. You often declare variables in the declaration section by giving them names, data types, and initial values.

- The execution section is required in a block structure and it must have at least one statement. The execution section is the place where you put the execution code or business logic code. You can use both procedural and SQL statements inside the execution section.

- The exception handling section is starting with the EXCEPTION keyword. The exception section is the place that you put the code to handle exceptions. You can either catch or handle exceptions in the exception section.

Example: To create Pl/SQL block which inserts 2 records in student table

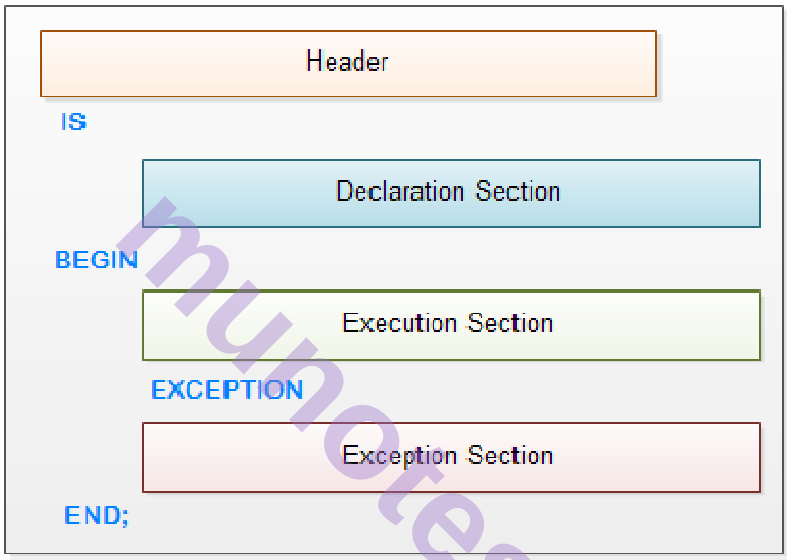
```
Begin
Insert into student
Values('A101','Om',50);
Insert into student
```

Values('A102','Ram',55);
End;

2. Named Block:

Named Block is a type of block which starts with the header section which specifies the name and the type of the block .There are two types of blocks namely :-
a) Procedures
b) Functions

Let's examine the PL/SQL block structure in greater detail.



Header:

Relevant for named blocks only, the header determines the way that the named block or program must be called. The header includes the name, parameter list and return clause(only for function)

Generate output from a PL/SQL block:
DBMS_OUTPUT is a built-in package that enables you to display output, debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers

Example: BEGIN
dbms_output.put_line('Hello'); dbms_output.put(' World');
dbms_output.put_line('Welcome');
END;
Output: Hello
World
Welcome

12.2 IDENTIFIERS AND KEYWORDS

The PL/SQL Identifiers PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, identifiers are not case-sensitive. So you can use integer or INTEGER to represent a numeric value. You cannot use a reserved keyword as an identifier.

The words listed in this appendix are reserved by **PL/SQL**. You should not use them to name program objects such as constants, variables, cursors, schema objects such as columns, tables, or indexes. These words reserved by **PL/SQL** are classified as **keywords** or reserved words.

12.3 PL/SQL - OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators "

- Arithmetic operators
- Relational operators
- Comparison operators
- Logical operators
- String operators

Here, we will understand the arithmetic, relational, comparison and logical operators one by one.

Arithmetic Operators:

Following table shows all the arithmetic operators supported by PL/SQL. Let us assume variable A holds 10 and variable B holds 5, then "

Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

Relational Operators:

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 20, the

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!=<>~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true

Comparison Operators:

Comparison operators are used for comparing one expression to another. The result is always either TRUE, FALSE or NULL.

Operator	Description	Example
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests whether a value lies in a	If x = 10 then, x between 5 and 20 returns true, x between

	specified range. x BETWEEN a AND b means that x >= a and x <= b.	5 and 10 returns true, but x between 11 and 20 returns false.
IN .	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If x = 'm' then, x in ('a', 'b', 'c') returns Boolean false but x in ('m', 'n', 'o') returns Boolean true. IS NULL The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL. If x = 'm', then 'x is null' returns Boolean false

Logical Operators:

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produce Boolean results. Let us assume variable A holds true and variable B holds false, then:

Operator	Description	Examples
and	Called the logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.
or	Called the logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.
not	Called the logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.	not (A and B) is true.

PL/SQL Operator Precedence:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.
The precedence of operators goes as follows: =, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN.

Operator	Operation
**	Exponentiation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
comparison	
NOT	logical negation
AND	Conjunction
OR	Inclusion

12.4 EXPRESSION

An expression is an arbitrarily complex combination of operands (variables, constants, literals, operators, function calls, and placeholders) and operators. The simplest expression is a **single** variable.

Expressions are constructed using operands and operators.

An **operand** is a variable, constant, literal, or function call that contributes a value to an expression. An example of a simple arithmetic expression follows:

-X / 2 + 3

Unary operators such as the negation operator (-) operate on one operand; binary operators such as the division operator (/) operate on two operands. PL/SQL has no ternary operators.

The simplest expressions consist of a single variable, which yields a value directly. PL/SQL evaluates an expression by combining the values of the operands in ways specified by the operators. An expression always returns a single value. PL/SQL determines the datatype of this value by examining the expression and the context in which it appears.

Operator Precedence:

The operations within an expression are done in a particular order depending on their *precedence* (priority). Table 2-1 shows the default order of operations from first to last (top to bottom).

Table: Order of Operations

Operator	Operation
**	exponentiation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
comparison	
NOT	logical negation
AND	conjunction
OR	inclusion

12.5 ORACLE / PLSQL: SEQUENCES

This Oracle tutorial explains how to **create and drop sequences** in Oracle with syntax and examples.

Description:

In Oracle, you can **create an autonumber field by using sequences**. A sequence is an object in Oracle that is used to generate a number sequence. This can be useful when you need to create a unique number to act as a primary key.

Create Sequence:

You may wish to create a sequence in Oracle to handle an auto number field.

Syntax
The syntax to create a sequence in Oracle is:

```
CREATE SEQUENCE sequence_name
INVALUE value
   MAXVALUE value
   START WITH value
   INCREMENT BY value
   CACHE value;
```

sequence_name
The name of the sequence that you wish to create.

Example
Let's look at an example of how to create a sequence in Oracle.

For example:

```
CREATE SEQUENCE supplier_seq
MINVALUE 1
MAXVALUE 9999999999999999999999999999
```

```
START WITH 1
INCREMENT BY 1
CACHE 20;
```

This would create a sequence object called *supplier_seq*. The first sequence number that it would use is 1 and each subsequent number would increment by 1 (ie: 2,3,4,...}. It will cache up to 20 values for performance.

If you omit the MAXVALUE option, your sequence will automatically default to:

MAXVALUE 999999999999999999999999999

So you can simplify your CREATE SEQUENCE command as follows:

CREATE SEQUENCE supplier_seq

MINVALUE 1
START WITH 1
INCREMENT BY 1
CACHE 20;

Now that you've created a sequence object to simulate an autonumber field, we'll cover how to retrieve a value from this sequence object. To retrieve the next value in the sequence order, you need to use *nextval*.

For example:

supplier_seq.NEXTVAL;

This would retrieve the next value from *supplier_seq*.
The *nextval* statement needs to be used in a SQL statement.

For example:

INSERT INTO suppliers

(supplier_id, supplier_name)

VALUES

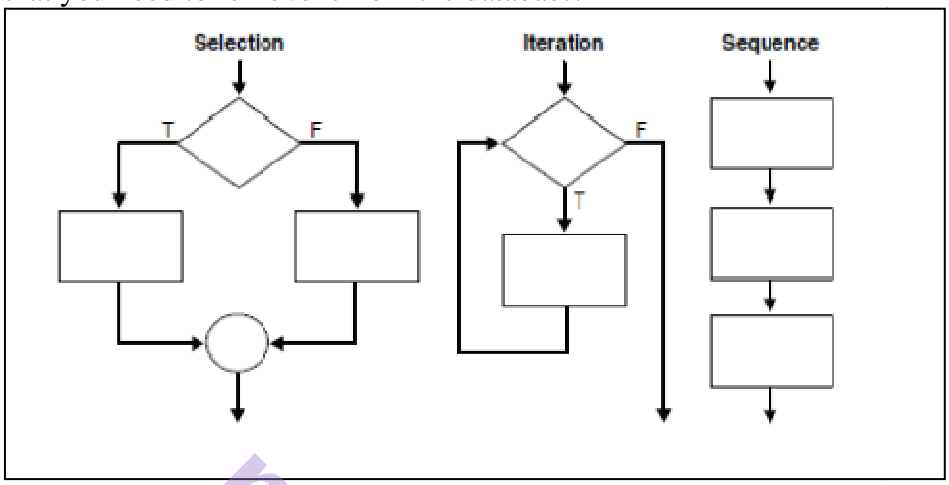
(supplier_seq.NEXTVAL, 'Kraft Foods');

This insert statement would insert a new record into the *suppliers* table. The *supplier_id* field would be assigned the next number from the

supplier_seq sequence. The *supplier_name* field would be set to Kraft Foods.

Drop Sequence:

Once you have created your sequence in Oracle, you might find that you need to remove it from the database.



Syntax

The syntax to a drop a sequence in Oracle is:

DROP SEQUENCE *sequence_name*;

sequence_name
The name of the sequence that you wish to drop.

Example

Let's look at an example of how to drop a sequence in Oracle.

For example:

DROP SEQUENCE supplier_seq;

This example would drop the sequence called *supplier_seq*.

PL/SQL Control Structures:

Procedural computer programs use the basic control structures

- The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a BOOLEAN value (TRUE or FALSE).
- The iteration structure executes a sequence of statements repeatedly as long as a condition holds true.

- The sequence structure simply executes a sequence of statements in the order in which they occur.

12.6 CONTROL STRUCTURES

Testing Conditions: IF and CASE Statements

The IF statement executes a sequence of statements depending on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF.

The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions. It makes sense to use CASE when there are three or more alternatives to choose from.

- **Using the IF-THEN Statement:**

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF)

The sequence of statements is executed only if the condition is TRUE. If the condition is FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement.

Example: Using a Simple IF-THEN Statement

```
DECLARE
sales NUMBER(8,2) := 1010
quota NUMBER(8,2) := 10000;
bonus NUMBER(6,2);
emp_id NUMBER(6) := 120;
BEGIN
IF sales > (quota + 200) THEN
bonus := (sales - quota)/4;
UPDATE employees SET salary = salary + bonus WHERE employee_id
= emp_id;
END IF;
END;
/
```

- **Using CASE Statements:**

Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions.

selector is an expression whose value is used to select one of several alternatives.

Example: Using the CASE-WHEN Statement

```
DECLARE
grade CHAR(1);
BEGIN
grade := 'B';
CASE grade
WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
END CASE;
END;
```

Controlling Loop Iterations: LOOP and EXIT Statements:

LOOP statements execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

- **Using the LOOP Statement:**

The simplest form of LOOP statement is the basic loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. You use an EXIT statement to stop looping and prevent an infinite loop. You can place one or more EXIT statements anywhere inside a loop, but not outside a loop. There are two forms of EXIT statements: EXIT and
EXIT-WHEN.

- **Using the EXIT Statement:**

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement.

- **Using the EXIT-WHEN Statement:**

The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop.

- **Labeling a PL/SQL Loop:**

Like PL/SQL blocks, loops can be labeled. The optional label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. When you nest labeled loops, use ending label names to improve readability.

- **Using the WHILE-LOOP Statement:**

The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

WHILE condition LOOP

sequence_of_statements

END LOOP;

Using the FOR-LOOP Statement:

Simple FOR loops iterate over a specified range of integers. The number of iterations is known before the loop is entered. A double dot (..) serves as the range operator. The range is evaluated when the FOR loop is first entered and is never re-evaluated. If the lower bound equals the higher bound, the loop body is executed once.

Example: Using a Simple FOR..LOOP Statement:

```
DECLARE

p NUMBER := 0;

BEGIN

FOR k IN 1..500 LOOP — calculate pi with 500 terms
p := p + ( ( (-1) ** (k + 1) ) / ((2 * k) - 1) );

END LOOP;
```

```
p := 4 * p;

DBMS_OUTPUT.PUT_LINE( 'pi is approximately : ' || p ); — print
result
END;
```**Sequential Control: GOTO and NULL Statements**

The GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.

Overuse of GOTO statements can result in code that is hard to understand and maintain. Use GOTO statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a GOTO statement.

- **Using the GOTO Statement:**

The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. The labeled statement or block can be down or up in the sequence of statements.

Example : Using a Simple GOTO Statement

```
DECLARE
p VARCHAR2(30);
n PLS_INTEGER := 37; — test any integer > 2 for prime
BEGIN
FOR j in 2..ROUND(SQRT(n)) LOOP
IF n MOD j = 0 THEN -- test for prime
p := ' is not a prime number'; — not a prime number
GOTO print_now;
END IF;
END LOOP;
p := ' is a prime number';
<<print_now>>
DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

• **Using the NULL Statement:**

The NULL statement does nothing, and passes control to the next statement. Some languages refer to such an instruction as a no-op (no operation).**Example: Using the NULL Statement to Show No Action**

```
DECLARE
v_job_id VARCHAR2(10);
v_emp_id NUMBER(6) := 110;
BEGIN
SELECT job_id INTO v_job_id FROM employees WHERE employee_id
= v_emp_id;
IF v_job_id = 'SA_REP' THEN
UPDATE employees SET commission_pct = commission_pct * 1.2;
ELSE
NULL; — do nothing if not a sales representative
END IF;
END;
```


12.7 SUMMARY

This chapter surveys the main features of PL/SQL and points out the advantages they offer. It also acquaints you with the basic concepts behind PL/SQL and the general appearance of PL/SQL programs. You see how PL/SQL bridges the gap between database technology and procedural programming languages.

12.8 REFERENCES

1. Database System and Concepts A Silberschatz, H Korth, S Sudarshan McGraw-Hill Fifth Edition
2. "Fundamentals of Database Systems" by Elmsari, Navathe, 5th Edition, Pearson Education (2008).
3. "Database Management Systems" by Raghu Ramakrishnan, Johannes Gehrke, McGraw Hill Publication.
4. "Database Systems, Concepts, Design and Applications" by S.K.Singh, Pearson Education.

12.9 UNIT END QUESTIONS

MCQs:

1. Which of the following is not a section in PLSQL Block?
   a. Endif
   b. Begin

c. Declare
d. Exception

2. _____blocks are PL/SQL blocks which do not have any names assigned to them.
   a. Consistent
   b. Named
   c. Anonymous
   d. Begin

3. _____blocks are PLSQL blocks have a specific or unique name assigned to them.
   a. Consistent
   b. Named
   c. Anonymous
   d. Begin

4. Named PLSQL block always start with _____ Keyword
   a. Declare
   b. Begin
   c. Create
   d. End

5. Words used in a PL/SQL block are called _____
   a. Numerals
   b. Symbols
   c. Compound Units
   d. Lexical Units

6. Assignment Operator in PLSQL block
   a. (:!=)
   b. (:=)
   c. (:==)
   d. (:&&)

7. Which of the following is not an executable statements supported by PLSQL?
   a. Insert
   b. Grant
   c. Select
   d. Update

8. _____uses a selector which is an expression whose value is used to return one of the several alternatives.
   a. IF Else
   b. Searched Case
   c. Loop
   d. CASE

9. Which of the following loop is not supported by PLSQL?
   a. While
   b. Do While
   c. For
   d. Loop

10. Which of the following is syntactically correct for declaring and assigning value e to a variable in PLSQL Block?
    a. a int=5;
    b. b =: int 7;
    c. c int :=10;
    d. d int=12:;

Answer the following:

1. Explain advantages of PL/SQL
2. Explain PL/SQL block structure.
3. Explain scalar data types
4. Explain the following:

   i)%Type

   ii) % Rowtype

   iii) Sequences in PL/SQL

   iv) Bind Variables
5. Explain various data types conversion functions with examples.
6. Write a PL/SQL block to demonstrate cube of an input number
7. Write a PL/SQL program to demonstrate the use of basic arithmetic operators on the numbers input by user.
8. Write a PL/SQL program to find out square of inputed number by the user.
9. Explain various looping/iterative constructs in PL/SQL. 9. 10. Explain various conditional statements in PL/SQL.

*****

13

CURSORS, PROCEDURE AND FUNCTIONS

Unit Structure

13.1 PL/SQL-CURSORS

Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors “

- Implicit cursors
- Explicit cursors

Implicit Cursors:

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement.

The following table provides the description of the most used attributes

Sr. No	Attribute & Description
1.	**%FOUND**Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2.	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3.	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4.	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example:

We will be using the CUSTOMERS table we had created and used in the previous chapter

Select * from customers;

```
+----+----------+------+-----------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY | +-
+----+----------+------+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi| 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP   | 4500.00 |
+----+----------+------+-----------+----------+
```

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected

```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE customers
   SET salary = salary + 500;
IF sql%notfound THEN
   dbms_output.put_line('no customers selected');
ELSIF sql%found THEN
   total_rows := sql%rowcount;
   dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result

```
6 customers selected

PL/SQL procedure successfully completed
```

If you check the records in customers table, you will find that the rows have been updated "

Select * from customers;

```
+----+------------+-------+-----------------+--------+
| ID | NAME    | AGE | ADDRESS | SALARY | +-
-----------+--------+------------+------------+------------+
| 1 | Ramesh | 32 | Ahmedabad | 2500.00 | |
2 | Khilan | 25 | Delhi  | 2000.00 |
| 3 | kaushik | 23 | Kota           | 2500.00 |
| 4 | Chaitali | 25 | Mumbai | 7000.00 | |
5 | Hardik | 27 | Bhopal          | 9000.00 | |
6 | Komal | 22 | MP         | 5000.00 |
-------+------------+------------+------------+
```

Explicit Cursors:

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the

declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is

CURSOR cursor_name IS select_statement;

Working with an explicit cursor includes the following steps:

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor:

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

CURSOR c_customers IS

SELECT id, name, address FROM customers;

Opening the Cursor:

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows:

```
OPEN c_customers;
```

Fetching the Cursor:

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows:

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor:

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows:

CLOSE c_customers;

Example:

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
c_id customers.id%type;
c_name customerS.No.ame%type;
c_addr customers.address%type;
CURSOR c_customers is
   SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
FETCH c_customers into c_id, c_name, c_addr;
   EXIT WHEN c_customers%notfound;
   dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
END LOOP;
   CLOSE c_customers;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

1. Ramesh Ahmedabad
2. Khilan Delhi
3. kaushik Kota
4. Chaitali Mumbai
5. Hardik Bhopal
6. Komal MP

PL/SQL procedure successfully completed.

PL/SQL-Transactions:

A database **transaction** is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed, i.e., made permanent to the database or rolled back (undone) from the database.

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

Transaction:

A transaction has a **beginning** and an **end**. A transaction starts when one of the following events take place

- The first SQL statement is performed after connecting to the database.
- At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place “

- A **COMMIT** or a **ROLLBACK** statement is issued.
- A **DDL** statement, such as **CREATE TABLE** statement, is issued; because in that case a COMMIT is automatically performed.
- A **DCL** statement, such as a **GRANT** statement, is issued; because in that case a COMMIT is automatically performed.
- User disconnects from the database.
- User exits from **SQL*PLUS** by issuing the **EXIT** command, a COMMIT is automatically performed.
- SQL*Plus terminates abnormally, a **ROLLBACK** is automatically performed.
- A **DML** statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

Committing a Transaction:

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is

```
COMMIT
```

For example,

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (1, ‘Ramesh’, 32, ‘Ahmedabad’, 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (2, ‘Khilan’, 25, ‘Delhi’, 1500.00 );


INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (3, ‘kaushik’, 23, ‘Kota’, 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
```

VALUES (4, ‘Chaitali’, 25, ‘Mumbai’, 6500.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (5, ‘Hardik’, 27, ‘Bhopal’, 8500.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (6, ‘Komal’, 22, ‘MP’, 4500.00);

COMMIT;

Rolling Back Transactions:

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

The general syntax for the ROLLBACK command is:

ROLLBACK [TO SAVEPOINT < savepoint_name>];

When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If you are not using **savepoint**, then simply use the following statement to rollback all the changes

ROLLBACK;

Savepoints:

Savepoints are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting savepoints within a long transaction, you can roll back to a checkpoint if required. This is done by issuing the **SAVEPOINT** command.

The general syntax for the SAVEPOINT command is

SAVEPOINT < savepoint_name >;

For example:

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (7, ‘Rajnish’, 27, ‘HP’, 9500.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (8, ‘Riddhi’, 21, ‘WB’, 4500.00);

SAVEPOINT sav1;

UPDATE CUSTOMERS

SET SALARY = SALARY + 1000;

```
ROLLBACK TO sav1;
UPDATE CUSTOMERS

SET SALARY = SALARY + 1000
WHERE ID = 7;
UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 8;

COMMIT;
```

ROLLBACK TO sav1: This statement rolls back all the changes up to the point, where you had marked savepoint sav1.

After that, the new changes that you make will start.

Automatic Transaction Control

To execute a **COMMIT** automatically whenever an **INSERT, UPDATE** or **DELETE** command is executed, you can set the **AUTOCOMMIT** environment variable as:

```
SET AUTOCOMMIT ON;
```

You can turn-off the auto commit mode using the following command "

```
SET AUTOCOMMIT OFF;
```

13.2 COLLECTIONS AND COMPOSITE DATA TYPES

PL/SQL-Collections

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types

- Index-by tables or Associative array
- Nested table
- Variable-size array or Varray

Oracle documentation provides the following characteristics for each type of collections:

Collection Type	Number of Elements	Subscript Type	Dense or Sparse	Where Created	Can Be Object Type Attribute
Associative array (or index-by table)	Unbounded	String or integer	Either	Only in PL/SQL block	No
Nested table	Unbounded	Integer	Starts dense, can become sparse	Either in PL/SQL block or at schema level	Yes
Variablesize array (Varray)	Bounded	Integer	Always dense	Either in PL/SQL block or at schema level	Yes

We have already discussed varray in the chapter **'PL/SQL arrays'**. In this chapter, we will discuss the PL/SQL tables.

Both types of PL/SQL tables, i.e., the index-by tables and the nested tables have the same structure and their rows are accessed using the subscript notation. However, these two types of tables differ in one aspect; the nested tables can be stored in a database column and the index-by tables cannot.

Index-ByTable:

An **index-by** table (also called an **associative array**) is a set of **key-value**pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an **index-by** table named **table_name**, the keys of which will be of the subscript_type and associated values will be of the *element_type*

TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY subscript_type;

table_name type_name;

Example:
Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

```
DECLARE
      TYPE salary IS TABLE OF NUMBER INDEX BY
      VARCHAR2(20);
      salary_list salary;
      nam e VARCHAR2(20);
BEGIN
```

— adding elements to the table
salary_list('Rajnish') := 62000;
salary_list('Minakshi') := 75000;
salary_list('Martin') := 100000;
salary_list('James') := 78000;

— printing the table
name := salary_list.FIRST;
WHILE name IS NOT null LOOP
      dbms_output.put_line
      ('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
      name := salary_list.NEXT(name);
END LOOP;
END;
/

When the above code is executed at the SQL prompt, it produces the following result:

```
Salary of James is 78000
Salary of Martin is 100000
Salary of Minakshi is 75000
Salary of Rajnish is 62000

PL/SQL procedure successfully completed
```

Example:

Elements of an index-by table could also be a **%ROWTYPE** of any database table or **%TYPE** of any database table field. The following example illustrates the concept. We will use the **CUSTOMERS** table stored in our database as:

Select * from customers

```
+----+---------+---+------------+-----------+

| ID | NAME    | AGE | ADDRESS | SALARY |
+------+--------------+-----+------------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi| 1500.00 |
```

```
| 3 | kaushik | 23 | Kota     | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP   | 4500.00 |
+----+------------+---------------+--------------+-------------+
```

```
DECLARE
CURSOR c_customers is
select name from customers;

TYPE c_list IS TABLE of customers.Name%type INDEX BY
binary_integer;
    name_list c_list;
    counter integer :=0;
BEGIN
    FOR n IN c_customers LOOP
        counter := counter +1;
        name_list(counter) := n.name;
        dbms_output.put_line('Customer('||counter||'):'||name_lis
        t(counter));
    END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal

PL/SQL procedure successfully completed
```

NestedTables:

A **nested table** is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects:

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
- An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A nested table is created using the following syntax “

TYPE type_name IS TABLE OF element_type [NOT NULL];

table_name type_name;

This declaration is similar to the declaration of an **index-by** table, but there is no **INDEX BY** clause.

A nested table can be stored in a database column. It can further be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

Example:

The following examples illustrate the use of nested table:

```
DECLARE
   TYPE names_table IS TABLE OF VARCHAR2(10);
   TYPE grades IS TABLE OF INTEGER;
   names names_table;
   marks grades;
   total integer;
BEGIN
   names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
   marks:= grades(98, 97, 78, 87, 92);
   total := names.count;
   dbms_output.put_line('Total '|| total || ' Students');
FOR i IN 1 .. total LOOP
   dbms_output.put_line('Student:'||names(i)||', Marks:' || marks(i));
end loop;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

```
Total 5 Students
Student:Kavita, Marks:98
Student:Pritam, Marks:97
Student:Ayan, Marks:78
Student:Rishav, Marks:87
Student:Aziz, Marks:92

PL/SQL procedure successfully completed.
```

Example

Elements of a **nested table** can also be a **%ROWTYPE** of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as:

```
Select * from customers;
+----+--------------+-----+-------------+-------------+
ID | NAME     | AGE | ADDRESS | SALARY |
+----+--------------+-----+-------------+-------------+
|1| Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan    | 25 | Delhiv| 1500.00 |
| 3 | kaushik  | 23 | Kota| 2000.00 |
| 4 | Chaitali |25 | Mumbai  |6500.0|
| 5 | Hardik  | 27 | Bhopal  |8500.00 |
| 6 | Komal  |22 | MP |4500.00 |
+----+--------------+-----+-------------+-------------+
DECLARE
  CURSOR c_customers is
    SELECT name FROM customers;
  TYPE c_list IS TABLE of customerS.No.ame%type;
  name_list c_list := c_list();
  counter integer :=0;
BEGIN
  FOR n IN c_customers LOOP
    counter := counter +1;
    name_list.extend;
    name_list(counter) := n.name;
ms_output.put_line('Customer('||counter||'):'||name_list(counter));
    END LOOP;
   END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal

PL/SQL procedure successfully completed.
```

Collection Methods:

PL/SQL provides the built-in collection methods that make collections easier to use. The following table lists the methods and their purpose:

Sr. No	Method Name & Purpose
1	**EXISTS(n)** Returns TRUE if the nth element in a collection exists; otherwise returns FALSE.
2	**COUNT** Returns the number of elements that a collection currently contains.
3	**LIMIT** Checks the maximum size of a collection.
4	**FIRST** Returns the first (smallest) index numbers in a collection that uses the integer subscripts
5	**LAST** Returns the last (largest) index numbers in a collection that uses the integer subscripts.
6	**PRIOR(n)** Returns the index number that precedes index n in a collection.
7	**NEXT(n)** Returns the index number that succeeds index n.
8	**EXTEND** Appends one null element to a collection.
9	**EXTEND(n)** Appends n null elements to a collection
10	**EXTEND(n,i)** Appends **n** copies of the ith element to a collection.
11	**TRIM** Removes one element from the end of a collection.
12	**TRIM(n)** Removes **n** elements from the end of a collection.
13	**DELETE** Removes all elements from a collection, setting COUNT to 0.
14	**DELETE(n)** Removes the **nth** element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If **n** is null, **DELETE(n)** does nothing.
15	**DELETE(m,n)** Removes all elements in the range **m..n** from an associative array or nested table. If **m** is larger than **n** or if **m** or **n** is null, **DELETE(m,n)**does nothing.

Collection Exceptions:

The following table provides the collection exceptions and when they are raised:

Collection	Exception Raised in Situations
COLLECTION_IS_NULL	You try to operate on an atomically null collection.
NO_DATA_FOUND	A subscript designates an element that was deleted, or a nonexistent element of an associative array.
SUBSCRIPT_BEYOND_COUNT	A subscript exceeds the number of elements in a collection.
SUBSCRIPT_OUTSIDE_LIMIT	A subscript is outside the allowed range.
VALUE_ERROR	A subscript is null or not convertible to the key type. This exception might occur if the key is defined as a **PLS_INTEGER** range, and the subscript is outside this range.

Composite Data Type:

A **composite data type** stores values that have internal components. You can pass entire composite variables to subprograms as parameters, and you can access internal components of composite variables individually. Internal components can be either scalar or composite. You can use scalar components wherever you can use scalar variables. PL/SQL lets you define two kinds of composite data types, collection and record. You can use composite components wherever you can use composite variables of the same type.

Composite data types falls in two categories:
1) PL/SQL Records
2) PL/SQL Collections

PL/SQL Records:

Records are another type of datatypes which oracle allows to be defined as a placeholder. Records are composite datatypes, which means it is a combination of different scalar datatypes like char, varchar, number etc. Each scalar data types in the record holds a value. A record can be visualized as a row of data. It can contain all the contents of a row.

The General Syntax to define a composite datatype is:

```
TYPE record_type_name IS RECORD
(first_col_name column_datatype,
```

second_col_name column_datatype, ...);

record_type_name – it is the name of the composite type you want to define.

first_col_name, second_col_name, etc.,- it is the names the fields/columns within the record.

column_datatype defines the scalar datatype of the fields.

There are different ways you can declare the datatype of the fields.

1) You can declare the field in the same way as you declare the fieds while creating the table.
2) If a field is based on a column from database table, you can define the field_type as follow

   col_name table_name.column_name%type;

The General Syntax to declare a record of a user-defined datatype is:
record_name record_type_name;

If all the fields of a record are based on the columns of a table, we can declare the record as follows:

record_name table_name%ROWTYPE;

Select using %RowType attribute

1) To get the names of the products and price having unit price than 1000

```
Create table product
select * from
product

set serveroutput on;
Declare
        cursor cur_prod is
        Select pname,unitprice from product;
        prec product%rowtype;
Begin
for prec in cur_prod loop
            dbms_output.put_line(prec.pname||' '||prec.unitprice);
End loop;
End;
```

PL/SQL Collection:

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types:

- Index-by tables or Associative array
- Nested table

- Variable-size array or Varray

Index-By Table:

An **index-by** table (also called an **associative array**) is a set of **key-value** pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an **index-by** table named **table_name**, the keys of which will be of the subscript_type and associated values will be of the *element_type*

TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY subscript_type;

table_name type_name;

Example:

Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

```
DECLARE
TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
salary_list salary;
name VARCHAR2(20);
BEGIN
— adding elements to the table
salary_list('Rajnish') := 62000;
salary_list('Minakshi') := 75000;
salary_list('Martin') := 100000;
salary_list('James') := 78000;
— printing the table
name := salary_list.FIRST;
WHILE name IS NOT null LOO
dbms_output.put_line
('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
name := salary_list.NEXT(name);
END LOOP; ·END;
/
```

When the above code is executed at the SQL prompt, it produces the following result

Salary of James is 78000

Salary of Martin is 100000

Salary of Minakshi is 75000

Salary of Rajnish is 62000

PL/SQL procedure successfully completed.

Example

Elements of an index-by table could also be a **%ROWTYPE** of any database table or **%TYPE** of any database table field. The following example illustrates the concept. We will use the **CUSTOMERS** table stored in our database as:

Select * from customers;

```
+----+--------------+------+-------------+-------------+
| ID | NAME    | AGE | ADDRESS | SALARY |
+----+--------------+------+-------------+-------------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi| 1500.00 |
| 3 | kaushik | 23 | Kotal 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP    | 4500.00 |
+----+--------------+------+-------------+-------------+
```

```
DECLARE
CURSOR c_customers is
select name from customers;
TYPE c_list IS TABLE of customers.Name%type INDEX BY
binary_integer;
name_list c_list;
counter integer :=0;
BEGIN
FOR n IN c_customers LOOP counter := counter +1;
name_list(counter) := n.name;
dbms_output.put_line('Customer('||counter||'):'||name_lis t(counter));
END LOOP; ·END;
```

When the above code is executed at the SQL prompt, it produces the following result:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
PL/SQL procedure successfully completed
```

13.3 PROCEDURES AND FUNCTIONS

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms "

- **Functions** " These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** " These subprograms do not return a value directly; mainly used to perform an action.

CreatingaProcedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows :

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
      < procedure_body >
END procedure_name;
```

Where

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Executing a Standalone Procedure

A standalone procedure can be called in two ways:

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as:

EXECUTE greetings;

The above call will display

Hello World

PL/SQL procedure successfully complete

The procedure can also be called from another PL/SQL block "

BEGIN

greetings;

END;

/

The above call will display:

Hello World

PL/SQL procedure successfully completed.

ParameterModesinPL/SQLSubprograms

The following table lists out the parameter modes in PL/SQL subprograms:

Sr. No.	Parameter Mode & Description
1	**IN**An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.
2.	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**

3.	**IN OUT** An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

Creating a Function:

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows:


```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])] RETURN
return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.

The AS keyword is used instead of the IS keyword for creating a standalone function.

Example:

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the PL/SQL Variables chapter:

```
Select * from customers;
+----+------------+-----+-------------+---------------+
| ID | NAME    | AGE | ADDRESS | SALARY |
+----+------------+-----+-------------+---------------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi| 1500.00 |
| 3 | kaushik | 23 | Kota           | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP    | 4500.00 |
+----+------------+-----+-------------+---------------+
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM customers;

RETURN total;
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result:

Function created.

Calling a Function:

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block :

```
DECLARE
   c number(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

```
Total no. of Customers: 6

PL/SQL procedure successfully completed.
```

Example:

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
a number;
b number;
c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
z number;
BGIN
IF x > y THEN
z:= x;
ELSE
Z:= y;
END IF;
```

```
RETURN z;
END;
BEGIN
a:= 23;
b:= 45;
c := findMax(a, b);
dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

```
Maximum of (23,45): 45
PL/SQL procedure successfully completed.
```

PL/SQL Recursive Functions:

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as:

```
n! = n*(n-1)!
= n*(n-1)*(n-2)!
...
= n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively:

```
DECLARE
    num number;
    factorial number;
FUNCTION fact(x number)
RETURN number
IS
    f number;
BEGIN
    IF x=0 THEN
f := 1;
    ELSE
    := x * fact(x-1);
```

```
END IF;
RETURN f;
END;

BEGIN
    num:= 6;
    factorial := fact(num);
    dbms_output.put_line(' Factorial 'll num ll ' is ' ll factorial);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

```
torial 6 is 720
PL/SQL procedure successfully completed.
```

13.4 EXCEPTIONS HANDLING

An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions "

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling:

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN***:

```
DECLARE
    <declarations section>
BEGIN
    <executable command(s)>
EXCEPTION
    <exception handling goes here >
   WHEN exception1 THEN
    exception1-handling-statements
   WHEN exception2 THEN
    exception2-handling-statements
   WHEN exception3 THEN
    exception3-handling-statements
.....
```

```
WHEN others THEN
    exception3-handling-statements
END;
```

Example:

Let us write a code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
    c_id customers.id%type := 8;
    c_name customerS.No.ame%type;
    c_addr customers.address%type;
BEGIN
SELECT name, address INTO c_name, c_addr
    FROM customers
    WHERE id = c_id;
    DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

EXCEPTION
WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

```
No such customer!
PL/SQL procedure successfully completed.
```

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND**, which is captured in the**EXCEPTION block**.

Raising Exceptions:

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception:

```
DECLARE
```

```
    exception_name EXCEPTION;
BEGIN
    IF condition THEN
RAISE exception_name;
    END IF;
EXCEPTION
    WHEN exception_name THEN
    statement;
END;
```

You can use the above syntax in raising the Oracle standard exception or any user-defined exception. In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

User-defined Exceptions:

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure**DBMS_STANDARD.RAISE_APPLICATION_ERROR**.

The syntax for declaring an exception is:

```
DECLARE
my-exception EXCEPTION;
```

Example:

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception **invalid_id** is raised.

```
DECLARE
    c_id customers.id%type := &cc_id;
    c_name customerS.No.ame%type;
    c_addr customers.address%type;
    — user defined exception
    ex_invalid_id EXCEPTION;
BEGIN
    IF c_id <= 0 THEN
    RAISE ex_invalid_id;
ELSE
    SELECT name, address INTO c_name, c_addr
    FROM customers
    WHERE id = c_id;
```

```
    DBMS_OUTPUT.PUT_LINE ('Name: '||c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
END IF;

EXCEPTION
   WHEN ex_invalid_id THEN
      dbms_output.put_line('ID must be greater than zero!');
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

```
Enter value for cc_id: -6 (let's enter a
value -6) old 2: c_id
customers.id%type := &cc_id;
new 2: c_id
customers.id%type := -6;
ID must be greater than zero!

PL/SQL procedure successfully completed.
```


Pre-defined Exceptions:

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions:

Exception	Oracle Error	SQLCODE	Description
ACCESS_INTO_NULL .	06530	-6530	It is raised when a null object is automatically assigned a value
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.

COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX .	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.
NOT_LOGGED_ON .	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an

			internal problem
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR .	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

13.5 PACKAGES

Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms.

A package will have two mandatory parts

- Package specification
- Package body or definition

Package Specification:

The specification is the interface to the package. It just **DECLARES** the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a**private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
```

When the above code is executed at the SQL prompt, it produces the following result:

```
Package created
```

Package Body:

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from the code outside the package.

The **CREATE PACKAGE BODY** Statement is used for creating the package body. The following code snippet shows the package body declaration for the***cust_sal*** package created above. I assumed that we already have CUSTOMERS table created in our database as mentioned in the PL/SQL - Variableschapter.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS

PROCEDURE find_sal(c_id customers.id%TYPE) IS
c_sal customers.salary%TYPE;
BEGIN
   SELECT salary INTO c_sal
   FROM customers
   WHERE id = c_id;
   dbms_output.put_line('Salary: '|| c_sal);
END find_sal;
END cust_sal;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

```
Package body created.
```

Using the Package Elements:

The package elements (variables, procedures or functions) are accessed with the following syntax:

package_name.element_name;

Consider, we already have created the above package in our database schema, the following program uses the ***find_sal*** method of the ***cust_sal*** package:

```
DECLARE
   code customers.id%type := &cc_id;
BEGIN
   cust_sal.find_sal(code);
END;
/
```

When the above code is executed at the SQL prompt, it prompts to enter the customer ID and when you enter an ID, it displays the corresponding salary as follows:

```
Enter value for cc_id: 1
Salary: 3000

PL/SQL procedure successfully completed.
```

Example:

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records:

```
Select * from customer
+----+--------------+------+-----------+---------------+
| ID | NAME    | AGE | ADDRESS | SALARY |
+----+--------------+------+-----------+---------------+
| 1 | Ramesh | 32 | Ahmedabad | 3000.00 |
| 2 | Khilan | 25 | Delhi       | 3000.00 |
| 3 | kaushik | 23 | Kota       | 3000.00 |
| 4 | Chaitali | 25 | Mumbai | 7500.00 |
| 5 | Hardik  | 27 | Bhopal    | 9500.00 |
| 6 | Komal  | 22 | MP          | 5500.00 |
```

The Package Specification:

```
CREATE OR REPLACE PACKAGE c_package AS
```

```
— Adds a customer
    PROCEDURE addCustomer(c_id customers.id%type,
    c_name customerS.No.ame%type,
    c_age customers.age%type,
    c_addr customers.address%type,
    c_sal customers.salary%type);

— Removes a customer
    PROCEDURE delCustomer(c_id customers.id%TYPE);

--Lists all customers
    PROCEDURE listCustomer;

END c_package;
/
```

When the above code is executed at the SQL prompt, it creates the above package and displays the following result "

```
Package created.
```

Creating the Package Body

```
CREATE OR REPLACE PACKAGE BODY c_package AS
   PROCEDURE addCustomer(c_id customers.id%type,
       c_name customerS.No.ame%type,
       c_age customers.age%type,
       c_addr customers.address%type,
       c_sal customers.salary%type)
IS
BEGIN
   INSERT INTO customers (id,name,age,address,salary)
       VALUES(c_id, c_name, c_age, c_addr, c_sal);
END addCustomer;

PROCEDURE delCustomer(c_id customers.id%type) IS
BEGIN
   DELETE FROM customers
   WHERE id = c_id;
END delCustomer;

PROCEDURE listCustomer IS
CURSOR c_customers is
   SELECT name FROM customers;

TYPE c_list is TABLE OF customerS.No.ame%type;
name_list c_list := c_list();
```

```
counter integer :=0;
BEGIN
   FOR n IN c_customers LOOP
   counter := counter +1;
   name_list.extend;
   name_list(counter) := n.name;
   dbms_output.put_line('Customer(' ||counter|| ')'||name_list(counter));
   END LOOP;

END listCustomer;

END c_package;
/
```

The above example makes use of the **nested table**. We will discuss the concept of nested table in the next chapter.

When the above code is executed at the SQL prompt, it produces the following result:

```
Package body created.
```

Using The Package:

The following program uses the methods declared and defined in the package*c_package*.

```
DECLARE
    code customers.id%type:= 8;

BEGIN
    c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
    c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
    c_package.listcustomer;
    c_package.delcustomer(code);
    c_package.listcustomer;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
```

Customer(8): Subham
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish

PL/SQL procedure successfully completed

13.6 HIERARCHICAL QUERIES

If a table contains hierarchical data, then you can select rows in a hierarchical order using the hierarchical query clause:

hierarchical_query_clause::=

Description of hierarchical_query_clause.gif follows
Description of the illustration hierarchical_query_clause.gif

START WITH specifies the root row(s) of the hierarchy.

CONNECT BY specifies the relationship between parent rows and child rows of the hierarchy.

The NOCYCLE parameter instructs Oracle Database to return rows from a query even if a CONNECT BY LOOP exists in the data. Use this parameter along with the CONNECT_BY_ISCYCLE pseudocolumn to see which rows contain the loop. Please refer to CONNECT_BY_ISCYCLE Pseudocolumn for more information.

In a hierarchical query, one expression in condition must be qualified with the PRIOR operator to refer to the parent row. For example,

... PRIOR expr = expr
or
... expr = PRIOR expr

If the CONNECT BY condition is compound, then only one condition requires the PRIOR operator, although you can have multiple PRIOR conditions. For example:

CONNECT BY last_name != ‘King’ AND PRIOR employee_id = manager_id ...
CONNECT BY PRIOR employee_id = manager_id and
PRIOR account_mgr_id = customer_id ...

PRIOR is a unary operator and has the same precedence as the unary + and - arithmetic operators. It evaluates the immediately following expression for the parent row of the current row in a hierarchical query.

PRIOR is most commonly used when comparing column values with the equality operator. (The PRIOR keyword can be on either side of the operator.) PRIOR causes Oracle to use the value of the parent row in the column. Operators other than the equal sign (=) are theoretically possible in CONNECT BY clauses. However, the conditions created by these other operators can result in an infinite loop through the possible combinations. In this case Oracle detects the loop at run time and returns an error.

Both the CONNECT BY condition and the PRIOR expression can take the form of an uncorrelated subquery. However, the PRIOR expression cannot refer to a sequence. That is, CURRVAL and NEXTVAL are not valid PRIOR expressions.

You can further refine a hierarchical query by using the CONNECT_BY_ROOT operator to qualify a column in the select list. This operator extends the functionality of the CONNECT BY [PRIOR] condition of hierarchical queries by returning not only the immediate parent row but all ancestor rows in the hierarchy.

13.7 TRIGGERS

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers:

- Triggers can be written for the following purposes “
- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables

- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers:

The syntax for creating a trigger is:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
    Declaration-statements
BEGIN
    Executable-statements
EXCEPTION
    Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name "Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} "This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} "This specifies the DML operation.
- [OF col_name] "This specifies the column name that will be updated.
- [ON table_name] "This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] "This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] "This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) "This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters:

Select * from customers;

```
+----+----------+-----+-----------+----------+
ID | NAME     | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh    | 32 |    Ahmedabad | 2000.00 |
 |2 | Khilan     | 25 |     Delhi         | 1500.00 |
| 3 | kaushik    | 23 |    Kota          | 2000.00 |
| 4 | Chaitali   | 25 |   Mumbai        | 6500.00 |
 | 5 | Hardik   | 27 |   Bhopal         | 8500.00|z
| 6 | Komal    | 22 |    MP           | 4500.00 |
```

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values;

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:

Trigger created.

The following points need to be considered here:

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.

- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it

again only after the initial changes are applied and the table is back in a consistent state.

- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger:

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table "

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00);

When a record is created in the CUSTOMERS table, the above create trigger,**display_salary_changes** will be fired and it will display the following result:

```
Old salary:
        New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table:

```
UPDATE customersSET
salary = salary + 500
```

WHERE id = 2;

trigger,**display_salary_changes** will be fired and it will display the following result :

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

13.8 SUMMARY

This chapter defines a Cursors and its types. It also describes collections and composite data types, Procedure and Functions. Also gives an idea about Exception Handling and Packages.

13.9 REFERENCES

1. Database System and Concepts A Silberschatz, H Korth, S Sudarshan McGraw-Hill Fifth Edition
2. "Fundamentals of Database Systems" by Elmsari, Navathe, 5th Edition, Pearson Education (2008).
3. "Database Management Systems" by Raghu Ramakrishnan, Johannes Gehrke, McGraw Hill Publication.
4. "Database Systems, Concepts, Design and Applications" by S.K.Singh, Pearson Education.

13.10 UNIT END QUESTIONS

MCQs:

1. Which statements are used to control a cursor variable?
   a. OPEN-FOR
   b. FETCH
   c. CLOSE
   d. All mentioned above
2. Which of the following is used to declare a record?
   a. %ROWTYPE
   b. %TYPE
   c. Both A & B
   d. None of the above
3. Which of the following has a return type in its specification and must return a value specified in that type?
   a. Function
   b. Procedure
   c. Package
   d. None of the above
4. Which collection exception is raised when a subscript designates an element that was deleted, or a nonexistent element of an associative array?
   a. NO_DATA_FOUND
   b. COLLECTION_IS_NULL
   c. SUBSCRIPT_BEYOND_COUNT
   d. SUBSCRIPT_OUTSIDE_LIMIT
5. Observe the following code and fill in the blanks "

```
DECLARE
total_rows number(2);
BEGIN
UPDATE employees
SET salary = salary + 500;
IF _____ THEN
dbms_output.put_line('no employees selected');
```

ELSIF _____ THEN
total_rows := _____;
dbms_output.put_line(total_rows || ' employees selected ');
END IF;
END;
a . %notfound, %found, %rowcount.
b . sql%notfound, sql%found, sql%rowcount.
c . sql%found, sql%notfound, sql%rowcount.
d . %found, %notfound, %rowcount.

6. Which of the following is true about PL/SQL index-by tables?
   A. It is a set of key-value pairs.
   B. Each key is unique and is used to locate the corresponding value.
   C. The key can be either an integer or a string.
   D. All of the above.

7. Which keyword is used instead of the assignment operator to initialize variables?
   a. NOT
   b. DEFAULT
   c. %TYPE
   d. %ROWTYPE

8. Which of the following returns all distinct rows selected by either query?
   a. INTERSECT
   b. MINUS
   c. UNION
   d. UNION ALL

9. For which Exception, if a SELECT statement attempts to retrieve data based on its conditions, this exception is raised when no rows satisfy the SELECT criteria?
   a. TOO_MANY_ROWS
   b. NO_DATA_FOUND
   c. VALUE_ERROR
   d. DUP_VAL_ON_INDEX

10. Which keyword and parameter used for declaring an explicit cursor?
   a. constraint
   b. cursor_variable_declaration
   c. collection_declaration
   d. cursor_declaration

Answer the following:

1. Explain PL/SQL Records with example.
2. Explain Explicit Cursors with example.
3. Explain Attributes of Explicit Cursors with example.

4. Explain the concept of Cursor for Loop with example.
5. Explain for Update clause and where current clause with example.
6. Explain Exception Handling in PL/SQL with example.
7. Differentiate Anonymous blocks and subprograms
8. Differentiate Procedures and Functions
9. Create a PL/SQL function to find out greatest two numbers. Call the function to display output.
10. Explain parts of Triggers with the help of example.

*****