Subject Title	RELATIONAL DATABASE MANAGEMENT SYSTEMS	Semester	III
Subject Code	21UCS04	Specialization	NA
Type	Core: Theory	L:T:P:C	41:3:0:5

COURSE OBJECTIVE:

1. Understand the basic concept of Data Base and database management system.
2. Understand and apply the SQL fundamentals.
3. Evaluate the Relational database design.

CO Number	CO Statement	Knowledge Level
CO1	Remember the concept of database.	K1
CO2	Understanding the data models and ER Diagram.	K2
CO3	Apply SQL commands.	K2, K3
CO4	Evaluate the DBMS in SQL.	K3,K4
CO5	Analyze the Transaction management.	K5

Subject Title	RELATIONAL DATABASE MANAGEMENT SYSTEMS	Semester	III	
Subject Code	21UCA06	Specialization	NA	
Type	Core: Theory	L:T:P:C	41:3:0:5	
Unit	Contents		Levels	Sessions
I	Introduction: Database System Applications-Purpose of Database Systems-View of Data-Database Languages-Transaction Management-Database Architecture-Database users and Administrators. Relational Model: Structure of Relational Databases – Database Design – ER Model-Overview of the Design Process – The Entity – relationship Model – Constraints – Entity Relationship Diagrams.		K1	10
II	Relational Algebra Operations –Relational Languages: The Tuple Relational Calculus –The Domain Relational Calculus – SQL: Background – Data Definition – Basic Structure of SQL Queries – Set Operations – Aggregate Functions – Null Values – Nested Sub-Queries – Views – Modification of the Database.		K2	7
III	Data Normalization: Pitfalls in Relational Database Design – Decomposition – Functional Dependencies – Normalization – First Normal Form – Second Normal Form – Third Normal Form – Boyce-Codd Normal Form – Fourth Normal Form – Fifth Normal Form – Denormalization – Database Security: Data Security Requirements – Protecting the Data within the Database – Granting and Revoking Privileges – Data Encryption.		K2,K3	8
IV	PL/SQL: A programming Language: History - Fundamentals – Block Structure – Comments – Data Types – Other Data Types – Declaration – Assignment operation – Bind variables – Substitution Variables – Printing – Arithmetic Operators. Control Structures and Embedded SQL: Control Structures – Nested Blocks – SQ L IN PL/SQL – Data Manipulation-Transaction Control statements. PL/SQL Cursors and Exceptions: Cursors – Implicit & Explicit Cursors and Attributes – Cursor FOR loops – SELECT...FOR UPDATE – WHERE CURRENT OF clause – Cursor with Parameters – Cursor Variables – Exceptions – Types of Exceptions.		K3,K4	8
V	PL/SQL Composite Data Types: Records – Tables – V arrays. Named Blocks: Procedures – Functions – Packages - Triggers – Data Dictionary Views.		K5	8
	Learning Resources			
Text Books	1. –Database System Concepts‖,Abraham Silberschatz, Henry F.Korth, S.Sudarshan, TMH 5th Edition (Units – I,II) 2. –Fundamentals of Database Management Systems‖, Alexis Leon, Mathews Leon, Vijay Nicole Imprints Private Limited. (Unit-III) 3. –Database Systems Using Oracle‖ Nilesh Shah,2nd edition,PHI.UNIT-IV: Chapters 10 & 11 UNIT-V:Chapters 12,13 & 14.			

Reference Books	1. Alexix Leon & Mathews Leon, "Essential of DBMS", 2nd reprint, Vijay Nicole Publications, 2009.
Website / Link	• https://www.w3schools.com/sql • https://www.tutorialspoint.com/sql • https://livesql.oracle.com

Mapping with Programme Outcomes

CO Number	PO1	PO2	PO3	PO4
CO1	S	S	S	-
CO2	S	M	M	S
CO3	S	L	L	M
CO4	M	S	M	S
CO5	S	L	S	S

S- Strong , M- Medium , L – Low

UNIT – 1

DBMS:

- 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.

DATABASE SYSTEM:

- Database systems are designed to manage large bodies of information.
- Management of data involves both defining structure for storage of information and providing mechanisms for the manipulation of information.

Database System Applications

Databases are widely used. There are some representative applications.

i) Banking: For customer information, accounts, loans and banking transaction.

ii) Airlines: For reservation and schedule information. Airlines were the first to use databases in a geographically distributed manner.

iii) Universities: For student information, course registration and grades.

iv) Credit Card Transaction: For purchase of credit cards and generation of monthly statements.

v) Telecommunication: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards and storing information about the communication networks.

vi) Finance: For storing information about holding sales and purchase of financial instruments such as stocks and bonds.

vii) Sales: For customer products and purchase information.

viii) On-line retailers: For sales data noted above plus on-line order tracking, generation of recommendation lists and maintenance of on-line product evaluations.

ix) Manufacturing: For management of supply chain and for tracking production of items in factories, inventories of items in warehouses and stores and orders for items.

x) Human Resources: For information about employee's salaries, payroll, taxes, and benefits and for generation of paychecks.

Purpose of Database Systems

- Consider part of a bank enterprise that, among other data, keeps information about all customers and savings accounts.
- One way to keep the information on the computer is to store it in operating system files.
- To allow users to manipulate the information, the system has a number of application programs that manipulates the files, including program to

1. Debit of Credit an account
2. Add a new account
3. Find the balance of an account
4. Generate monthly statements

- System programmers wrote these application programs to meet the needs of the bank.
- New application programs are added to the system as the need arises.
- For example, a savings bank decides to offer checking accounts.
- As a result, the bank creates new permanent files that contain information about all the checking accounts maintained in the bank.
- The system acquires more files and more application programs.
- This typical file processing system is supported by a conventional operating system.
- The system stores permanent records in various files, and it needs different application programs to extract records from and add records to the appropriate files.

File processing system has a number of major disadvantages

i) Data redundancy and inconsistency:

1. Since different programmers create the files and application programs over a long period, the various files are likely to have different structures.
2. The programs may be written in several programming languages.
3. For example, the address and telephone number of a particular customer may appear in a file that consists of savings account records and checking account records.
4. This redundancy leads to higher storage and access cost
5. Data inconsistency is the various copies of the same data ma no longer agree.
6. For example, a changed customer address may be reflected in saving account records but not elsewhere in the system.

ii) Difficult in accessing data:

1. Suppose one of the bank officers needs to find out the names of all customers who live within the city 78733 zip code.
2. The officer asks the data – processing department to generate such a list.
3. Because this request was not anticipated when the original system was designed, there is no application program on hand to meet it.
4. An application program to generate the list of all customers.
5. The bank officer has two choices. Either obtains the list of all customers and has the needed information extracted manually or to obtain the original system as designed.

iii) Data Isolation:

1. Data are scattered in various files and files ma be in different formats, writing new application programs to retrieve the appropriate data is difficult.

iv) Integrity Problems:

1. The data values stored in the database must satisfy certain types of consistency constraints.
2. For example, the balance of certain types of bank accounts may never fall below a prescribed amount (say, $25).
3. These constraints in the system by adding appropriate code in the various application programs.
4. However, when new constraints are added, it is difficult to change the programs to enforce them.
5. The problem is compounded when constraints involve several data items from different files.

v) Atomicity Problems:

1. A computer system, like any other mechanical or electrical device, is subject to failure.
2. In many applications, it is crucial to ensure that, once a failure has occurred and has been detected, the data are stored to the consistent state that existed prior to the failure.
3. Consider a program to transfer $50 from account A to B.
4. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account A, but was not credited to account B, resulting in an inconsistent database state.
5. The fund transfer must be atomic – it must happen in it entirely or not at all.

vi) Concurrent – Access Anomalies:

1. The overall performance of the system is improved and a faster response time is possible, many systems allow multiple users to update the data simultaneously.
2. Consider bank account A, Containing $500.
3. If two customers withdraw funds (say $500 and $100) from account A at about the same time, the result of the concurrent executions may leave the account in an incorrect state.
4. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value b the amount being withdrawn, and write the result back.
5. If the two programs run concurrently, the may both read the value $500, and write back $450 and $400, respectively.
6. Depending on which one writes the value last, the account may contain $450 or $400, rather than the correct value of $350.
7. The system must maintain some form the supervision.
8. Because data may be accessed b many different application programs that have not been coordinated previously, however, supervision is difficult to provide.

vii) Security Problems:

1. Not ever user of the database system should be able to access all the data.
2. For example, in a banking system, payroll, personnel need to see only that part of the database that has information about the various bank employees.
3. The do not need access to information about customer accounts.
4. Since application programs are added to the system in an ad hoc manner, it is difficult to enforce such security constraints.
5. These difficulties, among others, have prompted the development of DBMS.

1.3 View of Data

- A DBMS is 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.

i) *1.3.1 Data Abstraction*

- For the system to be usable, it must retrieve data efficiently.
- Since many database system users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify user's interactions with the system.

a) Physical Level

- The lowest level of abstraction describes how the data are actually stored.
- Complex low –level data structures are described in detail.

b) Logical Level

- The next higher level of abstraction describes what data are stored on the database, and what relationships exist among those data.
- The entire database is thus described in terms of a small number of relatively simple structures.
- Although implementation of the simple structures at the logical level may involve complex physical level structures, the user of the logical level does not need to be aware of this complexity.
- The logical level of abstraction is used by database administrators, who must decide what information is to be kept in the database.

c) View level

- The highest level of abstraction describes only part of the entire database.
- Despite the use of simpler structures of the logical level, some complexity remains, because of the large size of the database.
- Man users of the database system will not be concerned with all this information.
- Such users need to access only a part of the database.
- So that their interaction with n the system is simplified, the view level of abstraction is defined.
- The system may provide many views for the same database.
- The concept of data types in programming languages may clarify the distinction among levels of abstraction.
- Most high level programming languages support the notion of a record type.
- For example, in a Pascal – lie language, we may declare a record:

```
Type customer = record
        customer – name: string;
        social – security: string;
        customer – street: string;
        customer – city: string;
    end;
```

- This code defines a new record called customer with three 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 complier hides this level of detail from programmers

- The database system hides many of the lowest – level storage details form database programmers.
- Database administrators may be aware of certain details of the physical organization of the data.
- At the logical level, each such record is described by a tie definition, and the interrelationship and these record types is defined.
- Programmers using a programming language work at this level of abstraction.
- Database administrators usually work at this level of abstraction.
- At the view level, computer user see a set of application programs that hide details of the data types.
- 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 parts of the database.

ii) ***1.3.2 Instances and Schemes***

- Database changes overtime as information is inserted and deleted.
- The collection of information stored in the database at a particular moment called an instance of the database.
- The overall design of the database is called the database schema.
- Schemas are changed infrequently, at all.
- Analogies to the concept of data type, variables, and values in programming language.
- Returning to the customer – record type definition, in declaring the type customer we have not declared any variables
- A database schema corresponds to the programming language type definition.
- A variable of a given type has a particular value at a given instant.
- The value of a variable in programming languages corresponds to an instance of a database schema.
- Database systems have several schemas, at the lowest level is the physical schema. A the intermediate level is the logical schema, and at the highest level is a subschema,

iii) **13.3 Data Independence**

- The ability to modify a schema definition in one level without affecting a schema definition in the next higher level is called data independence.
- There are two levels of data independence:

  - **Physical Data Independence**
    - It is the ability to modify the physical schema without causing application programs to be rewritten.

- Modifications at the physical level are occasionally necessary to improve performance.

- **Logical Data Independence**

- It is the ability to modify the logical schema without causing application programs to be rewritten.
- Modifications at the logical level are necessary whenever the logical structure of the database is altered. For example, money market accounts are added to a banking system.
- Logical data independence is more difficult to achieve than is physical data independence.
- Application programs are heavily dependent on the logical structure of the data.
- The concept of the data independence is similar in many respects to the concept of abstract data types in modern programming language.

DATABASE LANGUAGE

A database system provides two different types of languages: One to specify the database schema, and the other to express database queries and updates.

- The two languages are:

Data Definition Language

- A database schema is specified by a set of definitions expressed by a special language called a data–definition language (DDL).
- The result of compilation of DDL statement is a set of tables that is stored in a special filed called data dictionary, or data directory.
- A data dictionary is a file that contains metadata.
- This file is consulted before actual data ate read or modified in the database system.
- The storage structure and access method used by the database system ate specified by a set of definitions in a special type of DDL called a data storage and definition language.
- The result of compilation of these definitions is a set of instructions to specify the implementation details of the data schemas.

Data Manipulation Language

- The levels of abstraction are not only to the definition of structuring of data, but also to the manipulation of data.

By data manipulation, we mean

- The retrieval of information stored in the database
- The insertion of new information into the database
- The deletion of information from the database.
- The modification of information stored in the database

- At the physical level, we must define algorithms that allow efficient access to data.
- At higher levels of abstraction, we emphasize the case of use.
- The goal is to provide efficient human interaction with the system.

- A data–manipulation language DML is a language that enables users to access or manipulate data as organized by the appropriate data model.
- There are basically two types:

Procedural DMLs

- It requires a user to specify what data are needed and how to get those data.

Nonprocedural DMLs

- It requires a user to specify what data are needed without specifying how to get those data.
- Nonprocedural DMLs are usually easier to learn and use than are procedural DMLs.
- However, since a user does not have to specify how to get the data, these languages may generate code that is not as efficient as that produced by procedural languages.
- A query is a statement requesting the retrieval of information.
- The portion of a DML that involves information retrieval is called a query language.

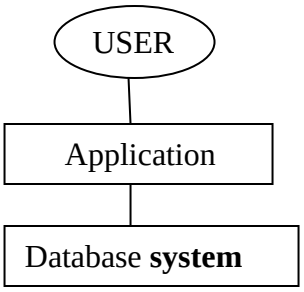
Transaction Management

- Several operations on the database form a single logical unit of work.
- An example, a fund transfer, in which one account (say A) is debited and another account (say B) is credited.
- It is essential that either both the credit and debit occur, or that neither occurs.
- That is, the fund transfer must happen in it do entirely or not at all. This all–or–none requirement is called atomicity.
- In addition, it is essential that the execution of the fund transfer preserve that consistency of the database.
- The value of the sum A+B must be preserved. This correctness requirement is called consistency.
- Finally, after the successful execution of a fund transfer, the new values of accounts A and B persist, despite the possibility of system failure. This persistency 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 database–consistency constraints.
- If the database was consistent a transaction started, the database must be consistent when the transaction successfully terminates.
- During the execution of a transaction, it may be necessary temporarily to allow inconstancy.
- The temporary inconsistency, although necessary, may lead to difficulty if a failure occurs.
- It is the responsibility of the programmer to define properly the various transactions, such that each preserves the consistency of the database.
- For example, the transaction to transfer funds from account A to account B could be defined to be composed of two separate programs. One debits account A, another credits account B.
- Ensuring the atomicity and durability properties is the responsibility of the database system itself–specifically, of the transaction–management component.
- Due to various types of failure, a transaction may not always complete its execution successfully.
- Several transactions update the database concurrently, the consistence of data may no longer be preserved, even though each individual transaction is correct.

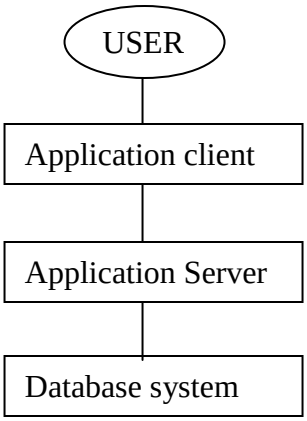
- It is the responsibility of the concurrency – control manager to control the interaction among the concurrent transactions, to ensure the consistency of the database.

DATABASE ARCHITECTURE:

- Database application are usually portioned into two or three parts.
- In a two-tier architecture, the application is portioned into a component that resides at the client machine, which invokes database system functionality at the server machine, which through query language statements.
- Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.
- In contrast in three tier architecture, the client machine acts a merely a front end and does not contain any direct databases calls. Instead the client end communicates with an application server, usually through a form interface.
- The application server in turn communities with a database system to access data.
- The business logic of the application, which says what action to carry out under what condition, 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.



Two-tier



Three-tier

Sophisticated Users:

- Interact with the system without writing programs.
- The form their requests in a database query language.
- Each such query is submitted to a query processor whose function is to break down DML statement into instructions that the storage manager understands.

Specialized Users:

- These are sophisticated users who write specialized database applications that do not fit into the traditional data–processing frame–work.
- These applications are computer – aided design systems, knowledge – base and expert system, systems that store data with complex data types. For example, graphics data and audio data.

Storage Manager:

- The storage manager components provide the interface between the low–level data stored in the database and the application program and queries submitted to the system.
- The storage components include:

1. ***Authorization and integrity manager:***
   - Authorization and integrity manager tests for the satisfaction of integrity constraints and checks the authority of users to access data.
2. ***Transaction manager:***
   - Transaction manager ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction execution proceed without conflicting.
3. ***File manager:***
   - File manager manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
4. ***Buffer manager:***
   - Buffer manager is responsible for fetching data from disj storage into main memory, and deciding what data to cache in memory.
   - Several data structures are required as per part of the physical system implementation.
5. ***Data file:***
   - Data files store the database itself.
6. ***Data dictionary:***
   - Data dictionary stores metadata about the structure of the database.
   - The data dictionary is used heavily. Therefore, great emphasis should be placed on developing a good design and efficient implementation of the dictionary.
7. ***Indices:***
   - An index provides fast access to data items that hold particular values.
8. ***Statistical data:***
   - The store statistical information about the data in the database.
   - This information is used by the query processor to select efficient way to execute a query.

DATABASE USERS AND ADMINISTRATOR:

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

Database user and user interfaces:

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

Navie users are unsophisticates users:

- Who interact with the system by invoking one of the application programs that have been written previously.
- For example, a bank teller who needs to transfer $50 from account A to account B invokes a program called transfer.
- This program asks the teller from the amount of money to be transferred, the account from which the money is to transferred and the account to which the money is to be transferred.

Application programmers:

- Application programmer is computer professionals who wrote application programs.
- Application programmers can choose from many tools to develop user interface.
- Rapid application development tools are tools that enable an application programmer to construct forms and reports without writing a program.
- There are also special types of programming language that combine imperative control statements of the data manipulation language.
- One of the main reasons for using DBMS is to have central control of both the data and the programs that access those data.
- A person who has such cental control over the system is called a database administrator(DBA).
- The function of DBA include:
- The DBA creates the original database schema by executing a set of data definition statements I the DDL. Storage structure and access-method definition.
- **Schema and physical organization modification:**

  The DBA carried out changes to the schema and physical organization reflect the changing need of the organization or to alter the physical organization to improve performance.

- **Granting of authentication for data access:**

  By granting different types if authorization the database administrator can regulate which parts of the database various user can access. The authorization information is kept in a special system structure that the database system consults whenever some one attempts to access the data in the system.

- **Routine maintenance:**

  Examples of the database administrator routine maintenance activities are:

  Periodically backing up the database, either onto tapes or onto remote servers prevent loss of data in case of disasters such as flooding.

  Ensuring that enough free disk space is available for normal operations and upgrading disk space as required.

  Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

RELATIONAL MODEL

- The relational model has established as the primary data model for commercial data– processing applications.
- The database systems were based on either the network model or the hierarchical model.
- A substantial theory exists for relational databases. This theory assists in the design of relational databases and in the efficient processing of user requests for information from the database.

STRUCTURE OF RELATIONAL DATABASES:

- A relational database consists of a collection of Tables, each of assigned a unique name.
- A row in a table represents a Relationship among a set of values.
- Since a table is a collection of such relationships, there is a close correspondence between the concept of table and the mathematical concept of relation, from the relational data model takes its name.

a) Basic Structure

- It has three column headers: branch-name, account-number, and balance.
- For each attribute - there is a set of permitted values, called the Domain of that attribute.
- For the attribute branch-name - for example, the domain is the set of all branch names.

THE ACCOUNT RELATION

Branch–Name	Account–Number	Balance
Downtown	A-101	500
Mianus	A-215	700
Perry ridge	A-102	400
Round Hill	A-305	350
Brighton	A-201	900
Redwood	A-222	700
Brighton	A-217	750

- Let D1 denote the set of all branch-name, D2 denote the set of all account number, and D3 denote the set of all balances.
- An row of account must consist of a 3-tule (v1,v2, v3), where v1 is a branch name(that is, v1 is in domain D1), v2 is an account number(that is, v2 is in domain D2), and v3 is a balance (that is, v3 is in domain D3).
- Account will contain on a subset of the set of all possible rows.
- Therefore, account is a subset of
  - D1 x D2 x D3
- A table of n attributes must be a subset of
  D1 x D2 x ----------x Dn-1 x Dn
- Mathematicians define a relation to be a subset of a Cartesian product of a list of domains.

- It have assigned names to attributes the mathematicians rely on numeric "Names", using the integer 1 - to denote the attribute whose domain appear first in the list of domains, 2- for the attribute whose domain appear second and so on.
- The mathematical terms ***Relation*** and ***Tuple*** in place of the terms ***Table*** and ***Row.***
- The tuple variable t refers to the first tuple of the relation
- The notation t[branch-name] to denote the value of t on the branch-name attribute.
- Thus, t[branch-name] = "Downtown," and t[balance] = 500.
- t[1] to denote the value of tuple t on the first attribute (branch-name), t[2] to denote account number, and so on.
- Since a relation is a set of tuples, the mathematical notation of t € r to denote that tuple t is in relation r.
- All relations r, the domains of all attributes of r be atomic.
- A domain is atomic if elements of the domain are considered to be indivisible units.
- For example, the set of integers is an atomic domain, but the set of all sets of integers is a non–atomic domain.
- One domain value that is a member of a possible domain is the null value, which signifies that the value is unknown or does not exist.
- For example, the attribute telephone-number in the customer relation, a customer does not have a telephone

DATABASE DESIGN:

The database design by observing that this is typically just one part, although central parts in data-intensive applications of a larger software system design.

The database design process can be divided into six steps. The ER model I most relevant to the first three steps.

1. Requirement Analysis: The very first step designing a database application is to understand what data is to be stored in the database, what application must be built on top of it and what operation are most frequent. This can be done by succession with user groups study of current operation environment.
2. Conceptual database design: The information gathered in the requirement analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints known to hold over this data. The ER model is one of several high-level or semantic, data model used in database design.
3. Logical database design: We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model.

ERMODEL:

- The Entity-Relationship(ER) data model is used to develop an initial database design. It provides useful concept that allow to move from an informal description of what users want from their database to more detailed.
- The ER diagram is just an approximate description of the data, constructed through a subjective evaluation of the information collected during requirements analysis. Finally, we must address

security issues and ensure that users are able to access the data they need, but not data that we wish to hide from them.

- Steps:

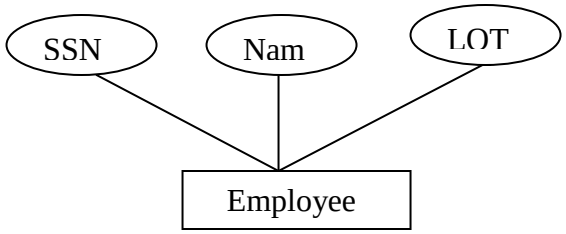
1. **Shema Refinement:** This step is used to analyze the collection of relations in our relational schema to identify potential problems, and to refine it.
2. **Physical Database Design**: This step involves building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier dasign steps.
3. **Application and security Design:** We must identify the enities and processes involved in the application. We must describe the role of each entity in every process that is reflected in some application task, as part of a complete workflow for that task. For each role, we must identify the parts of the database that must be accessible and the parts of th e database that must not be accessible and we must take steps to ensure that these access rules are enforced.

Entities, Attributes and entity set:

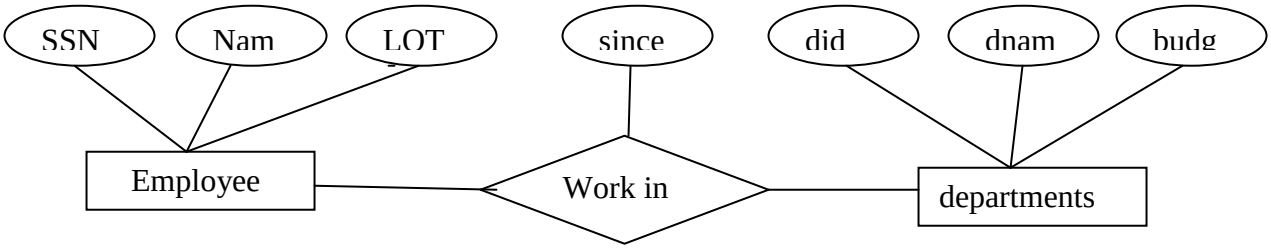
- An entity is an object in the real world that is distinguishable from other objects. Collection of similar entities is called an entity set.
- An entity is described using a set of attributes. All entities in a given entity set have the same attributes this is what we mean by similar.
- For each entity attributes associated with an entity set, we must identify a domain of possible values.
- For each entity set, we choose a key. A key is a minimal set of attributes whose values uniquely identify an entity in the set.
- There could be more than one candidate key, if so, we designate one of them as the primary key. A primary key is that uniquely identifies an entity.
- A relationship is an association among two or more entities. Collection of similar relationship is called relationship set.

  Rectangles, which represent entity sets.
  Ellipses, which represent attributes
  Diamonds, which represent relationship sets
  Lines, which link attributed to entity sets and entity sets to relationship set5s
  Double ellipses, which represent mulivalued attributes
  Dashed ellipses, which denote derived attributes
  Double lines, which indicate total participation of an entity in a relationship set
  Double rectangles, which represent weak entity sets
- We show the relationship set works in which each relationship indicates a department in which an employee works.

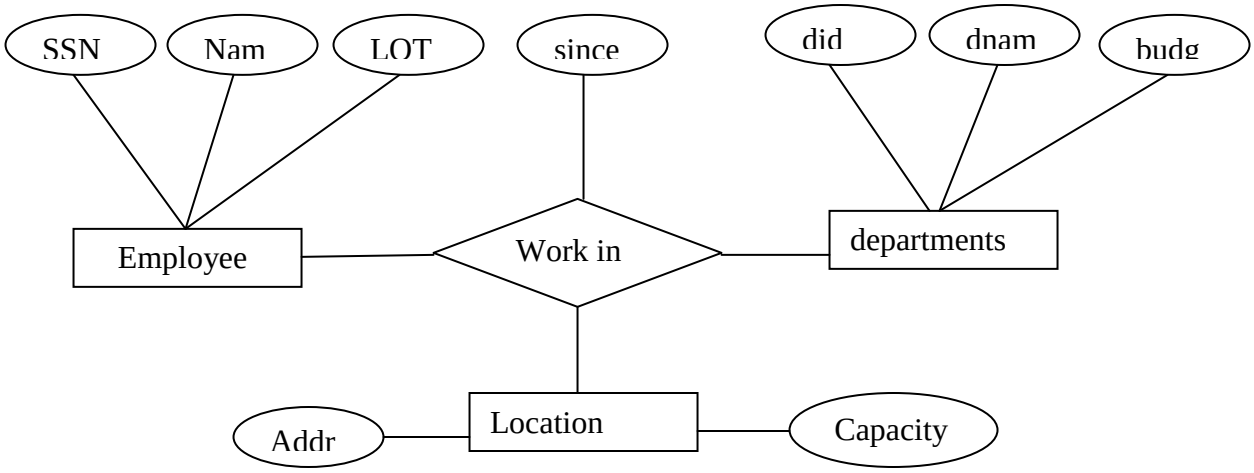
Entity set



- Note that several relationship sets might involve the same entity sets. For example we could also have a manages relationship set involving employees and departments.



- A relationship can also have descriptive attributes. Descriptive attributes are used to record information about the relationship, rather than about any one of the participating entities.
- An instance of a relationship set is a relationship, rather than about any one of the participat9ing entities.
- An instance of a relationship set is a relationship. Intuitively, an instance can be thought of as a snapshot of the relationship set at some instant in time.
- As another example of an ER diagram, suppose that each department has offices in several location and we want to record the location at which each employee works.
- This relationship is ternary because we must record an association between an employee, a department and a location.

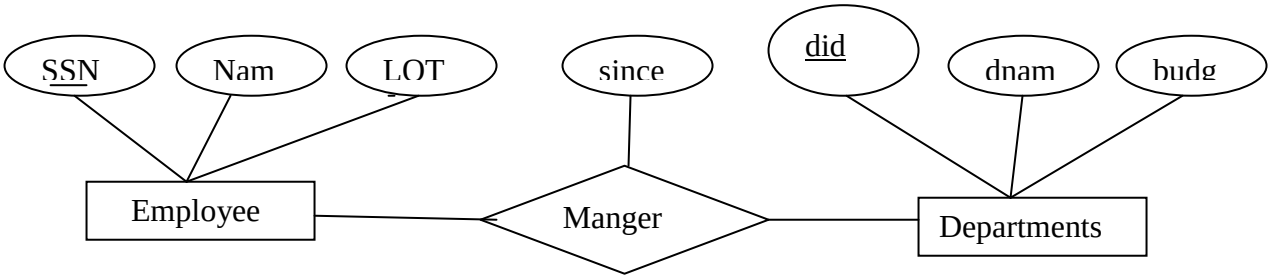


Teranary Relation Set

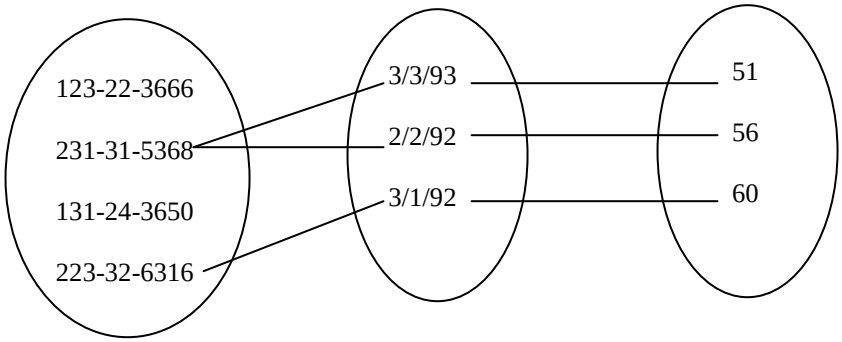
Additional feature of the ER Model:

Key constraints:

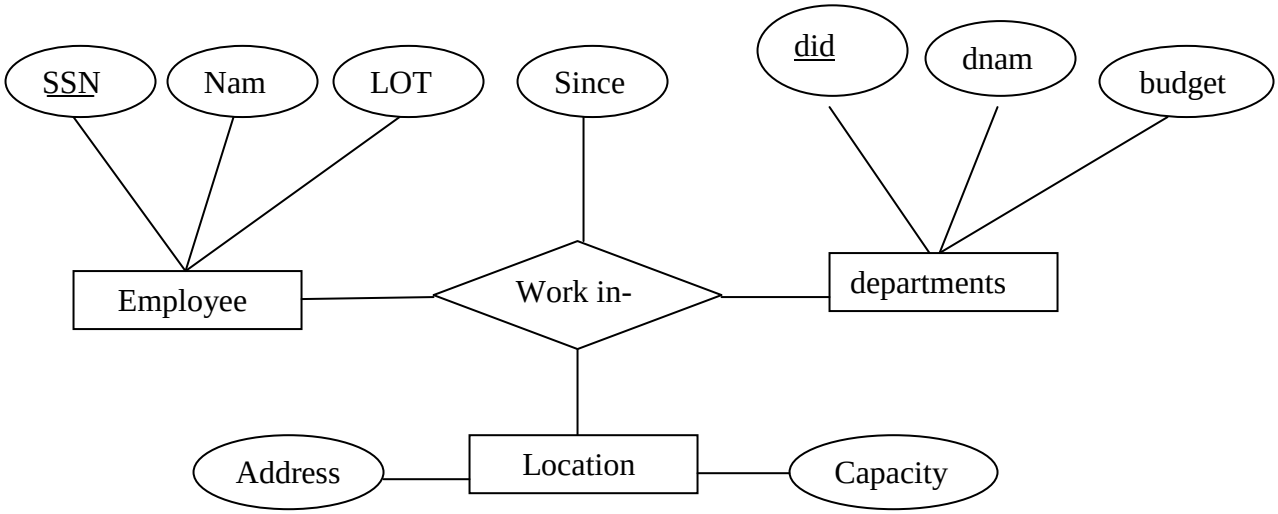
- In relationship, an employee can work in the work in several departments, and a department can have several employees.
- Now consider another relationship set called manages between the employee and departments entity sets such that each department has at most one manager, although a single employee is allowed to manage more than one department.
- The restriction that each department has at most one manager is an example of a key constraint.
- A relationship set like manages is sometimes said to be one-to-many to indicate that one employee can be associated with many departments whereas each department can be associated with at most in employee as its manager.
- In contrast the work in relationship set, in which an employee is allowed to work in several departments and a department is allowed to have several employees, is said to be many-to-many.



Key constraints on manager



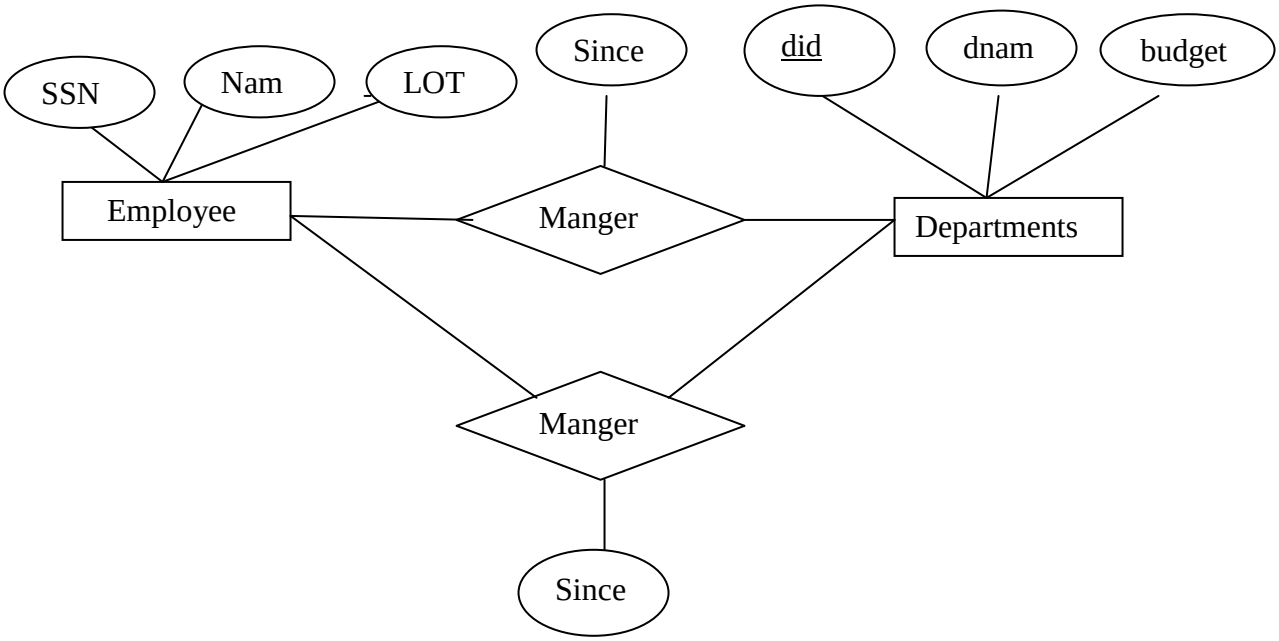
An Instance of the manages Relationship set



Key constraint for Teranary Relation Set

Participation Constraints:

- The key constraint on manages tell us a department has at most one manager. A natural question to ask is whether every department has a manager.
- Let us say that every department is required to have a manager.
- This requirement is an example of a participation constraint.
- The participation of the entity set department in the relationship set manages is said to be a total. A participation that is not total is said to be partial.



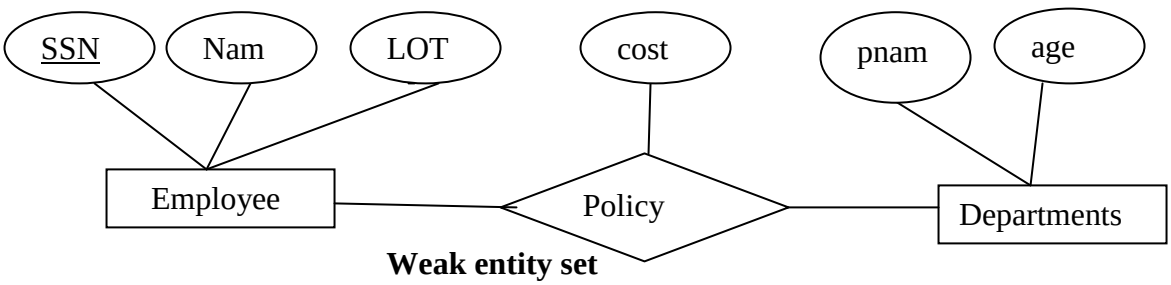
(Manages and work_in)

Weak Entities:

- We have assumed that the attributes associated with an entity set include key. This assumption does not always hold.
- For example, suppose that employees can purchase insurance policies to cover their dependents, we wish to record information about polices, including who is covered by each policy, but this information our only interest in the depends of an employee. If an employee quits, any policy owned by the employee is terminated from the database.
- A dependent is an example of a weak entity set. A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the identifying owner.

The following restrictions must hold:

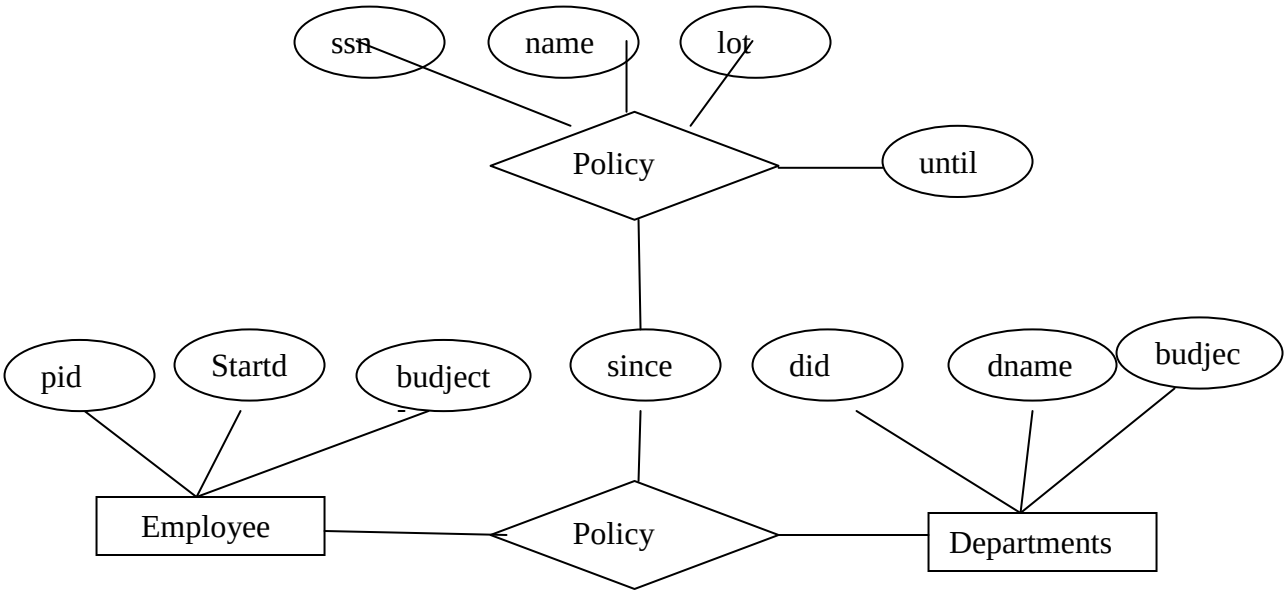
- The owner entity set and weak entity set must participate in a one-to-many relationship set. This relationship set is called identifying relationship set of the weak entity set.
- The weak entity set must have total participation in the identifying relationship set.



Weak entity set

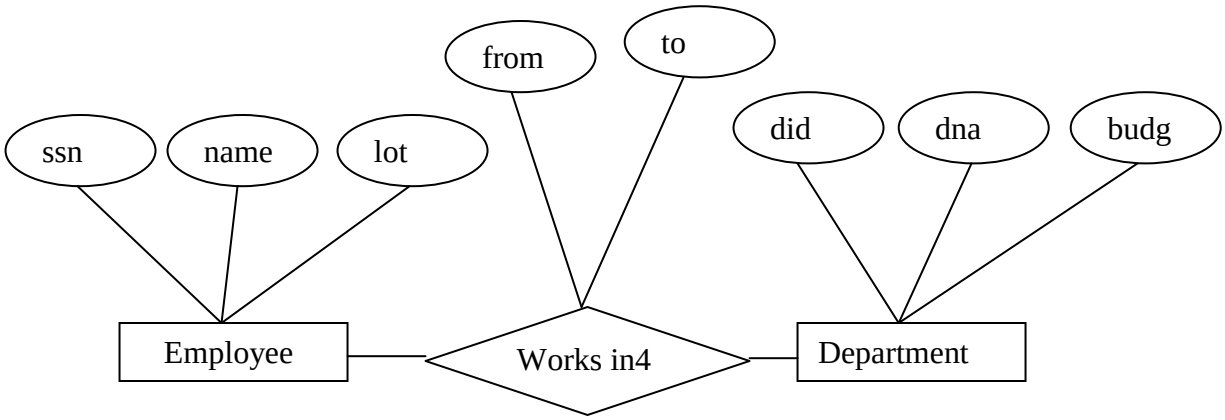
Aggregation:

- A relationship set is an association between entity sets. Sometimes, we have to model a relationship between a collection of entities and relationship.
- Suppose that we have an entity set called projects and that each projects entity is sponsored by one or more departments.
- The sponsors relationship set captures this information.
- A department that sponsors a project might assign employees to monitor the sponsorship. Monitor should be a relationship set that associates a sponsors relationship with an employees entity.
- Aggregation allow us to indicate that a relationship set participates in another relationship set.



Aggregation

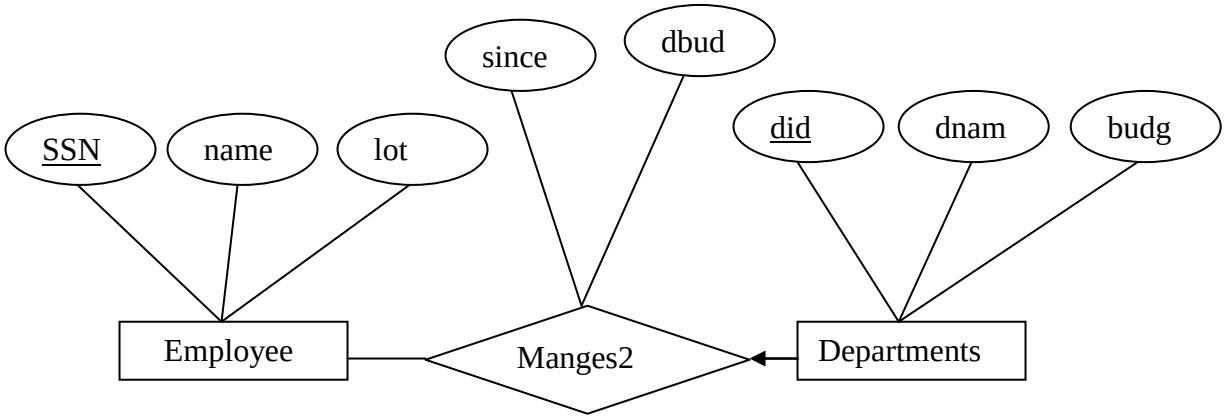
Entity versus attribute:



- While identifying the attributed of an entity set, it is sometimes not clear whether a properly should be modeled as an attribute or as an entity set.
- In the above relationship set has attribute from and to. It records the interval during which an employee works for a department.
- Now suppose that is possible for an employee to work in a given department over more than one period.

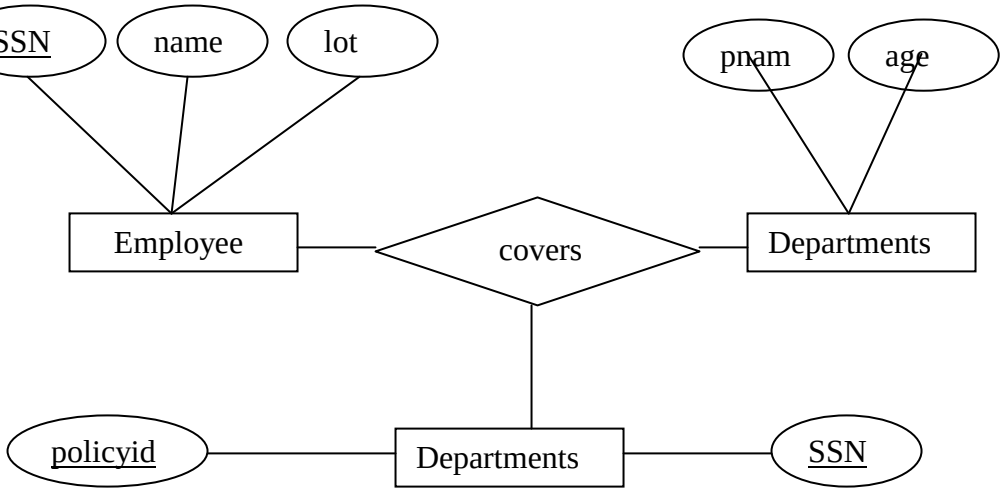
Entity versus Relationship:

- If we assume that a manager receives a separate discretionary budget for each department. But what if the discretionary budget is a sum that covers all departments managed by that employee.



Binary versus Ternary relationship:

Consider the ER diagram shown below.



QUESTION BANK

UNIT – I

5 Marks:

1. Write about Database system application.
2. Explain about view of data.
3. Explain about Database Languages.
4. Explain about transaction management.
5. Explain about database user and administrators.
6. Explain about ER-Model.

10 Marks:

7. Explain briefly about Database Architecture.
8. Explain briefly about Relational model.

UNIT – II

RELATIONAL ALGEBRA

Introduction:

The inputs and outputs of a query are relations. A query used to evaluate using instances of each input relation and it produces an instance of the output relation.

- Query language can be categorized as procedural or nonprocedural.
- In nonprocedural language, the user describes the information desired without giving a specific procedure for obtaining that information (ex: relational calculus)
- In procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result (ex: relational algebra).

Relational algebra:

- Relational algebra is one of the two formal query languages associated with the relational model.
- Queries in algebra are composed using a collection of operators.
- Every operator in the algebra accept relation instance as arguments and returns a relation instance as the result.
- Each relational query describes a step-by-step procedure for computing the answer based on the order in which operators are applied in the query.

SELECTION:

- This operation is used to select the rows / Tuples from a relation, which satisfies the given condition.

- It is denoted by Greek letter sigma (σ). The selection operator σ specifies the tuple to retain through a selection condition.
- The selection condition is a Boolean combination that has the form

Attribute op constant

(Or)

Attribute1 op attribute2

Where op is one of the comparison operator <, <=, =, =, >=, or >. The reference to an attribute can be a position or by name. For example consider the following relation instance.

Student

Sid	Sname	Age
S1	David	23
S2	Smith	19
S3	John	22
S4	Henry	21
S5	Joni	20

- **Find out all the name whose age is greater than 21.**

(student) = $\sigma_{age>21}$

Sid	Sname	Age
S3	John	22
S5	Joni	20

- **Find out all the name that starts with the letter J and age >20**

(student) = $\sigma_{\text{name ="J\%" } \wedge \text{ age}>20}$

Sid	Sname	Age
S3	John	22

Write a note on projection.

PROJECTION:

Projection operator (π) is used to extract columns of the relation, which satisfy the given condition.

For example consider the following relation instance.

Sid	Sname	Age
S1	David	23
S2	Smith	19
S3	John	22
S4	Henry	21
S5	Joni	20

Student

- **Find out all student names in the student relation.**

(student) = π_{Sname}

Sname
David
Smith
John
Henry
Joni

SET OPERATIONS:

Union: (U)

- **R U S** returns a relation instance containing all tuples that occur in either relation instance **R** or relation instance **S** (or both).
- **R** and **S** must be union compatible and the schema of the result is defined to be identical to the schema of **R.**
- Two relation instances are said to be union-compatible it hold two conditions which are
  1. They have the same number of the fields
  2. Corresponding fields taken in order form left to right, have the same domain.

Sid	Sname	Age
21	Madhan	18
20	Kamala	21

Relation R

Sid	Sname	Age
19	Mahesh	30
21	Madhan	18

Relation S

Example

Sid	Sname	Age
21	Madhan	18
20	Kamala	21
19	Mahesh	30

Relation R U S

Intersection: (∩)

- **R ∩ S** returns a relation instance containing all tuples that occur in both **R** and **S**.
- **R** and **S** must be union compatible and the schema of the result is defined to be identical to the schema of **R.**

Sid	Sname	Age
21	Madhan	18

Relation R ∩ S

Set difference: (-)

- **R-S** returns a relation instance containing all tuples that occur in R but not in S.
- Set difference is denoted by **minus (-)**
- **R** and **S** must be union compatible and the schema of the result is defined to be identical to the schema of **R.**

Sid	Sname	Age
20	Kamala	21

Relation R – S

Cross Product: (X)

- **R X S** returns a relation instance whose schema consists of all the fields of R followed by all the fields of S.
- This operation is denoted by a cross **(X).** This operation is sometimes called Cartesian product.
- The result of R X S contains one tuple < r, s> for each pair of tuples r ∈ R, s ∈ S.

Example:

Consider the relations **R** and **S**,

Sid	Sname	Age	Sid	Sname	Age
21	Madhan	18	19	Mahesh	30
21	Madhan	18	21	Madhan	18
20	Kamala	21	19	Mahesh	30
20	Kamala	21	21	Madhan	18

Relation R X S

RENAMING: (ρ)

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 lower-case Greek letter rho (ρ).

oldname new name (or) position newname

Example:

- Consider the expression $\rho(R(F), \overline{E})$ takes an arbitrary relational algebra expression E and returns an instance of a (new) relation called R. R contains the same tuples as the result of E and has the same schema as E but some fields are renamed.
- The expression $\rho_X(E)$ returns the result of expression E under the name X. 'X' contains the same tuples as E, and same scheme as E, but some fields can be renamed. For example, consider the relation

Another form of the rename operation is as follows. Consider a relational-algebra expression E has n attributes. Then, the expression

$$\rho_X(a1,a2,a3\ldots)^{(E)}$$ Consider the query for rename operations

$$\pi_{name}(\sigma_{student.age > d.age}(student \times \rho_d(student)))$$

JOINS:

- It is one of operation in relational algebra that allows us to combine information from two or more relation.
- A join can be defined as a cross product followed by selection and projections.
- The result of a cross product is typically much larger than the result of a join.
- It is denoted by the "join" symbol . It has various types.

➔ **Condition joins:**

It accept a join condition c and a pair of relation instances as arguments and return instance. The join condition is identical to a selection condition in form. The operation is defined as, $R \ S = \sigma_c(R \times S)$

Example:

Sid	Sname	Age
21	Madhan	18
20	Kaman	21
24	Mages	30

Sid	Bid
21	10
20	12
20	10

Relation R X S

Sid	Sname	Age	(Sid)	Bid
21	Madhan	18	21	10
21	Madhan	18	20	12
21	Madhan	18	20	10
20	Kaman	21	21	10
20	Kaman	21	20	12
20	Kaman	21	20	10
24	Mages	30	21	10
24	Mages	30	20	12
24	Mages	30	20	10

Relation R Relation S

Sid	Sname	Age	(Sid)	Bid
21	Madhan	18	20	12
21	Madhan	18	20	10
24	Mages	30	21	10
24	Mages	30	20	12
24	Mages	30	20	10

the result of $\sigma_c(R \times S)$

Equijoin:

When the join operation contains only equalities then the join operation is called equijoin. The scheme of the result of an euijoin contains the fields of R followed by the fileds of S that do not appear in the join condition.

Example:

The result of R (R.Sid = S.Sid) S

Sid	Sname	Age	(Sid)	Bid
21	Madhan	18	21	10
20	Kaman	21	20	12
20	Kaman	21	20	10

➔ **Natural join:**

Natural join is an equijoin in which equalities are specified on all fields having the same name in R and S. Here we can omit the condition; generally, join condition is a collection of equalities on all common fields. If the two relations have no attributes in common, S1 R1 is simply the cross product.

DIVISION:

The division operation used for expressing certain kinds of queries. Consider two relation A and B in which A has exactly two fields x and y and B has just one field y, with the same domain as in A. we define the division operation A/B as the set of all x values such that for every y value in B, there is tuple <x, y> in A. For example, consider the relations A and B1, and B2, In general we can define A/B as,

$\pi_x(A)$ -

Sno	Pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

$\pi_x((\pi_x(A) \text{ X } B) - A)$

B1

Pno
P2

A/B1

Sno
S1
S2
S3
S4

B2

Pno
P2
P4

Sno
S1
S4

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 **nonprocedural** 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 for *t*. Following our earlier notation, we use *t*[*A*] to denote the value of tuple *t* on attribute *A*, andwe use

$t \in r$ to denote that tuple *t* is in relation *r*.

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 *loan* 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

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

In English, we read the preceding expression as "The set of all tuples *t* such that there exists a tuple *s* in 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 (*loannumber*).

Formal Definition

We are now ready for a formal definition. A tuple-relational-calculus expression is of the form

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

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 $\exists$ or $\forall$. Thus, in

$$t \in loan \wedge \exists\, s \in customer(t[branch\text{-}name] = s[branch\text{-}name])$$

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

- $s \in r$, where s is a tuple variable and r is a relation (we do not allow use of the $\notin$ operator)
- $s[x]\ \Theta\ u[y]$, 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 ($<, \leq, =, \neq, >, \geq$); we require that attributes x and y have domains whose members can be compared by Θ
- $s[x]\ \Theta\ c$, 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 P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $P_1 \Rightarrow P_2$.
- If $P_1(s)$ is a formula containing a free tuple variable s, and r is a relation, then

  $$\exists\, s \in r\ (P_1(s)) \text{ and } \forall\, s \in r\ (P_1(s))$$

  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:

1. $P_1 \wedge P_2$ is equivalent to $\neg\,(\neg(P_1) \vee \neg(P_2))$.
2. $\forall\, t \in r\ (P_1(t))$ is equivalent to $\neg\, \exists\, t \in r\ (\neg P_1(t))$.
3. $P_1 \Rightarrow P_2$ is equivalent to $\neg(P_1) \vee P_2$.

The Domain Relational Calculus

A second form of relational calculus, called **domain relational calculus**, uses *domain* variables that take on values from an attributes 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

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

where $x_1, x_2, \ldots, x_n$ 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:

- $< x_1, x_2, \ldots, x_n > \in r$, where r is a relation on n attributes and $x_1, x_2, \ldots, x_n$ are domain variables or domain constants.
- $x \ \Theta \ y$, where x and y are domain variables and Θ is a comparison operator ($<, \leq, =, \neq, >, \geq$). We require that attributes x and y have domains that can be compared by Θ.
- $x \ \Theta \ c$, 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 P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $P_1 \Rightarrow P_2$.
- If $P_1(x)$ is a formula in x, where x is a domain variable, then

$$\exists\, x\, (P_1(x)) \text{ and } \forall\, x\, (P_1(x))$$

are also formulae.

As a notational shorthand, we write

$$\exists\, a, b, c\, (P(a, b, c))$$

for

$$\exists\, a\, (\exists\, b\, (\exists\, c\, (P(a, b, c))))$$

We now give domain-relational-calculus queries for the examples that we considered earlier. Note the similarity of these expressions and the corresponding tuple relational- calculus expressions.

- Find the loan number, branch name, and amount for loans of over $1200:

$\{< l, b, a > \mid < l, b, a > \in loan \wedge a > 1200\}$

- Find all loan numbers for loans with an amount greater than $1200:

$\{< l > \mid \exists\, b, a\, (< l, b, a > \in loan \wedge a > 1200)\}$

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 $\exists\, s$ for some tuple variable *s*, we bind it immediately to a relation by writing $\exists\, s \in r$. However, when we write $\exists\, b$ 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 subformula $< l, b, a > \in loan$ constrains *b* to branch names that appear in the *loan* relation.

SQL

- SQL uses a combination of relation algebra and relational calculus constructs.
- It includes features for defining the structures of the data for defining the structure of the data, for the structure of the data, for modifying data in the database, and for specifying security constraints.

Background:

SQL, standard relational database language.

- There are numerous versions of SQL. The original version was developed at IBM's Sanjose research laboratory.
- This language, originally called 'sequel', was implemented as part of the system R project in the early 1970's
- The sequel language has evolved language since than, and its name has changed to SQL (Structured Query Language).

The SQL language has several parts.

DATA DEFINITION [LANGUAGE (DDL):]

The SQL DDL provides commands for defining relation schemas, deleting relations, creation indices and modifying relation schemas.

Interactive Data Manipulation Language (DML):

The SQL DML includes a query language based on both the relational algebra and the tuple relational calculus.

Embedded DML:

The embedded form of SQL is designed for use within general purpose programming languages, such as PL/I, COBOL, PASCAL, FORTRON, and C

View definition:

The SQL DDL includes commands for defining views.

Authorization:

The SQL DDL includes commands for specifying access rights to relations and views.

Integrity:

The SQL includes commands for specifying integrity constraints for specifying integrity constraints that the data stored in the database must satisfy.

Transaction control:

SQL includes commands for specifying the beginning and ending of transactions.

The following relation schemas:

- Branch-schema= (branch-name, branch-city.assets)
- Customer-schema= (customer-name, customer-street, customer-city)
- Loan-schema = (branch-name, loan-number, amount)
- Borrower-schema= (customer-name, loan-number)
- Account-schema = (branch-name, account-number, balance)
- Depositor-schema= (customer- name, account-number)

BASIC STRUCTURE OF SQL:

- A relational database consists of a collection of relations, each of which is assigned a unique name the basic structure of an SQL expressions consists of three
- **Clauses:** select, from, and where.

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

A typical SQL query has the form,
Select A_1, A2….A_n.
From r_1, r2….r_m.
Where P.

Each A_i represents an attribute, and each r_i a relation, p is a predicate. The relational-algebra expression.

π A_1, A2…A_n (σ p(r_1 x r_2 x…x r_m)).

The select clause:

The result of an SQL query is a relation.
To find, the names of all branches in the loan relation.

SQL> Select branch_name from loan;

The result is a relation consisting of a signal attribute with the heading branch-name.
If we want to force the elimination of duplicates, we insert the keyword "distinct" after "select".

SQL> select distinct branch_name from loan;

The select clause can also contain arithmetic expressions involving the operators +, -, *, and /, and operating on constant or attributes of tuple. For example, the query.

SQL> select branch_name, loan_number, amount *100 from loan;

The where clause:

To find all loan numbers for loans made at the Los Angels branch with loan amounts greater that $1200."This query can be written in SQL as

SQL> select lone_number from loan where branch_="Los Angel" and amount > 1200;

SQL uses the logical connectives and, or and not- rather than the mathematical symbols.
SQL includes a between comparison operator to simplify where clauses that specify that a value be less than or equal to some other value.
To find the loan-number of those loans with loan amount between $90000 and $10000,

We can use the between comparison to write,

SQL> select loan_number from loan where amount between 90000 and 100000;
Inside of:
SQL> select loan_number from loan where amount <= 100000 and amount >= 90000;

The from clause:

The 'from' clause by itself defines a Cartesian of the relation in the clause.
"For all customers who have a loan from the bank, find there names and loan numbers".

SQL> select distinct customer_name, borrower, loan_number from borrower, loan where borrower.loan_number = loan.loan_number;

The rename operation:

SQL provides a mechanism for renaming both relations and attributes. It uses the'as'clause, taking the form;

Old-name as new-name.

The 'as' clause appear in both the 'select' and 'form' clauses.

SQL> select distinct customer_number from borrower, loan where borrower.loan_number = loan.loan_number and branch_name="los angels";

The result of this query is a relation of this query is a relation with the following two attributes:

Customer_name, loan_number

Tuple variables:

A tuple variable in SQL must be associated with a particular relation. Tuple variables are defined in the 'form' clause via the use of the 'as' clause.

"For all customer who have a loan from the bank, find their names and loan numbers" as

SQL> select distinct customer_name, T.loan_number as T, loan as S where T.loan_number = S.loan_number;

Find the names of all branches that have assets greater than at least one branch located in Los Vegas, we can write the SQL expression,

SQL> select distinct T.branch_name from branch as T, branch as S where T.assets s.assets and S.branch.city = "losvegas";

String operations:

That must commonly used operation on string is pattern matching using the operator "like".

We describe patterns using two special characters:

- Percent (%): the character matches any substring.
- Underscore (_): the_ character matches any character.

Patterns are case sensitive; that is upper case characters do not match lower case characters, or vice versa.

Let us consider the following example:

- "Perry %" matches any string beginning with "perry".
- "% idge %" matches any string contain "idge" as a sub string,

for example,"perryridge","Rock ridge", "Mianus bridge", and "ridge way".

- "-----"matches any string of exactly three characters.
- "-----%" matches any string of at least three characters.

Patterns are expressed in SQL using the "like" comparison operator.

"Find the names of all customers whose street address includes the substring 'main' this query can be written as,

SQL> select customer_name from customer where customer_street like "%main%";

For pattern to include the special pattern characters (that is, %, and_) SQL allows the specification of an escape character.

Consider the following patterns, which use a backslash (\) as the escape character:

- Like “ab \ %” escape “\” matches all string beginning with “ab % cd”.
- Like”ab\\cd” escape”\” matches all string beginning with “ab\cd”.

SQL allows us to search for mismatches instead of matches by using ‘Not like’ comparison operator.

Ordering the display of tuple:

SQL offers the user some control over the order in which tuples in a relation are displayed.

The ‘orderby’clause causes the tuples in the result of a query to appear in sorted order.

To list in alphabetic order all customer who have a loan at the Los angels branch, we write,

SQL> select distinct customer_name from borrower, loan where barrower.loan_number = loan.loan_number and breanch_name =”los angels” order by customer_name;

If several loans have the same amount, we order them in ascending order by loan-number. We express this query in SQL as follows:

SQL> select * from loan order by close, loan_number asc;

Duplicates:

We can define the duplicate semantics of an SQL query using ‘multiset’ versions of the relational operators.

We define the multiset versions of several of the relation-algebra operators here. Given multiset relations r_1, and r_2.

1. If there are c_1 copies of tuple t_1 in r_1 and t_1 satisfies selection σ xxx, then there are $c_1$copies of $t_1$in $\sigma\ \theta\ (r_1)$.
2. For each copy of tuple t_1 in r_1, there is a copy of tuple $\pi_A\ (t_1)$ in $\pi_A(r_1)$, where $\pi_A(t_1)$ denotes the projection of the signal tuple t_1.
3. If there are c_1 copies of tuple t_1 in $r_1$and c_2 copies of tuple t_2 in r_2, there are c_1 x c_2 copies of the $t_1.t_2$ in r_1 x r_2.

SET OPERATIONS:

The SQL-92 operations union, intersect, and except operate on relations and except on relation and correspond to the relations and correspond to the relation-algebra operations $\cup, \cap$ and _.like union, intersection, and set difference in relational algebra, the relations participating in the operations must be compatible.

We will construct queries involving the union, intersect, and except operations of two sects: the set of all customers who have an account at the bank, which can be derived by

SQL> select customer_name from depositor;

And the set of customer who have a loan at the bank, which can be derived by

SQL> select customer_name from barrower;

The union operation:

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

SQL> (select customer_name from depositor) union (select customer_name from barrower);

The union operation automatically eliminates duplicates, unlike the select clause.
If we want to retain all duplicates, we must write "union all" in place of union.

SQL> (select customer_name from depositor) union all (select customer_name from borrower);

The intersect operation:

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

SQL> (select distinct customer_name from depositor) intersect (select distinct customer_name from borrower);

The intersect operation automatically eliminates duplicates.

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

SQL> (select customer_name from depositor) intersect all (select customer_name from borrower);

The except operation:

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

SQL> (select distinct customer_name from depositor) except (select customer_name from borrower);

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

SQL> (select customer_name from depositor) except all (select customer_name from borrower);

Aggregate functions:

Aggregate functions are functions that take a collection (a set or multiset) of values as input and offer five built-in aggregate functions:

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

The input to sum and avg must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as string, as well.

To find the average account balance at the Los Vegas branch", we write the query as follows

SQL> select avg (balance) from account where branch_name= "Los Vegas"

The result of this query is a relation with a signal attribute, containing a single row with a numerical value corresponding to the average balance at the Los Vegas branch.

"Find the average account balance at each branch" we write the query as follows:

SQL> select branch_name avg (balance) from account group by branch_name;

There are cases where we must eliminate duplicates, prior to computing an aggregate function.
If we do want to eliminate duplicates, we use the keyword 'distinct' in the aggregate expression.

"To find the number of depositor for each branch".

SQL> select branch_name, count (distinct customer_name) from depositor. account where depositor.account_number = account.account_number group by branch_name;

Conditions that applies to groups rather than to tuples.
For example, we might be interested in only those branches where the average account balance is more than $1200.
This condition does not apply to a signal tuple; rather, it applies to each group constructed by the group by clause.
To express such a query, we use the "having" clause of SQL.

SQL> select branch_name, avg (balance) from account group by branch_name having avg (balance) > 1200;

Find the average balance for all accounts.

SQL> select avg (balance) from account;

To find the number of tuples in the customer relation, we write

SQL> select count (*) from customer;

Find average balance for each customer who lives in Hamilton and has at least three accounts.

SQL> select depositor.customer_name, avg (balance) from depositor, account, customer where depositor.account_number = account.account_number and depositor.customer_name = customer.customer_name and customer.city = "hamilton") group by depositor. Customer_name having count (distinct depositor.account_number) >=3;

Null values:

SQL allows the use of null values of indicate absence of information about the value of an attribute.
We can use the special keyword "null" in a predicate to test for a null value.

Thus, to find all loan numbers that appears in the loan relation, with null values for amount, we write,

SQL> select loan_number from loan where account is null;

The predicate is "not null" tests for the absence of a null value.

NESTED SUB QUERIES

SQL provides a mechanism for the nesting of sub queries.

A sub query is a select –from-where expression that is nested with in another query.

Common use of sub queries is to perform tests for set membership, set comparisons, and cardinality.

Set membership:

SQL draws of the relational calculus for operations that allow testing tuples for membership in a relation.

The 'in' connective tests for set membership, where the set is a collection of values product by a **'select'** clause.

To find all account holders the sub query.

SQL> select customer_name from depositor;

We then need to find those customers who are borrowers from the bank and who appear in the list of account holders obtained in the sub query in an outer 'select'.

The resulting query is,

SQL> select distinct customer_name from borrower where customer name in (select customer_name from depositor);

To find all customers do have a loan at the bank, but do not have an account at the bank, we can write.

SQL> select distinct customer_name from borrower where customer_name not in (Select customer_name from depositor);

The **'in'** and **'not in'** operators can also be on enumerated sets.

SQL> select distinct customer_name from borrower where customer_name not in ("smith", "Jones");

Set comparison:

Find the names of all branches that have assets greater than those of at least one branch located in 'Salem'.

SQL> select distinct T.branch_name from branch as T, branch as S where T.assets > S.assets and S.branch_city = "Salem";

The phrase 'greater than at least one" is represented in SQL by>some.

SQL> select branch_name from branch where assets > some (select assets from branch where branch_city = "Salem") the sub query, (select assets from branch where branch_city = "Salem");

Generates the set of all values for all branches in Salem.

The > some comparison in the 'where' clause of the outer select is true if the assets value of the tuple is greater than at least one number of the set of all asset values for branches in Salem.

SQL also allows <some, <=some, >=some, =some and<>some comparison.

The construct >all corresponds to the phrase" greater than all". Using this construct, we write the query as follows:

SQL> select branch_name from branch where assets > all (select assets from branch where branch_city = "salem");

SQL allows <all, <all, >=all, =all, and <>all comparisons.

To fine the branch that has the highest average balance. Aggregate functions cannot be composed in SQL.

SQL> select branch_name from account group by branch_name having avg (balance) >= all (select avg (balance) from account group by branch_name);

Test for empty relations:

SQL includes a feature for testing whether a sub query has any tuples in its result.

The exists constructs returns the value true if the argument sub query is non-empty.

"Find all customers who have both an account and a loan at the bank".

SQL> select customer_name from borrower where exists (select * from depositor where depositor.customer_name = borrower.customer_name);

We can test the non –existent of tuples in a sub query by using the not-exists constructs.

Test for the Absence of duplicate tuples:

SQL includes a feature for testing whether a sub query has any duplicate tuples in its result. The unique construct returns the value 'true' if the argument sub query contains no duplicate tuples.

Find all customers who have only one account at the Salem branch.

SQL> select T.customer_name from depositor as T where unique (select R.custsomer_name from account, depositor as all where T.customer_name = R.customer_name and R.account_number = account.account_number and account.branch_name = "salem");

Derived relations:

SQL-92 allows expressions to be used in the from clause. If such an expression is used then the result relation must be given a name, and the attributes can be renamed. Consider a sub query.

SQL> select branch_name, avg (balance) from depositor group by branch_name) as result (branch_name, avg_balance);

VIEWS:

We define a view in SQL using the "create view" command. The form of the create view command is

Create view Vas <query-expression> is any legal query expression. The view name is represent by V.

SQL> create view all_customer as (select branch_name, customer_name from depositor, account where depositor.account_number = account.account_number) union (select branch_name, customer_name from borrower, loan where borrower.loan_number = loan.loan_number);

MODIFICATION OF THE DATA BASE:

Deletion:

A delete request is expressed in much the same way as a query.
We can delete only whole tuples; we cannot delete values on only particular attributes.
In SQL a deletion is expressed by

SQL> delete from r where P;

Where P represents a predicate and 'r' represents a relation.
The result,

SQL> delete from loan.delete all tuples from the 'loan' relation;
Delete all tuples from the 'loan' relation.

Here are examples of SQL delete requests.
Delete all of 'smith's' account record.

SQL> delete from depositor where customer_name ="smith";

Delete all loans with loan amounts between $1300 and $ 1500.

SQL> delete from loan where amount between 1300 and 1500;

Insertion:

The simplest 'inset' statement is a request to insert one tuple. If we wish to insert the fact that there is an account A-9732 at the 'Salem' branch and that is has a balance of $1200. We write.

SQL> insert into account values ("salem", "A-9732", 1200);

If some insertion were carried out even as the select statement were being evaluated, a request such as,

SQL> insert into account select * from account;

Updates:

In certain situations, we may with to change a value in a tuple without changing all values in the tuple.

For this purpose, the update statement can be used. As we could for insert and delete, we can allow the tuples to be updated using a query.

Suppose that annul insert payments are being made, and all balance are to be increased by 5 percent write,

SQL> update account set balance = balance * 1.05;

The preceding statement is applied once to each tuple in account.

Suppose that account with balances over $10000 receive 6 percent interest, where as all others receive 5 percent. We write twp update statements:

SQL> update account set balance = balance * 1.06 where balance >10000;

SQL> update account set balance = balance * 1.05 where balance <= 10000;

Update of a view:

View-update exists also in SQL.

SQL> create view branch_loan as select branch_name, loan_number from loan;

We can write

SQL> insert into branch_loan values ("salem","L-307");

SQL-based database system imposes the followed constraint on modification allowed through views:

A modification is permitted through a view only if the view in question is defined in terms of one relation of the actual relation data base that is, of the logical-level database.

Joined relations:

SQL also provide various other mechanisms for joining relation, including conditions joins, and natural joins as well as various forms of outer joins.

Examples:

Branch_name	Loan_number	AMOUNT
Downtown	L-170	3000
Redwood	L-230	4000
perryvidge	L-260	1700

Loan

Customer_name	Loan_number
Joues	L-170
Smith	L-230
Hayes	L-155

Borrower

Branch_name	Loan_number	Amount	Customer_name	Loan_number
Downtown	L-170	3000	Joues	L-170
Redwood	L-230	4000	Smith	L-230

Result of loan inner join borrower on

SQL>loan.loan_number = borrower.loan_number.resul of the expression, loan inner join borrower on loan.loan_number = borrower.loan_number;

We rename the result relation of a join and the attributes of the result relation using an 'as' clause,

SQL> loan inner join borrower on loan.loan_number = borrower. loan number as lb (branch, loan_number, amount cust, cust_loan_num);

We consider an example of the use of the left outer join operation:

SQL>loan left outer join borrower on loan.loan_number = borrower.loan_number

The tuple (down town, L-170, 3000) and (red wood,, L-230, 4000) are joined with tuple from borrower and appear in the result of the inner join, and hence in the result of the left outer join.

On the other hand, the tuple (Perry ridge, L-260, 1700) did not match any tuple from borrower in the inner join, and hence a tuple (perry ridge, L-260, 1700, null, null) in present in the result of the left outer join.

Branch name	Loan. number	Amount	Customer name	Loan number
Down town	L-170	3000	Jones	L-170
Red wood	L-230	4000	Smith	L-230
Perry ridge	L-260	1700	Null	Null

Result of loan left outer join borrower on

SQL>Loan. loan_number = borrower.loan_number.

Finally, we consider an example of the use of the natural join operation.

SQL>loan natural inner join borrower

Join types and conditions:

The join condition defines which tuples in the two relations match and what attributes are present in the result of the join.

Join condition
Inner join
Left outer join
Right outer join
Full outer join

Join condition
Natural
On <predicate>
Using (A_1, A_2,A_3,…A_n)

Data definition language:

- The SQL DDL allows the specification of not only a set of relations, but also information about each relation, including.
- The schema for each relation.
- The domain of values associated which each attribute
- The integrity constraints
- The set of indices to be maintained for each relation
- The security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domain types in SQL:

The SQL-92 standard supports a Varity of built-in domain types, including the following:

- Char (n) is a fixed- length character string, with user-specified length n.the full form, character, can be used instead.
- Varchar (n) is a variable-length character string, with user-specified maximum length n. the full form, character varying, is equivalent.
- Int is an integer. The full form, integer, is equivalent.
- Smallint is a small integer.
- Numeric (p, d) is a fixed-point numbers, with user-specified precision. Thus, numeric
- (3, 1) allows 44.5 to be stored exactly. Real-double precisions are following point and double-precision floating- point numbers, with machine-dependent precision.
- Float (n) is a floating-point numbers, with user-specified precision of at least n digits.
- Date is a calendar date, containing a (four digit) year, month and day of the month.
- Time is the time of day, in hours minutes, and seconds.

Ex:

SQL> create domain person-name char (20).

Schema definition in SQL:

We define an SQL relation using the create table command:

SQL> create table r (A_1D_1, A_2D_2… AnDn, <integrity- constraint$_1$>… <Integrity-constraint>).

Where r is the name of the relational each A_i is the name of an attribute in the schema of relation r, and D_i is the domain type of values in the domain of attribute A_i.

The allowed integrity constraints include,

Primary key (A_{j1}, A_{j2} ….A_{jm})

Chick (p).

The 'primary key' specification says that attributes A_{j1}, A_{j2}…A_{jm} from the primary key for the relation.

SQL>create table customer (customer-name char (20) not null, customer-street char (30), customer-city char (30),primary key (customer-name));

SQL>create table branch (branch-name char (15) not null, branch-city char (30), assets integer, primary key (branch-name), check (assets>=0));

To remove a relation from an SQL database, we use the drop table command. the drop table command deletes all information about the dropped relation from the database. The command.

SQL>drop table r

Is a more drastic action than

SQL> delete from r.

We use the alter table command in SQL to add attributes to an existing relation. All tuples in the relation are assigned null attribute null as the new attribute. The form of the attribute. The form of all the alter table command is

SQL> alter table r add AD.

Where r is the name of an existing relation, A is the name of the attribute to be added and D is the domain of the added attribute. We can drop attributes. From a relation using a command.

SQL>alter table r drop A.

Where r is the name of an attribute of the relation, and is the name of an attribute of the relation.

QUESTION BANK

UNIT – II

5 Marks:

1. Write about Relational algebra.
2. Explain about Set operation.
3. Explain about aggregate functions.
4. Explain nested sub queries.
5. Explain DDL.

10 Marks:

6. Write in detail about basic structure of SQL.
7. Describe modification of the data base.

UNIT-III

DATA NORMALIZATION

INTRODUCTION:

Normalization is the formal process for deciding which attributes should be grouped together in a relation. We can use commonsense to decide which fields or attributes should be grouped together, but normalization provides us with a systematic and scientific process for doing this.

PITFALLS IN RELATIONAL DATABASE:

- ✓ Storing database in relational tables can result in many problems if the database is not designed properly. We use normalization and various other techniques to make the database designs more efficient and proper.
- ✓ Two most problematic issues in the design of relational databases are,
  - ✷ Repitition of information (redundancy)
  - ✷ Inability to represent certain information

Example:

```
CREATE TABLE BOOKS
(TITLE VARCHAR(50) NOT NULL,
PUBLISHER VARCHAR(50) NOT NULL,
PUBLISHER-CITY VARCHAR(50) NOT NULL,
AUTHOR VARCHAR(50) NOT NULL,
AUTHOR-CITY VARCHAR(50));
```

The contents of the table are given below:

Title	Publisher	Publisher-city	Author	Author-city
A guide to SCM	Artech House	Boston	Alexis	Cochin
ERP Demystified	Tata Mc-Graw hill	New Delhi	Alexis	Cochin
MS-word 2000	Tata Mc-Graw hill	New Delhi	Mathews	Chennai
Internet Security	Artech House	Boston	David	Newyork

REPITITION OF INFORMATION:

- ✓ Suppose we wish to add a new book to the table, the title of the book is Ergonomics, published by Artech House and written by Alexis. So we will add the following tuple to the table:
  (Ergonomics, Artech House, Boston, Alexis, Cochin)
  Note that the publisher and author information are repeated.
- ✓ This repetition of information is undesirable and it is waste of space. It also complicated modifications.
- ✓ If an author has moved to another city, then the Author-city field should be changed in all rows where the author appears. Ina large table where there are 1000's of tuples, the may not get incorporated in all rows and will result in corrupted data and databases with no data integrity.

INABILITY TO REPRESENT CERTAIN INFORMATION:

- ✓ A publisher has just started the operations, but no book has been released. In this case, we cannot add the publisher details to the table as all fields should have values in it. This problem can be solved by using null values is that they are difficult to handle and can create problems for the inexperienced users.

SOME OTHER PITFALLS ARE:

Spreadsheet design, Too much data, Compound fields, Missing keys, Bad keys, Missing relations, Unnecessary relationships, Incorrect relation, Duplicate field names, cryptic field and table names, Missing or incorrect business rules, Missing or incorrect constraints, Referential integrity, Database Security and International Issues.

DECOMPOSITION:

The problems created by the bad design of relations suggest that we should decompose a relation schema that has several attributes into several schemas with fewer attributes. But careless decomposition or decomposing without any valid reason can result in another bad design.

Disadvantages of decomposition:

If you want to find the title of the books published by a particular publisher, You need to reconstruct the book relation. But this can be done by performing a join operation on the PUBLISHER and AUTHOR relations.

LOSSY-JOIN DECOMPOSITION:

- ➢ We have loss information in a database while joining is called lossy-join decomposition.
- ➢ If an author happens to have several books with more than one publisher, we cannot tell which book belongs to which publisher.

Publisher	P-City	S-Editor	Author	Title	Price
Artech	London	Tim	Alexis	SCM	4000
THM	Delhi	Chandra	Alexis	ERP	395
VNI	Chennai	Madhavan	Alexis	EDBMS	250
TMH	Delhi	Chandra	Mathews	E-Biz	300
Artech	London	Tim	Alexis	Guide to SCM	3500

If an author happens to have several books with more than one publisher, we cannot tell which book belongs to which publisher.

LOSSLESS-JOIN DECOMPOSITION:

- A decomposition that is not a lossy-join decomposition is called a lossless-join decomposition
- The lossy-join decomposition is a case of bad database design.

PROPERTIES OF DECOMPOSITION:

The desirable properties of decomposition are :

Attribute preservation:

- This is a simple and an obvious requirement that involves preserving all that attributes that were there in the relation that is being decomposed.

Lossless-Join Decomposition:

- We decomposed a relation intuitively. If relations are decomposed carelessly it can lead to many problems including loss of information. We need a better bases for deciding decompositions since intuition may not always be correct. Lossless-join decomposition guarantees that the join will result in exactly the same relation as was decomposed.
- Let R be a relation schema and F be a set of functional dependencies on R. Let R1 and R2 form a decomposition of R. This decomposition is a lossless-join decomposition of R if atleast one of the following functional dependencies is in

  F: $R1 \cap R2 \rightarrow R1$ and $R1 \cap R2 \rightarrow R2$

Dependency Preservation:

- When an update is made to the database, the system should be able to check that the update does not create an illegal relation (ie) a relation that does not satisfy all the given functional dependencies.
- If we are to check updates efficiently, we should design relational database schemas that allow update validation without the computation of joins.

Lack of redundancy:

- We have discussed the problems of repetition of information in database.
- Such repetitions should be avoided

FUNCTIONAL DEPENDENCIES:

- Functional dependencies are constraints on the set of legal relations. They allow the database designers to express facts about the real world entity which is being modeled and designed as the database.
- Functional dependency is a property of the meaning or semantics of the attributes in a relation. The semantics indicate how attributes relate to one another and specify the functional dependencies between attributes when a functional dependency is present, the dependency is specified as a constraint between the attributes.

  For example,

  1. A and B are attributes of the relation R, B is functionally dependent on A (represented as $A \rightarrow B$), if each value of A is associated with exactly one value of B.
  2. Emp-no and emp-name attributes the following functional dependencies hold:

     Emp-no→emp-name

     Emp-name→emp-no

Rules for Functional Dependencies:

Reflexivity : If B is a subset of A, $A \rightarrow B$

Augmentation : If $A \rightarrow B$, then $A,C \rightarrow B,C$

Transitivity : If $A \rightarrow B$ & $B \rightarrow C$, then $A \rightarrow C$

Self-determination : $A \rightarrow A$

Decomposition : If $A \rightarrow B,C$ then $A \rightarrow B$ & $A \rightarrow C$

Union : If $A \rightarrow B$ & $A \rightarrow C$, then $A \rightarrow B,C$

Composition : If $A \rightarrow B$ & $C \rightarrow D$ then $A,C \rightarrow B,D$

NORMALIZATION

Normalization is a formal process of developing data structures in a manner that eliminates redundancy and promotes integrity. Data normalization is a corner stone of the relational theory.

There are many steps to Data Normalization. Each normal form builds upon the last and functions as both a process and a criterion.

For example,

A database structure that can be developed to the third normal form can be said to satisfy the second normal form.

NORMAL FORMS:

There are six normal forms exists.

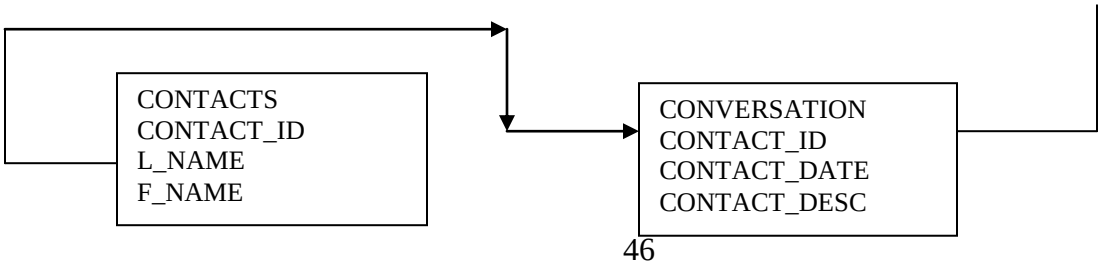
- ⇒ First normal form (1 NF)
  The multivalued attributes should be removed. (ie) Elimination of repeating groups.
- ⇒ Second normal form (2 NF)
  The partial functional dependencies have to be removed. (ie) Elimination of redundant data.
- ⇒ Third normal form (3 NF)
  The transitive dependencies have to be removed. (ie) Elimination of columns not dependent on the key.
- ⇒ Boyce-codd normal form (BCNF)
  The remaining anamolies that result from functions dependencies are removed.
- ⇒ Fourth normal form (4 NF)
  Multi-valued dependencies are removed (ie) isolation of independent multiple relationship.
- ⇒ Fifth normal form (5 NF)
  Any remaining anamolies are removed. In this normal form we isolate semantically related multiple relationships.

FIRST NORMAL FORM (1 NF):

- ➢ First Normal form (1NF) is a relation in which the intersection of each row and column contains one and only one value.
- ➢ To transform the unnormalized table (a table that contains one or more repeating groups) to first normal form, we identify and remove the repeating groups without the table.
- ➢ A repeating group is a set of columns that store similar information that repeats in the same table.

Example,

```
CREATE TABLE CONTACTS
( CONTACT-ID INTEGER NOT NULL,
L_NAME VARCHAR(20) NOT NULL,
F_NAME VARCHAR(20),
CONTACT_DATE1 DATE,
CONTACT_DESC1 VARCHAR(20),
CONTACT_DATE2 DATE,
CONTACT_DESC2 VARCHAR(50));
```



Advantages:

- The only advantage or designing the table like this is that it avoids the need for a relationships.

Disadvantages:

- The above data structure contains a repeating group of the date and description of two conversations
- This structure limits the number of conversations to two, conversations need to be stored.
- This structure also makes it difficult to do any kind of meaningful searching using the columns

For example,

- To locate a conversation on a specific date (here date columns have to be searched which will result in a clumsy SQL code)
- To eliminate the repeating group, the group is moved to another table, which is then related to the parent table. The primary key of the parent table (CONTACT_ID) is stored in the second table. Moving the repeating group into another table allows any number of conversations to be recorded and searched easily.
- The DDL statement for crating the table is

```
CREATE TABLE CONTACTS
( CONTACT-ID INTEGER NOT NULL,
L_NAME VARCHAR(20) NOT NULL,
F_NAME VARCHAR(20));

CREATE TABLE CONVERSATION
(CONTACT_ID INTEGER NOT NULL,
CONTACT_DATE DATE,
CONTACT_DESC VARCHAR(50));
```

Conclusion:

- Every table should have a primary key, and each set of repeating groups should appear in its own table.
- When these criteria are satisfied, we say that the first normal form is achieved.

SECOND NORMAL FORM (2NF):

- Second normal form is based on the concept of full functional dependency.
- A relation is in second normal form if it is in first normal form and every non-primary key attribute is fully and functionally dependent on the primary key.
- Thus no non-key attribute is functionally dependent on the primary key.
- A relation in the first normal form will be in the second normal form if one of the following conditions is satisfied:
  - The primary key consists of only one attribute (field or column)
  - No non-key attributes exist in the relation
  - Every non-key attribute is functionally dependent on the full set of primary key attributes.

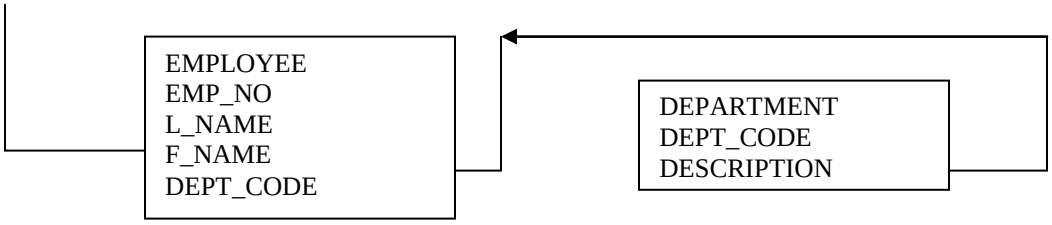
Example:

```
CREATE TABLE EMPLOYEE
(EMP_NO INTEGER NOT NULL,
L_NAME VARCHAR(20) NOT NULL,
F_NAME VARCHAR(20),
DEPT_CODE INTEGER,
DESCRIPTION VARCHAR(50));
```

- In this table each attribute depend on 2 primary key attribute such as Emp_no, Dept_code.
- It is also redundancy.
- So by storing the department code and description in a different table the redundancy is eliminated.

```
CREATE TABLE EMPLOYEE
(EMP_NO INTEGER NOT NULL,
L_NAME VARCHAR(20) NOT NULL,
F_NAME VARCHAR(20),
DEPT_CODE INTEGER));

CREATE TABLE DEPARTMENT
(DEPT_CODE INTEGER NOT NULL,
DESCRIPTION VARCHAR(50) NOT NULL);
```



EMPLOYEE DEPARTMENT Relationship

THIRD NORMAL FORM (3NF):

- A relation is in third normal form (3NF) if it is in second normal form and no transitive dependencies exit.
- Transitive dependency is a condition where A, B & C are attributes such that if A→B and B→C, Then C is transitively dependent on A via B.
- Transitive dependency is a type of functional dependency.

The function dependencies:

Emp_no→department
department→dept_head

The transitive dependency emp_no→dept_head exists via the department attribute. This condition holds as emp_no is not functionally dependent on department or dept_head.

Example:

```
CREATE TABLE CONTACTS
(CONTACT_ID INTEGER NOT NULL,
L_NAME VARCHAR(20) NOT NULL,
F_NAME VARCHAR(20),
COMPANY_NAME VARCHAR(20),
COMPANY_LOCATION VARCHAR(50));
```

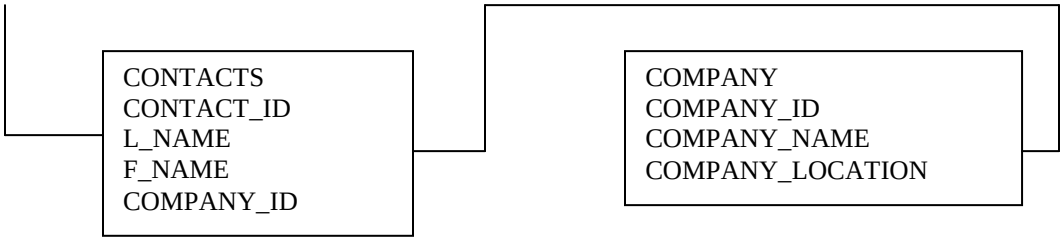
- In the above relation CONTACT_ID is the primary key, so that all the remaining attributes are functionally dependent on this attribute
- However, there is a transitive dependency COMPANY_LOCATION is dependent on COMPANY_NAME and COMPANY_NAME is functionally dependent on CONTACT_ID
- So unless the location of the company differs on an individual basis, this column is not dependent on the key value and should be removed to another table
- Here one thing that should be noted is that, as a result of the transitive dependency, there are update anomalies in the CONTACTS table as follows:

- **Insertion Anomaly:**
  A new company cannot be inserted to the CONTACTS table until a contact person has been assigned to that company.
- **Deletion Anomaly:**
  If a company that has only one contact person is deleted from the table, we will lose the information about that company, as the company information is associated with that person.
- **Modification Anomaly:**
  If a company changes its location, we will have to make the change in all the records wherever the company name appears. Suppose, if the company has five contact persons, then we will have to make the changes in five places.

✷ The insertion, deletion, and modification anomalies are arise as a result of the transitive dependency. The transitive dependency can be removed by decomposition.

```
CREATE TABLE CONTACTS
(CONTACT_ID INTEGER NOT NULL,
L_NAME VARCHAR(20) NOT NULL,
F_NAME VARCHAR(20),
COMPANY_ID INTEGER);

CREATE TABLE COMPANY
(COMPANY_ID INTEGER NOT NULL,
COMPANY_NAME VARCHAR(20) NOT NULL,
COMPANY_LOCATION VARCHAR(50));
```



Conclusion:

When all the columns in a table describe and depend upon the primary key. The table is said to satisfy the third normal form.

BOYCEE-CODD NORMAL FORM (BCNF):

✷ Database relations are designed so that they have neither partial dependencies nor transitive dependencies because these types of dependencies result in update anomalies.
✷ A functional dependency describe the relationship between attributes in a relation.
✷ For example,
✷ If A and B are attributes in relation R, B is functionally dependent on A (denoted by A→B), if each value of A is associated with exactly one value of B.
✷ In the CONTACTS table, we can say that L_NAME, F_NAME and COMPANY_ID are functionally dependent on CONTANCT_ID.

These dependencies are expressed as follows:

CONTACT_ID → L_NAME
CONTACT_ID →F_NAME
CONTACT_ID →COMPANY_ID
CONTACT_ID→{L_NAME, F_NAME, COMPANY_ID}
{L_NAME, F_NAME, COMPANY_ID} → CONTACT_ID

- The left hand side and right-hand side of a functional dependency are sometimes called the determinant and dependent.
- As the definition states, the determinant and the dependent are both, sets of attributes.
- 3NF, BCNF is equivalent when in relation have only one key.

FOURTH NORMAL FORM (4NF):

- A group of tables that satisfies the first, second and third normal forms are sufficiently well-designed
- However, isolating independent multiple relationships will further improve the data model when one one-to-many and many-to-many relationships between tables are involved
- In other words, no table should contain two or more one-to-many or many-to-many relationships that are not directly related to the key.
- These kinds of relationships are called multi-valued dependencies (MVDS).

Multi-valued Dependency:

- Multi-valued dependencies are the result of the 1NF, which prohibited an attribute from having a set of values.
- If we have two or more multi-valued independent attributes in the same relation (table), we get into a situation where we have to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain independence among the attributes involved. This constraint is specified by a Multi-valued dependency.

Example:

A Multi-valued dependency represents a dependency between attributes A, B and C in a relation, such that for each value A. there is a set of values for B and set of values for C. However, the set of values for B and C are independent of each other. We represent a MVD between the attributes A,B and C in a relation as follows:

A ≫ B
A ≫ C

Another Example:

- Consider a table EMPLOYEE that has the attributes Name, Project and Hobby.
- A row in the EMPLOYEE table represents the fact that an employee works for a project and has a hobby. But an employee can work in more than one project and can have more one hobby.
- The employee's projects and hobbies are independent of one another. To keep the relation state consistent, we must have a separate tuple to represent every combination of an employee's project and an employee's hobbies.
- This constraint is specified as a multi-valued dependency on the EMPLOYEE relation. So, whenever two independent one-to-many relationships (A:B and A:C) are mixed in the same relation, a multi-valued dependency arises. We will see the employee table and how the multi-valued dependency can be avoided using the fourth normal form.

NAME	PROJECT	HOBBY
Alexis	Microsoft	Reading
Alexis	Oracle	Music
Alexis	Microsoft	Music
Alexis	Oracle	Reading
Mathews	Intel	Movies
Mathews	Sybase	Riding
Mathews	Intel	Riding
Mathews	Sybase	Movies

EMPLOYEE TABLE

- The above relation has two-multi valued dependencies (name, project) and (name, hobby)
- Decomposing the EMPLOYEE table into two tables that satisfy the 4NF as follows:

Project

Name	Project
Alexis	Microsoft
Alexis	Oracle
Mathews	Intel
Mathews	Sybase

Hobby

Name	Hobby
Alexis	Reading
Alexis	Music
Mathews	Movies
Mathews	Riding

FIFTH NORMAL FORM (5NF):

- The chances that you will ever get to use the 5NF are very few, because it requires semantically related multiple relationships, which are rare.
- Semantically relates multiple relationships are two or more relationships among tables that are related closely enough. So that they can be resolved into a single relationship.
- 5NF specifies that they remain separate.
- 5NF is a relation that has no join dependency.
- Join dependency describes a type of dependency. In a relation R with subsets of the attributes of R demotes as A,B, ……Z. a relation R satisfies a join dependency if and only if , every legal value of R is equal to the join of its projections on A, B, …..Z.

```
CREATE TABLE LAB-PRODUCT-COMPANY
(LAB_ID  INTEGER NOT NULL,
PRODUCT_ID INTEGER NOT NULL,
COMPANY_ID INTEGER NOT NULL);
```

- The table contains three foreign keys expressing two relationships:
- The relationship between LABS and PRODUCTS and that between LABS and COMPANIES. The relationships are semantically related because they can be expressed using the same table.
- There is nothing wrong with the above data structure, but it does not satisfy the 5NF and more than necessary entries are required in the table if the relationships are separated. Because a lab can test the same product for all companies who offer a given product, the following structure will be better:

```
CREATE TABLE LAB_PRODUCT
(LAB_ID INTEGER NOT NULL,
PRODUCT_ID INTEGER NOT NULL);

CREATE TABLE LAB_COMPANY
(LAB_ID INTEGER NOT NULL,
COMPANY_ID INTEGER NOT NULL);
```

DENORMALIZATION:

Disadvantage from Normalization:

- ✓ In the real world, with live data, demanding users and real demands on performance and ease of use, this flexibility is fundamental to success. Normalization is analysis, not design.
- ✓ Design encompasses issues, particularly related to performance, ease of use, maintenance, and straightforward completion of business tasks, things which are unaccounted for in normalization.

DENORMALIZATION:

- ✓ Denormalization is the opposite of normalization
- ✓ It is the process of increasing redundancy in the database either for convenience or to improve performance

DATABASE SECURITY

INTRODUCTION

- ✓ Security is an important issue in database management because information stored in a database is very valuable and many a time, very sensitive commodity.
- ✓ So, the data in a database management system need to be protected from abuse they should be protected from unauthorized access and updates
- ✓ Database security involves allowing or disallowing users from performing actions on the database and the objects within it
- ✓ All database management systems provide comprehensive discretionary access control
- ✓ Discretionary access control regulates all user access to named objects through privileges
- ✓ A privilege is permission to access a named object in a prescribed manner
- ✓ Example permission to query a table. As privileges are granted to users as the discretion of other users, this is called discretionary security.
- ✓ Security also requires access control, data integrity, system availability and auditing
- ✓ Many people strongly believe that firewalls make the data secure. But statistics show that more than 40% of internet break-ins occur in spite of a firewall being in place
- ✓ To design a security solution that will truly protect your data, you must understand the security requirements relevant to your site and the scope of current threats to your data

DATA SECURITY REQUIREMENTS:

- ✓ We should use technology to ensure a secure computing environment of the organization. Although it is not possible to find a technological solution for all problems, most of the security issued could be resolved using appropriate technology
- ✓ The basic security standards which technology can ensure are confidentiality, integrity and availability

CONFIDENTIALITY:

- ✓ A secure system ensures the confidentiality of data. This means that it allows individuals to see only the data that they are supposed to see
- ✓ Confidentiality has several aspects like privacy of communications, secure storage of sensitive data, authenticated users and granular access control etc as described below

Privacy of communication:

- The DBMS should be capable of controlling the spread of confidential personal information such as health, employment and credit records. It should also keep the corporate data such as trade secrets, proprietary information about products and processes, competitive analyses, as well as marketing and sales plans secure and away from the unauthorized people

Secure Storage of Sensitive Data:

- How can you ensure that data remains private, once it has been collected? Once confidential data has been entered, its integrity and privacy must be protected on the databases and servers wherein it resides

Authenticated Users:

- How can you designate the persons and organizations who have the right to see data?
- Authentication is a way of implementing decisions about whom to trust
- Authentication methods seek to guarantee the identity of system users that a person is who he says he is and not an impostor

Granular Access Control:

- Access control is the ability to hide portions of the database. So that access to the data does not become and all or nothing proposition

  Example:
  - A clerk in the HR department, might need some access to the EMPLOYEE table. But he should not be permitted to access salary information for the entire company
  - The granularity of access control is the degree to which data access can be differentiated for particular tables, views, rows and columns of a database.

Integrity:

- A secure system ensures that the data it contains is valid
- Data integrity means that data is protected from deletion and corruption, both while it resides within the database and while it is being transmitted over the network
- **<u>Integrity has several aspects:</u>**
  1. System and object privileges access to application tables and system commands, so that only authorized users can change data
  2. Referential integrity is the ability to maintain valid relationships between values in the database according to rules that have been defined
  3. A database must be protected against viruses designed to corrupt the data
  4. The network traffic must be protected from deletion, corruption and eavesdropping

Availability:

- A secure system makes data available to authorized users, without delay
- Denial-of-service attacks are attempts to block authorized users ability to access and use the system when needed
- System availability has a number of aspects

Resistance:

- The system can be protected against users consuming too much memory or too many processes, thus preventing others from doing their work

Scalability:

- System performance must remain adequate regardless of the number of users or processes demanding service.

Flexibility:

- Administrators must have adequate means of managing the user population. They might do this by using a directory.

Ease of use:

- The security implementation itself must not diminish the ability of valid users to get their work done.

PROTECTING THE DATA WITHIN THE DATABASE:

- Confidentiality, Integrity And Availability are the hallmarks of database security.
- Authorization is permission given to a user, process to access an object or set of objects.
- The type of data access granted to a user can be read only or read and write, privileges specify the type of data manipulation language (DML) operations like SELECT, INSERT, UPDATE, DELETE etc.. which the user can perform upon data.
- The two methods by which the access control is done are by using privileges and roles.
- A privilege is a permission to access a named object in a prescribed manner.
  For Example,
  Permission to query a table.

DATABASE PRIVILEGES:

- A privilege is a right to execute a particular type of SQL statement or to access another user's object.
- Some examples of privileges include:
  - The right to connect to the database
  - The right to create a table
  - The right to select rows from another user's table
  - The right to execute another user's stored procedure
- Privileges are granted to users so that these can accomplish tasks required for their job. You should grant a privilege only to a user who absolutely requires the privilege to accomplish necessary work
- Excessive granting of unnecessary privileges can lead to compromised security. A user can receive a privilege in to two different ways
- We can grant privileges to users explicitly
  For example,
  Explicitly grant the privilege to insert records into the Emp table to the user Alexis.
- We can also grant privileges to a role and then grant the role to one or more users.
  For example,
  Grant the privileges to select, insert, update and delete records from the Emp table to the role named ACCOUNTANT, which in turn you can grant to the users RAM and KUMAR.

- ❖ Because roles allow for easier and better management of privileges, you should normally grant privileges to roles and not to specific users.
- ❖ There are two distinct categories of privileges are,
    1. System privileges
    2. Object privileges

1. System Privileges:

- ♦ A system privilege is the right to perform a particular action or to perform a particular action on a particular type of object.
  Example,
    - The privileges to create tables and to delete the rows of any table in a database are system privileges.
    - System privileges are granted to or revoked from users and roles using the SQL commands GRANT & REVOKE.

2. Object privileges:

- ♦ An object privilege is a privilege or right to perform a particular action on a specific table, view, sequence, procedure, function or package.
- ♦ Object privileges granted for a table, view, sequence, procedure, function or package apply whether referencing the base object by name or using a synonym. If grant object privileges on a table, view, sequence, procedure, function or package to a synonym for the object, the effect is the same as if no synonym were used. If a synonym is dropped, all grants for underlying object remain in effect, even if the privileges were granted by specifying the dropped synonym.
- ♦ Object privileges can be granted to and revoked from users and roles.
- ♦ Object privileges can be granted to and revoked from users and roles using the SQL command GRANT and REVOKE respectively.

ROLES:

- ➢ Database management systems provide easy and controlled privilege management through roles. Roles are named groups of related privileges that you grant to users or other roles.
- ➢ Roles are designed to ease the administration of end user system and object privileges. However, application developers should not use roles in their applications. This is because the privileges to access objects within stored programmatic constructs need to be granted directly.
- ➢ The following properties of roles allow for easier privilege management within a database

Reduced privilege administration:

Rather than explicitly granting the same set of privileges to several users, you can grant the privileges for a group of related users to a role. Then only the role needs to be granted to each member of the group.

Dynamic privilege management:

If the privileges of a group must change, only the privileges of the role need to performed and modified. The security domains of all users who are granted the groups role will automatically reflect the changes to made to the role.

Selective availability of privileges:

Selectively enable or disable the roles granted to a user. This allows specific control of a users privileges in any given situation.

Application awareness:

Because the data dictionary records which roles exist, you can design database application to query the dictionary and automatically enable and disable selective roles when a user attempts to execute the application via a given username.

Application specific security:

- ✓ You can protect role use with a password, Applications can be created specifically to enable a role when supplied the correct password.
- ✓ Users cannot enable the role if they do not know the password.

USES FOR ROLES:

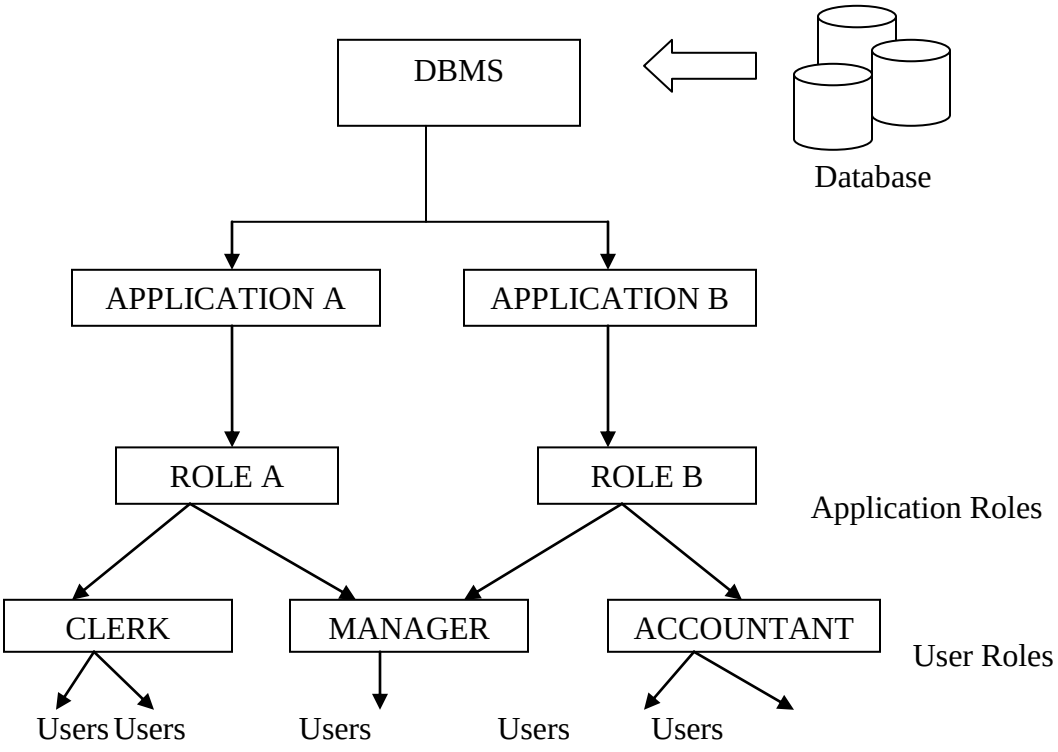
In general, you create a role to serve one of two purposes to manage the privileges for a database application or to manage the privileges for a database application or to manage the privileges for a user group.

APPLICATION ROLES:

Create an application role by granting all the privileges necessary to run a given database application. Then, you grant the application role to other roles or to specific users. An application can have several different roles. With each role assigned a different set of privileges that allow for greater or lesser data access with using the application.

USER ROLES:

Create user role for a group of database users with common privilege requirements. You manage user privileges by granting application roles and privileges to the user role and then granting the user role to appropriate users.



The Functionality of database role include the following:

- ✓ A role can be granted system or object privileges
- ✓ Any role can be granted to any database user
- ✓ A role can be granted to other roles. For example, role A cannot be granted to role B if role B has previously been granted role A.
- ✓ Each role granted to a user at a given time is either enabled or disabled. A User's security domain dies not includes the privileges of any roles currently disabled for the user. The DBMS allows database applications and users to enable and disable roles to provide selective availability of privileges.
- ✓ An indirectly granted role can be explicitly enabled or disabled for a user. However, by enabling a role that contains other roles, you implicitly enable all indirectly granted roles of the directly granted role.

GRANTING AND REVOKING PRIVILEGES AND ROLES

Grant or revoke privileges and roles from users or other roles using the SQL commands GRANT & REVOKE.

THE GRANT COMMAND:

- ✓ In most multiuser database management systems you need to be a special user DBA or owner or you need to get explicit permission from either the DBA or owner to perform any operation, even to run a simple query.
- ✓ The syntax of the GRANT command is given below:

```
GRANT{ALL/Privilege-list}
ON{Table-name[(column-comma-list)]/View-name[(column-comma-list)]}
TO{PUBLIC/user-list}
[WITH GRANT OPTION]
```

Examples,

- Grant the SELECT authority on the BOOK table to all users.
  GRANT SELECT ON BOOK TO PUBLIC;
- Grant the SELECT, DELETE and UPDATE authority on CATALOG table to user ALEXIS.
  GRANT SELECT, DELETE, UPDATE ON CATALOG TO ALEXIS;
- Grant the SELECT, DELETE and UPDATE authority with the capability to grant those privileges to other users on CATALOG table to user ALEXIS.
  GRANT SELECT, UPDATE ON CATALOG TO ALEXIS WITH GRANT OPTION;
- Grant all privileges on BOOK table to user MATHEWS.
  GRANT ALL ON BOOK TO MATHEWS;
- Give the system privileges for creating table and views to ALEXIS.
  GRANT CREATE TABLE, CREATE VIEW TO ALEXIS;
- Grant the UPDATE authority on the PRICE column of the CATALOG to user ALEXIS.
  GRANT UPDATE (PRICE) ON CATALOG TO ALEXIS;

THE REVOKE COMMAND:

- ✓ The REVOKE command as we have mentioned before is used to take away a privilege that was granted
- ✓ **Syntax:**

REVOKE{ALL/Privilege-list}
ON{table-name[(column-comma-list)]/view-name
[(column-comma-list)]}
FROM{PUBLIC/user-list}

Examples:

- REVOKE the system privileges for creating tables from ALEXIS
  REVOKE CREATE TABLE FROM ALEXIS.
- REVOKE the SELECT PRIVILEGE on CATALOG table from ALEXIS
  REVOKE SELECT ON CATALOG FROM ALEXIS;
- REVOKE the UPDATE privileges on CATALOG table from users
  REVOKE UPDATE ON CATALOG FROM PUBLIC;
- Remove ALL privileges on CATALOG tale from user MATHEWS
  REVOKE ALL ON CATALOG FROM MATHEWS;
- Remove DELETE & UPDATE authority on the PRICE and YEAR columns of the CATALOG table from user ALEXIS
  REVOKE DELETE, UPDATE (PRICE, YEAR) ON CATALOG FROM ALEXIS;

UNIT-III

QUESTION BANK

5 MARKS:

1. Write about the pitfalls in relational database.
2. Write the short notes on 1NF.
3. Explain about boyce-codd normal form.
4. Write about Database privileges.
5. How to protecting the data within the database?
6. Define roles. Explain about uses of roles.
7. Write the short note on User roles.

10 MARKS

1. Define Normalization. Write about any three types of normal forms.
2. Explain briefly about types of Roles.
3. Explain about GRANT & REVOKE command in databases.

UNIT – IV

A BRIEF HISTORY OF PL/SQL:

Before PL?SQL was developed users embedded SQL statements into hot languages like C++ and java. PL/SQL version was introduced with oracle 6.0 in 1991. Version 1.0 had very limited capacilities, however and was far from being a full-fleged programming language. It was merely used for batch processing.

With version 2.0, 2.1and 2.2 the following new features were introduced:

- ✓ The transaction control statements SAVEPOINT, ROLLBACK and commit.
- ✓ The DML statements INSERT, DELETE and update.
- ✓ The extended data types BOLLEAN_INTEGER, PL/SQL records and PL/SQL tables.
- ✓ Built-in-functions-character, numeric, and date functions.
- ✓ Built-in packages.
- ✓ The control structures sequences, selection and looping. A name should be the same as the name of a column used in the block.
- ✓ Database access through work area called cursors.
- ✓ Error handling.
- ✓ Modular programming with procedures and functions.
- ✓ Stored procedure, function and packages.
- ✓ Programmer-defined subtypes.
- ✓ DDL support through the DBMS_SQL package.
- ✓ The PL/SQL wrapper.
- ✓ The DBMS_JOB scheduler.
- ✓ File I/O with the UTF_FILE package.

FUNDAMENTALS OF PL/SQL

A PL/SQL program consist of statements. You may use upper or lower case letters in your program. In other words, PL/SQL is not case sensitive except for character string values enclosed in single quotes. Like any other programming language, Pl/SQL statements consist of reserved words, identifiers, delimiters, literals and comments.

Reserved words

The reserved words or key words are words provided by the language that have a specific use in the language. For example, DECLARE, BEGIN, END, IF, WHILE, EXCEPTION, PROCEDURE, FUNCTION, PACKAGE and trigger are some of the reserved words in PL/SQL.

User-Defined identifiers

User-defined identifiers are used to name variables, constants, procedure, functions, cursors, tables, records and exception. A user must obey the following rules in naming these identifiers:

- ✓ The name can be from 1 to 30 characters in length.
- ✓ The mane must start with a letter.
- ✓ Letters (A-Z, a-z), numbers, the dollar sign ($), number sign (#) and the underscore (_) are allowed.
- ✓ Spaces are not allowed.
- ✓ Other special characters are not allowed.
- ✓ Key words cannot be used as used-defined identifiers.
- ✓ Names must be unique within a block.

Literals

Literals are values that are not represented by user-defined identifiers. Literals are of three types numeric, character and Boolean.

For example:

Numeric	**100, 3.14,-55, 5.25E7 or NULL**
Character	**'A', 'this is a string', '0001', '25-MAY-00', '', or NULL**
Boolean	**TRUE, FALSE or NULL**

In this list of values, '25-MAY-00' looks like a date value, but it is a character string. It can be converted to date format by using the TO_DATE function. The value"(two single quotes having nothing within) is another way of entering the NULL value.

PL/SQL is case sensitive regarding character values within single automation marks. The values 'ORACLE', 'Oracle', and 'oracle' are three different values in PL/SQL. To embed a single quote in a string value, two single quote symbols are entered for example, 'New Year"s Day'.

Numeric values can be entered in scientific notation with the letter E or e. Boolean values are not enclosed in quotation marks.

PL/SQL BLOCK STRUCTURE

PL/SQL is a block structured language. A program can be divided into logical blocks. The block structure gives modularity to a PL/SQL program, and each object within a block has 'scope". Blocks are of two types:

1. **An anonymous block** is a block of code without a name. It can be used anywhere in a program and is sent to the server engine for execution at runtime
2. **A named block** is a block of code that is named. A subprogram is a named block that can be called and can take arguments. A procedure is a subprogram that can perform an action, whereas a function is a subprogram that returns a value. A package is formed from a procedure and functions. A trigger is a block that is called implicitly by a DML statement.

A PL/SQL block consists if three sections:

A declaration section.
An executable section.
An exception-handling section.

Of the three sections in a PL/SQL block, only the executable section is mandatory. The declaration and exception-handling section are optional. The general syntax

[DECLARE
Declaration of constants, variables , cursors, and exception]
BEGIN
Executable PL/SQL and SQL statements
[Exception
Actions for error conditions]
END;

Section	Use
Declaration	An optional section to declare variable, constants, cursors, PL/SQL composite data types, and user-defined exception, which are referenced in execute and exception-handling section.
Executable	A mandatory section that contains PL/SQL statements to manipulate data in the block and SQL statements to manipulate the database.
Exception handling	Specifies action statements to perform when an error condition exits in the executable section. It also an optional section.

COMMENTS

Comments are used to document programs. They are written as part of a program, but they are not executed. In fact, comments are ignored by the PL/SQL engine. It is a good programming practice to add comments to a program, because this helps in readability and debugging of the program. There are two ways to write comments in PL/SQl:

1 To write a single-line comment, two dashes(--) are entered at the beginning of a new line for example,

--This is a single-line comment.

2. To write a multiline comment, comment text is placed between/*and*/. A multiline comment can be written on a separate line by itself, or it can be used on a line of code as well. For example,

/* This is a multiline comment that ends here.*/

A programmer can use a comment anywhere in the program.

Data types

Each constant and variable in the program neda a datatype.The data type decides the type of value that can be stored in a variable. PL/SQL has four data types.

- ✓ Scalar
- ✓ Composite
- ✓ Reference
- ✓ LOB.

Data types

A scalar data type is not made up pf a group of elements. It is atomic in nature. The composite data types are made up of elements or components. PL/SQL supports three composite data types records, tables and vaarays, which are discussed in a later chapter. The reference data types deal with objects, which are briefly introduced in Appendix D.

There are fur major categories of scalar data types:

- ✓ Character
- ✓ Number
- ✓ Boolean
- ✓ Date

Other scalar data types include raw, row id and trusted.

Character

Variables with a character data type can store text. The text may include letters, numbers and special characters. The text in character-type variable can be manipulated with built-in character function. Character data types include CHAR and VARCHAR2.

CHAR

The CHARdata type is used for fixed-length string values. The allowable string length is between 1 and 32,767.

VARCHAR2

The VARCHAR2 type is used for variable-length String values. The allowable length is between 1 and 32,767.

NUMBER

PL/SQL has a variety of numeric data types. Whole number or integer values can be handled by following data types:

BINARY_INTEGER
INTEGER
INT
SMALLINT
POSSITIVE
NATURAL

Similarly, there are various data types for decimal number

NUMBER
DEC (fixed-point number)
DECIMAL (fixed-point number)
NUMERIC (fixed-point number)
Fl.OAT (floating- point number)
REAL (floating-point number)
DOUBLE PRECISION (floating-point number)

BOOLEAN

PL/SQL has a logical data type, Boolean that is not available in SQL. It is used for Boolean data TRUE, FALSE or NULL only. These values are not enclosed in single quotation marks like character and data values.

DATE

The data type is a special data type that store date and time information. A date is always stored in standard 7-byte format.

OTHER DATA TYPES

NLS

The National Language (NLS) data type is for character sets in which multiple bytes are used for character representation. NCHR and NVARCHAR2 are examples of NLS data types.

LOB

It allows up to 4giga byte of data. LOB variable can be given one of the following data types:

BLOB
CLOB
NCLOB
BFILE

VARIABLE DECLARATION

A scalar variable or a constant is declared with a data type and an initial value assignment. The declaration are done in the declare in the DECLARE section of the program block. The initial value assignment for a variable is optional unless it has a NOT NULL constraint. The constants and NOT NULL type variable must be initialized. The general syntax is

DECLARE
Identifier name [Constant]data type[NOT NULL][:=\DEFAULT expression];

ASSIGNMENT OPERATION

The assignment operation is one of the way to assign a value to a variable. You have already learned that a variable can be initialized at the time of declaration by using the DEFAULT option or:=. The assignment operation is used in the executable section of the program block to assign a literal another variable value or the result of an expression to a variable the general syntax

VariableName:=Literal\VariableName\Expression;

BIND VARIABLES

Bind variable are also known as host variables. These variable are declared in the host SQL*Plus environment and are accessed by a PL/SQL block. Anonymous block do not take any arguments, but they can access host variable with a colon prefix(:) and the host variable name.

The syntax

VARIABLE variablename datatype

ARITHMETIC OPERATORS

Five standard arithmetic operators are available in PL/SQL for calculations

Exponentiation is performed first, multiplication and division are performed next and addition and subtraction are performed last.

If more than one operator of the same priority is present, they are performed last.

Whatever is in parentheses is performed first.

Arithmetic operator	Use
+	Addition
*	Multiplication
-	Subtraction
/	Division

CONTROL STRUCTURES AND EMBEDDED SQL

CONTROL STRUCTURE

✓ The basic programming control structure:

1. Sequential structure: A series of instructions are performed from the beginning to the end in a linear order.

2. Selection structure or decision structure or IF-structure: It involves conditions with a TRUE or False outcome.

✓ Based on the outcome, one of the options is performed, and the other option is skipped.

✓ Selection statements are also available for multiple options.

3. Looping structure: A series of instructions is performed repeatedly.

✓ There are different looping statements appropriate for a variety of situations.

I-SELECTION STRUCTURE:

✓ There are three selections or conditional statements in PL/SQL.

- Relational operators
- Logical operators
- Other special operators

✓ The AND and OR operator are binary operators, because they work on two conditions.

✓ The NOT operator is a unary operator, because it works on a single condition.

Relational operators

Relational operator	Meaning
=	Equal to
<> or !=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Logical operators

Logical Operator	**Meaning**
AND	Returns TRUE only if both conditions are true.
OR	Return TRUE if one or both conditions are true.
NOT	Return TRUE if the condition is false.

Truth tables for AND, Or, and NOT operators

AND	OR	NOT
TRUE AND TRUE =TRUE	TRUE AND TRUE =TRUE	NOT TRUE = FALSE
TRUE AND FALSE = FALSE	TRUE AND FALSE = TRUE	NOT FALSE = TRUE
FALSE AND TRUE = FALSE	FALSE AND TRUE = TRUE	NOT NULL = NULL
FALSE AND FALSE = FALSE	FALSE AND FALSE = FALSE	
NULL AND TRUE = NULL	NULL AND TRUE = TRUE	
NULL AND FALSE = FALSE	NULL AND FALSE = NULL	
NULL AND NULL = NULL	NULL AND NULL = NULL	

PL/SQL has five conditional or selection statements available for decision making:

1. **IF…THEN…END IF.**
2. **IF…THEN…ELSE…END IF.**
3. **IF…THEN…ELSEIF…END IF.**
4. **CASE…END CASE.**
5. **Searched CASE.**

1. **IF…THEN…END IF:**

- ✓ A simple IF statement performs action statements if the result of the condition is TRUE.
- ✓ If the condition is FALSE, no action is performed and the program continues with next statement in the block.
- ✓ The general syntax;

```
If CONDITIONS then
    Action statements
END IF;
```

Simple IF statement

```
SQL> DECLARE
  2       V_DAY VARCHAR2(9) := '&DAY';
  3       BEGIN
  4          IF (V_DAY = 'SUNDAY,) THEN
  5       DBMA_OUTPUT.PUT_LINE('SUNDAY IS A
                              HOLIDAY');
  6       END IF;
  7       END;
  8       /
  Enter value for day: SUNDAY
  SUNDAY IS A HOLIDAY
  PL/SQL procedure successfully completed.

  SQL>/
  Enter value for day: MONDAY
  PL/SQL procedure successfully completed
```

2. IF…THEN…ELSE…END IF:

- ✓ The IF…THEN…ELSE…END IF statement is an extension of the simple IF statement.
- ✓ It provides action statements for the TRUE outcome as well as for the FALSE outcome.
- ✓ The general syntax is:

IF condition(s) THEN
Action statements 1
ELSE
Action statements 2
END IF

- ✓ If the condition's outcome is TRUE, action statements 1 are performed.
- ✓ If the outcome is FALSE, action statements 2 are performed.

IF…THEN…ELSE…END IF STATEMENT

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2       V_AGE NUMBER(2) :'&AGE';
  3  BEGIN
  4       IF(V_AGE>=18)THEN
  5          DBMS_OUTPUT.PUT_LINE('AGE: '|| V_AGE || 'ADULT')
  6       ELSE
  7          DBMS_OUTPUT.PUT_LINE('AGE: '|| V_AGE || 'MINOR')
  8       END IF;
  9    END;
  10   /
Enter value for age :21
AGE: 21-ADULT
PL/SQL procedure successfully completed.
SQL>/
Enter value for age :21
AGE: 21-MINOR
PL/SQL procedure successfully completed.
```

2. IF…THEN…ELSEIF…END IF:

- ✓ The IF…THEN…ELSEIF…END IF statement is an extension to the previous statement.
- ✓ The ELSEIF alternative is more efficient than the other two.
- ✓ The DECODE function in SQL is not allowed in PL/SQL and the IF…THEN…ELSEIF…END IF statement is not allowed in SQL.
- ✓ The general syntax

```
IF conditions 1 THEN
   Action statements 1
  ELSEIF conditions 2 THEN
   Action statements 2
 …….
ELSEIF conditions N THEN
   Action statements N
[ELSE
      Else Action statements]
END IF;
```

- ✓ ELSEIF is a single word, but END IF uses two words.

DECODE function

```
SQL> SELECT  LNAME,  FName,
  2         DECODE (Positional, 1, Salary*1.2,
  3                   2, Salary*1.15,
  4                   3, Salary*1.1,
  5                    4, Salary*1.05,
  6                         Salary)  "New Salary"
  7  FORM employees;

LNAME            FNAME              New Salary
-------------    --------------     ------------------
Smith            John               3128000
Roberts          Sandi               86250
Dev              Derek               73150
Shaw             Jinku               92000
Garner           Stanley             24500
Chen             Sunney              51750

6 rows selected.

SQL>
```

ELSEIF statement

```
SQL> DECLARE
  2       v_pos     NUMBER(1) := &Position;
  3   BEGIN
  4       IF v_pos=1 THEN
  5            DBMS_OUTPUT.PUT_LINE('20% increase')
  6        ELSEIF v_pos=2 THEN
  7            DBMS_OUTPUT.PUT_LINE('15% increase')
  8       ELSEIF v_pos=3THEN
  9            DBMS_OUTPUT.PUT_LINE('10% increase')
 10       ELSEIF v_pos=4 THEN
 11            DBMS_OUTPUT.PUT_LINE   ('5% increase')
 12       ELSE
 13            DBMS_OUTPUT.PUT_LINE   ('No increase')
 14       END IF;
 15   END;
 16   /
Enter value for position:2
15% increase
PL/SQL procedure successfully completed.
SQL>
```

- ✓ The five simple IF statements to accomplish the same task as that performed by a single compound ELSEIF statement.
- ✓ It assigns a grade of A, B, C, D, or F based on v_score.
- ✓ The score is within the range of 0 to 100.
- ✓ Five simple IF statements with total of 10 conditions or two conditions per each statement
- ✓ The first statement's condition is TRUE, so v_grade will be assigned 'A'.
- ✓ Because all simple IF statements are independent statements, the execution will continue with the next IF, and so on.

SIMPLE IF WITH MULTIPLE CONDITIONS

```
SQL> DECLARE
  2       S NUMBER(3)  := &SCORE;
  3       GRADE     CHAR;
  4  BEGIN
  5      IF S >=  90  AND  S <= 100 THEN
  6            GRADE    := 'A';
  7         END IF;
  8     IF S >=  80  AND S <= 89 THEN
  9            GRADE    := 'B';
 10         END IF;
 11       IF S >=  70  AND S <= 79 THEN
 12            GRADE    := 'C';
 13         END IF;
 14       IF S >=  60  AND S <= 69 THEN
 15            GRADE    := 'D';
 16          END IF;
 17      IF S >= 0  AND S <= 59 THEN
 18            GRADE    := 'F';
 19         END IF;
 20     IF S < 0  AND S > 100 THEN
 21            GRADE    := 'U';
 22        END IF;
 23        DBMS_OUTPUT.PUT_LINE('SCORE IS '|| TO_CHAR(S));
 24         DBMS_OUTPUT.PUT_LINE   ('GRADE IS '|| GRADE);
 25  END;
 26  /
Enter value for score:93
SCORE IS 93
GRADE IS A

PL/SQL procedure successfully completed.

SQL>
```

ELSEIF statement

```
SQL> DECLARE
  2       S NUMBER(3)  := &SCORE;
  3       GRADE      CHAR;
  4  BEGIN
  5      IF S >=  90  AND  S <= 100 THEN
  6            GRADE    := 'A';
  7     ELSIF S >=  80  AND S <= 89 THEN
  8            GRADE    := 'B';
  9      ELSIF S >=  70  THEN
 10            GRADE    := 'C';
 11      ELSIF S >=  60  THEN
 12            GRADE    := 'D';
 13     ELSIF S >= 0 THEN
 14            GRADE    := 'F';
 15     ELSIF S < 0  AND S > 100 THEN
 16            GRADE    := 'U';
 17        END IF;
 18        DBMS_OUTPUT.PUT_LINE('SCORE IS '|| TO_CHAR(S));
 24         DBMS_OUTPUT.PUT_LINE   ('GRADE IS '|| GRADE);
 25  END;
 26  /
Enter value for score:77
SCORE IS 77
GRADE IS C

PL/SQL procedure successfully completed.

SQL>
```

- ✓ The ELSIF statement reduces the number of conditions form 10 to 5 and the number of statements form five to one.
- ✓ The condition is TRUE in the first IF clause and v_grade is assigned value 'A'.
- ✓ The statement will not continue down anymore, because it will not enter the ELSEIF part.

4. CASE:

- ✓ The CASE statement is an alternative to the IF…THEN…ELSEIF…ENDIF statement.
- ✓ The CASE statement begins with key word CASE and ends with the key words END CASE.
- ✓ The body of the CASE statement contains WHEN clauses, with values or conditions, action statements.
- ✓ WHEN clause's value/condition evaluates to TRUE, its action statements are executed.
- ✓ The general syntax is

```
CASE[variable_name]
    WHEN value1\condition1 THEN action_statement1;
    WHEN value2\condition2 THEN action_statement2;
     WHEN valueN\conditionN THEN action_statementN;
       ELSE action_statement;
     END CASE;
```

CASE statement

```
SQL> DECLARE
  2     V_NUM    NUMBER := &ANY_NUM;
  3     V_RES NUMBER;
  4  BEGIN
  5     V_RES := MOD(V_NUM,2);
  6     CASE V_RES
  7       WHEN 0 THEN DBMS_OUTPUT.PUT_LINE(V_NUM || ' IS
                                                  EVEN');
  8       ELSE DBMA_OUPUT.PUT_LINE(V_NUM || ' IS ODD');
  9       END CASE;
 10       END
 11       /
Enter value for any_num:5
5 is ODD

PL/SQL procedure successfully completed.

SQL>
```

(a) Searched CASE:

* A statement with a value is known as a CASE statement, and a statement with conditions is known as a searched CASE statement.

* A CASE statement uses variable_name as a selector, but a searched CASE does not use variable_name as a selector.

Searched CASE statement

```
SQL> DECLARE
  2     V_NUM    NUMBER := &ANY_NUM;
  3  BEGIN
  4     CASE
  5      WHEN  MOD(V_NUM,2) =0 THEN
  6        DBMS_OUTPUT.PUT_LINE(V_NUM || ' IS EVEN');
  7      ELSE
  8        DBMA_OUPUT.PUT_LINE(V_NUM || ' IS ODD');
  9      END CASE;
 10  END
 11  /
Enter value for any_num:5
5 is ODD

PL/SQL procedure successfully completed.
SQL>
```

(b) Nested IF:

- ✓ The nested IF statement contains an IF statement within another IF statement.
- ✓ If the condition in the outer IF statement is TRUE, the inner IF statement is performed.
- ✓ Any IF statement with a compound condition a nested IF statement.
- ✓ For example, insurance surcharge based on an individuals gender and age.
- ✓ There are four categories:
  1. Male 25 or over
  2. Male under 25
  3. Female 25 or over
  4. Female under 25

Simple IF with multiple condition

```
SQL> DECLARE
  2        V_GENDER CHAR := '&SEX';
  3        V_AGE NUMBER(2) := '&AGE';
  4        V_CHARGE NUMBER(3,2);
  5  BEGIN
  6     IF (V_GENDER = 'M' AND V_AGE >=25) THEN
  7          V_CHARGE := 0.05;
  8     END IF;
  9     IF (V_GENDER = 'M' AND V_AGE < 25) THEN
 10          V_CHARGE := 0.10;
 11     END IF;
 12      IF (V_GENDER = 'F' AND V_AGE >=25) THEN
 13          V_CHARGE := 0.03;
 14     END IF;
 15      IF (V_GENDER = 'F' AND V_AGE < 25) THEN
 16          V_CHARGE := 0.06;
 17      END IF;
 18      DBMA_OUTPUT.PUT_LINE('GENDER: ' || V_GENDER);
 19      DBMA_OUTPUT.PUT_LINE('AGE: || TO_CHAR(V_AGE));
 20      DBMA_OUTPUT.PUT_LINE('SURCHARGE: ' ||
                                    TO_CHAR(V_CHARGE));
 21  END;
 22  /
Enter value for sex: F
Enter value for age: 18
GENDER: F
AGE:18
SURCHARGE:.06
PL/SQL procedure successfully completed.
SQL>
```

5. LOOPING STRUCTURE

- ✓ Looping means iterations.
- ✓ A loop repeats a statement or a series of statement a specific number of times, as defined by the programmer.
- ✓ Three types of looping statements:
  1. **Basic loop**
  2. **WHILE loop**
  3. **FOR loop**

(a) Basic loop:

- ✓ A basic loop is a loop that is performed repeatedly.
- ✓ Once a loop is entered, all statements in the loop are performed.
- ✓ The loop will continue infinitely.
- ✓ An infinite loop, or a 'never-ending loop,' is a logical error in programming.
- ✓ A basic loop is by adding an EXIT statement inside the loop.
- ✓ The general syntax is:

```
LOOP
        Looping statement1;
    Looping statement2;
      …..
    Looping statementN;
      EXIT [WHEN condition];
END LOOP;
```

- ✓ The EXIT statement in a loop could be an independent statement,

```
IF v_count > 10 THEN
 EXIT

END IF
```

- ✓ It also add a condition with the optional WHEN clause that will end the loop when the condition becomes true.

For **example**,

```
EXIT WHEN v_count > 10;
```

Counter-controlled basic loop

(B) WHILE LOOP:

- ✓ The WHILE loop is an alternative too the basic loop and is performed as along as the condition is true.
- ✓ It terminates when the condition becomes false.
- ✓ If the condition is false at the beginning of the loop, the loop is not performed at all.
- ✓ The WHILE loop does not need an EXIT statement to terminate.
- ✓ The general syntax is

```
WHILE condition LOOP
    Looping statement1;
    Looping statement2;
    …….
    Looping statement;
END LOOP;
```

COUNTER-CONTRILLED WHILE LOOP

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2     V_COUNT       NUMBER(2);
  3     V_SUM         NUMBER(2) := 0;
  4     V_AVG          NUMBER(3, 1);
  5  BEGIN
  6  V_COUNT   := 1;
  7  WHILE V_COUNT <= 10 LOOP   /* CONDITION*/
  8      V_SUM  := V_SUM + V_COUNT;
  9      V_COUNT  := V_COUNT  + 1;
 10   END LOOP;
 11   V_AVG := V_SUM / (V_COUNT-1);
 12    DBMA_OUTPUT.PUT_LINE(' AVERAGE OF 1 TO
                                10 IS' || TO_CHAR(V_AVG));
 13   END;
 14   /
AVERAGE OF 1 TO 10 IS 5.5

PL/SQL procedure successfully completed.

SQL>
```

DIFFERENCES BETWEEN A BASUC LOOP AND A WHILE LOOP

BASIC LOOP	**WHILE LOOP**
1. It is performed as long as the condition is false. 2. It tests the condition inside the loop 3. It is performed at least one time. 4. It needs the EXIT statement to terminate.	1. It is performed as long as the condition is true. 2. It checks the condition before entering the loop 3. It is performed zero or more time. 4. There is no need for an EXIT statement.

(C) FOR LOOP:

- ✓ The FOR loop is the simplest loop.
- ✓ There is no need to use an EXIT statement, and the counter need not be declared.
- ✓ The counter used in the loop is implicitly declared as an integer, and it is destroyed on the loop's termination.
- ✓ The general syntax is

FOR counter IN [REVERSE] lower…upper LOOP
Looping statement1
Looping statement2
……
Looping statementN
END LOOP;

- ✓ The counter varies from the lower value to the upper value, incrementing by one with every loop execution.
- ✓ The loop can also be used with the counter starting at a higher value and decrementing by one with every loop execution.
- ✓ The REVERSE is used for varying the counter in the reverse order, or from a higher to a lower value.

FOR loop

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2     V_COUNT       NUMBER(2);
  3     V_SUM         NUMBER(2) := 0;
  4     V_AVG          NUMBER(3, 1);
  5  BEGIN
  6  FOR V_COUNT  IN 1….10  LOOP
  7      V_SUM  := V_SUM  + V_COUNT;
  8  END LOOP;
  9   V_AVG := V_SUM / 10;
 10   DBMA_OUTPUT.PUT_LINE(‘ AVERAGE OF 1
                        TO 10 IS’ || TO_CHAR(V_AVG));
 13  END;
 14  /
AVERAGE OF 1 TO 10 IS 5.5

PL/SQL procedure successfully completed.

SQL>
```

(D) Nested loops:

- ✓ Loops can be nested too many levels.
- ✓ It can end an outer loop from within the inner loop by labeling each loop and then using the EXIT statement.
- ✓ EXIT WHEN exists the current loop, but EXIT out_loop WHEN also exits the outer loop.
- ✓ The loop labels use the same naming rules as those used for identifiers.
- ✓ The loops are labeled before the key word LOOP on the same line or on a separate line.

- ✓ The loop label is enclosed within two pairs of angel brackets (<< and >>).
- ✓ For example,

```
<< out _loop>>
LOOP
…….
   EXIT WHEN condition;
<<in_loop>>
      LOOP
      …….
      EXIT out_loop WHEN condition;
      EXIT WHEN condition;
      ……
      END LOOP in_loop;
     ……
  END LOOP out_loop;
```

6. NESTED BLOCKS

- ✓ The execution starts with the outer block and continues with the inner block.
- ✓ The variables declared in the outer block are global too the inner block, and they are accessible in the inner block.
- ✓ The variables declared in the inner block, are nor accessible in the outer block.
- ✓ For example,

```
DECLARE                    /* OUTER BLOCK STARTS */
    Var1 NUMBER;
BEGIN
    ……
   DECLARE          /* INNER BLOCK STARTS */
      Var2 NUMBER;
   BEGIN
      ……
      END;
      …….
  END;
```

SQL IN PL/ SQL

- ✓ The PL/SQL statements have control structures for calculations, decision making, and iterations.
- ✓ SQL can be used to retrieve and change information.
- ✓ PL/SQL supports all Data Manipulation Language (DML) statements, such as INSERT, UPDATE, and DELETE.
- ✓ It also supports the Transaction Control Language statements ROLLBACK, COMMIT, and SAVEPOINT.
- ✓ It can retrieve data using the data retrieval statement SELECT.
- ✓ A row of data can be used to assign values to variables.
- ✓ PL/SQL statements can use single-row functions, but group functions are not available for PL/SQL statements
- ✓ PL/SQL does not support (DDL) and (DCL) data control language.

(A) SELECT Statement in PL/SQL:

- ✓ The general syntax

```
SELECT columnnames
INTO variablenames/RecordName
FORM tablename
WHERE condition;
```

- ✓ **Columnnames** must contain at least one column and may include arithmetic or string expressions, single-row functions and group functions.
- ✓ **Variablenames** must contain a list lf local or host variables to hold values retrieved by the SELECT clause.
- ✓ The **INTO** clause must contain one variable for each value retrieved form the table.
- ✓ The order and data type of the columns and variables must correspond.
- ✓ The **SELECT…INTO** statement must return one and only one row.
- ✓ The **EMPLOYEE** table is retrieved into a series of variables.

SELECT-INTO in PL/SQL

```
SQL> DECLARE
  2      V_LAST
  3      V_FIRST
  4      V_SAL
  5   BEGIN
  6   SELECT LNAME, FNAME, SALARY
  7        INTO V_LAST, V_FIRST, V_SAL
  8        FROM EMPLOYEE
  9  WHERE EMPLOYEEID = 200;
 10  DBMS_OUTPUT.PUT_LINE('EMPLOYEE NAME:'|| ' ' ||
 11                                              V_LAST);
 12 DBMS_OUTPUT.PUT_LINE('SALARY:    '|| TO_CHAR(V_SAL));
 13  END;
 14  /
EMPLOYEE NANE: Kavitha

Salary:              25000

PL/SQL procedure successfully completed.

SQL>
```

7. DATA MANIPULATION IN PL/SQL

- ✓ All DML statements in PL/SQL with the same syntax are used in SQL. The three DML Statements which manipulate data are:

1. The **INSERT** statement to add a new row in a table.
2. The **DELETE** statement to remove a row or rows.
3. The **UPDATE** statement to change values in a row or rows.

(A) INSERT Statement

- ✓ An INSERT statement adds a new employee in the EMPLOYEE table.
- ✓ The statement uses the sequences created earlier.
- ✓ For simplicity a few columns are used in the statement as shown.
- ✓ NEXTVAL uses the next value from the sequence as the new EmployeeId, and CURRVAL uses the current value of the department from the sequence.
- ✓ To insert today's date as the hire date SYSDATE function is used for the value.

```
SQL>        BEGIN
  2        INSERT INTO EMPLOYEE
  3          (EMPLOYEEID, LNAME, FNAME, SALARY, DEPTID)
  4          VALUES
  5            (EMPLOYEE_EMPLOYEEID_SEQ.NEXTVAL,'RAI'
  6               'AISH', 90000, DEPT_DEPTID_SEQ.CURRVAL);
  7        COMMIT;
  8   END;
  9   /

  PL/SQL procedure successfully completed.
  SQL>
```

(B) DELETE Statement

- ✓ The DELETE statement in the PL/SQL blocks to remove some rows. NamanNavan (N2) Corporation decides to remove the IT Department.
- ✓ All the employees belonging to that department must be removed from the EMPLOYEE table as shown in the DELETE statement in PL/SQL.

```
SQL>DECLARE
   2        V_DEPTID   DEPT.DEPTID%TYPE;
   3          BEGIN
   4        SELECT DEPTID
   5          INTO V_DEPT
   6        FROM DEPT
   7        WHERE UPPER(DEPTNAME) = '&DEPT_NAME'
   8        DELETE FROM EMPLOYEE
   9          WHERE DEPTID = V_DEPTID;
  10        COMMIT;
  11  END;
  12  /
Enter value for dept_name: IT

PL/SQL procedure successfully completed.

SQL>
```

(C) UPDATE Statement

- ✓ The UPDATE statement can be used in PL/SQL block for modification of data.
- ✓ The company decides to give a bonus commission to all the employees who are entitled to commission.
- ✓ The bonus is 10% of the commission received. In an example UPDATE statement in PL/SQL block shows the modify commission.

```
SQL>DECLARE
    2       V_INCREASE NUMBER :=
&DECIMAL_INCREASE;
    3  BEGIN
    4       UPDATE EMPLOYEE
    5          SET SALARY = SALARY * (1 + V_INCREASE)
    6       WHERE EMPLOYEEID = &EMP_ID;
    7       COMMIT;
    8  END;
    9  /
Enter value for decimal_increase: 0.15
Enter value for emp_id: 545

PL/SQL procedures successfully completed.
SQL>
```

8 - TRANSACTION CONTROL STATEMENT

- ✓ After performing a DML statement from the INSERT, DELETE and UPDATE tables as shown, the sample blocks have used a COMMIT statement.
- ✓ To commit within the PL/SQL block, the data will be written to the disk right away, and the locks from those rows will be released.
- ✓ All transaction control statements are allowed in the PL/SQL and are as follows:
- ✓ The COMMIT statement to commit the current transaction.
- ✓ The SAVEPOINT statement to mark a point in your transaction.
- ✓ The ROLLBACK [TO SAVEPOINT *n*] statement to discard all or part of the transaction.

PL/SQL CURSORS AND EXCEPTIONS

- ✓ Oracle assigns a private work area for that statement.
- ✓ The work area, called a cursor, stores the statement and the results returned by execution of that statement.
- ✓ A cursor is created either implicitly or explicitly.

The cursor in PL/SQL is of two types:

2. STATIC CURSOR

- ✓ The contents are known at compile time.
- ✓ The cursor object for such an SQL statement is always based on one SQL statement.

3. DYNAMIC CURSOR

- ✓ The cursor variable that can change its value is used.
- ✓ The variable can refer to different SQL statements at different times.

The static cursor is of two types:

1. IMPLICIT CURSOR
2. EXPLICIT CURSOR

1. IMPLICIT CURSORS

- ✓ PL/SQL creates an implicit cursor when an SQL statement is executed from within the program block.
- ✓ The implicit cursor is created only if an explicit cursor is not attached to that SQL statement.
- ✓ Oracle opens an implicit cursor, and the pointer is set to the first row in the cursor.
- ✓ The PL/SQL engine closes the implicit cursor automatically.
- ✓ A programmer cannot perform on an implicit cursor all the operations that are possible on explicit cursor statements.
- ✓ PL/SQL creates an implicit cursor for each DML statements in the PL/SQL code.
- ✓ An explicit cursor for DML statements cannot be used.
- ✓ Having no control over the implicit cursor, the implied queries perform operations on it.

- ✓ PL/SQL actually tries to fetch twice to make sure that a TOO_MANY_ROWS exception does not exist.
- ✓ The explicit cursor is more efficient, because it does not try that extra fetch.
- ✓ It is possible to use an explicit cursor for a SELECT statement that returns just one row, because one has control over it.

For example,

```
CURSOR deptname_cur IS
SELECT DeptName, Location FROM dept WHERE DeptId = 10;
```

- ♠ Here, only one row is retrieved by the cursor with two column values, Finance and Charlotte, it can be assigned to variables by fetching that row.

2. EXPLICIT CURSORS

- ✓ An explicit cursor is declared as a SELECT statement in the PL/SQL block.
- ✓ There are cursor attributes in PL/SQL to get the status information on explicit cursors.

Four actions can be performed on an explicit cursor:

1. **Declare it.**
2. **Open it.**
3. **Fetch row(s) from it.**
4. **Close it.**

(a) Declaring an Explicit Cursor

- ✓ A cursor is declared as a SELECT statement.
- ✓ The SELECT statement must not have an INTO clause in a cursor's declaration.
- ✓ An ORDER BY clause can be used in the SELECT statement.
- ✓ The ***general syntax*** is

```
DECLARE
   CURSOR cusrsorname IS
      SELECT statement;
```

- ✓ Where cursorname is the name of the cursor that follows identifier-naming rules.
- ✓ The SELECT statement is any valid data-retrieval statement.
- ✓ The cursor declaration is done in the DECLARE section of the PL/SQL block, but a cursor cannot be used in programming statements of expressions, as with other variables.

For example, the two cursors are declaration.

- ♠ In one, the cursor is based on a SELECT query that will retrieve all rows from the DEPT table in the work area.
- ♠ In the other, two columns, EmployeeId and Salary are selected into the cursor with DeptId equal to 20.

EXPLICIT CURSOR- Ex1:

.

```
SQL> DECLARE
  2        CURSOR DEPT_CUR
  3         IS
  4              SELECT *
  5              FROM DEPT;
  6  BEGIN
  7       . . .
  8  END;
```

EXPLICIT CURSOR-Ex2:

```
SQL> DECLARE
  2     CURSOR EMPLOYEE_CUR
  3      IS
  4         SELECT EMPLOYEEID, SALARY
  5         FROM EMPLOYEE
  6         WHERE DEPTID = 20;
  7  BEGIN
  8       . . .
  9  END;
```

(b) Action on Explicit Cursors:

- ✓ Action are performed on cursors declared in the DECLARE section of the block.

(c) Opening a Cursor:

- ✓ When a cursor is opened, its SELECT query is executed.
- ✓ The active set is created using all tables in the query and then restricting to rows that meet the criteria.
- ✓ The cursor points to the first row in the active set.
- ✓ PL/SQL uses an OPEN statement to open a cursor.
- ✓ The ***general syntax*** is:

OPEN cursorname;

For example,

OPEN employee cur;

(d) Fetching Data from a Cursor:

- ✓ The SELECT statement creates an active set based on tables in the FROM clause, column names in the SELECT clause, and rows based on conditions in the WHERE clause.
- ✓ The ***general syntax*** is
  - **FETCH cursorname INTO variablelist / recordname;**
- ✓ **variablelist** may include a local variable, a table, or a bind variable and **recordname** is the name of a record structure.
- ✓ For example,
  - **FETCH employee_cur INTO v_empnum, v_sal;**

OR FETCH employee_cur INTO emp_rec;

➢ emp_rec is declared with %ROWTYPE declaration attribute:

Emp_rec employee_cur%ROWTYPE

(e) Closing a cursor:

- ✓ It is done with a cursor, it should close it.
- ✓ A closed cursor can be reopened again.
- ✓ PL/SQL uses the CLOSE statement to close a cursor.
- ✓ The general syntax,

CLOSE cursorname;

For example,

CLOSE employee_cur;

EXPLICIT CURSOR ATTRIBUTES:

- ♠ The four explicit cursor attributes are:

%ISOPEN – It returns TRUE, the cursor is open, otherwise, it returns FALSE.
%FOUND – It returns TRUE, if the last fetch returned a row, otherwise, it returns FALSE.
%NOTFOUND – It returns TRUE if the last fetch did not return a row, otherwise, it returns FALSE. It complements the %FOUND attribute.
%ROWCOUNT - It returns total number of rows returned.

Cursor attributes

```
SQL>SET SERVEROUTPUT ON
SQL>DECLARE
  2     V_LAST EMPLOYEE.LNAME%TYPE
  3     V_FIRST EMPLOYEE.FNAME%TYPE;
  4     V_SAL EMPLOYEE.SALARY5TYPE;
  5     CURSOR EMPLOYEE_CUR IS
  6     SELECT LNAME, FNAME, SALARY
  7     FROM EMPLOYEE
  8     WHERE DEPID = 20;
  9   BEGIN
 10     IF NOT EMPLOYEE_CUR%ISOPEN THEN
 11          OPEN EMPLOYEE_CUR
 12     END IF;
 13     LOOP
 14          FETCH EMPLOYEE_CUR
 15               INTO V_LAST, V_FIRST, V_SAL;
 16          EXIT WHEN NOT EMPLOYEE_CUR%FOUND;
 17          DBMS_OUTPUT.PUT_LINE(V_FIRST || ' ' || V_LAST || ' ' ||
 18                                                        V_SAL)
 19     END LOOP;
 20     DBMS_OUTPUT.PUT_LINE(EMPLOYEE_CUR%ROWCOUNT
 21                                    || 'EMPLOYEES FOUND');
 22     END;
```

```
23      /
Kumar 66500
Ravi    80000
2 employees found

PL/SQL procedure successfully completed.

SQL>
```

IMPLICIT CURSOR ATTRIBUTES:

- ✓ An implicit cursor cannot be opened, fetched from, or closed with a statement.
- ✓ The cursor attributes are available for an implicit cursor with the name SQL as a prefix.
- ✓ The four attributes for a implicit cursor are:

1. SQL%ISOPEN
2. SQL%ROWCOUNT
3. SQL%NOTFOUND
4. SQL%FOUND

CURSOR FOR LOOPS:

- ✓ The cursor FOR loop to writes a loop for explicit cursors.
- ✓ The cursor is opened implicitly the loop starts, a row is then fetched into the record from the cursor with every iteration of the loop.
- ✓ The cursor is closed automatically the loop ends.
- ✓ The general ***syntax*** is

```
FOR recordname IN cursor LOOP
    Loop statements;
END LOOP;
```

- ➢ recordname is the name of the record that is declared implicitly in the loop.
- ➢ cursorname is the name of declared explicit cursor.

CURSOR FOR LOOP

(a) Cursor FOR Loop Using a Subquery:

- ✓ Use of a subquery in the cursor FOR loop eliminates declaration of an explcit cursor.
- ✓ The cursor is created by a subquery in the FOR loop statement itself.
- ✓ An explicit cursor is used with implicit actions.
- ✓ This subquery is similar to the inline view covered in the SQL section.

Cursor FOR loop with a subquery

```
SQL>BEGIN
  2       FOR EMP_REC IN
  3            (SELECT FNAME, LNAME, SALARY, COMMISSION
  4            FROM EMPLOYEE
  5            WHERE DEPID = 10) LOOP
  6            DBMS_OUTPUT.PUT_LINE(EMP_REC.FNAME || ' '
  7                    ||  EMP_REC.LNAME ||'$'||TO_CHAR(EMP_REC.
  8                      SALARY + NVL(EMP_REC.COMMISSION, 0)));
  9   END LOOP;
 10  END;
 11   /
JOHN SMITH $ 3000000
Roberts $ 750000
Sunny $ 350000
PL/SQL procedure successfully completed.
SQL>
```

SELECT….FOR….UPDATE CURSOR:

```
SQL> SET SERVEROUTPUT ON
SQL>DECLARE
  2       CURSOR EMPLOYEE_CUR IS
  3       SELECT LNAME, FNAME, SALARY
  4       FROM EMPLOYEE;
  5  BEGIN
  6       FOR EMP_REC IN EMPLOYEE_CUR LOOP
  7            IF EMP_REC.SLARY > 75000 THEN
  8            DBMS_OUTPUT.PUT(EMP_REC.FNAME || ' ');
  9            DBMS_OUTPUT.PUT(EMP_REC.LNAME || ' ');
 10            DBMS_OUTPUT.PUT_LINE(EMP_REC.SALARY || ' ');
 11       END IF;
 12   END LOOP;
 13  END;
 14   /
JOHN SMITH 255000
KUMAR  150000
DEREK DEV  800000
PL/SQL procedure successfully completed.
SQL>
```

- ✓ The SELECT query, the result is returned to without locking any rows in the table.
- ✓ Row locking is kept to a minimum.
- ✓ The FOR UPDATE clause is used with the SELECT query for row locking.
- ✓ Rows that are locked for update do not have to be updated.
- ✓ The ***general syntax*** is,

```
CURSOR cursorname IS
    SELECT columnnames
        FROM tablenames
            [WHERE condition]
                FOR UPDATE [OF columnnames] [NOWAIT];
```

- ✓ The optional part of a FOR UPDATE clause is OF columnnames, which enables to specify columns to be updated.
- ✓ The optional word NOWAIT – one or more rows are already locked by another user, to wait until the lock is released.

WHERE CURRENT OF CLAUSE:

- ✓ The WHERE CURRENT OF clause allows to perform data manipulation only on a recently fetched row.
- ✓ The ***general syntax*** is

```
UPDATE tablename
SET clause
WHERE CURRENT OF cursorname;
DELETE FROM tablename
WHERE CURRENT OF cursorname;
```

CURSOR WITH PARAMETERS:

- ✓ A cursor can be declared with parameters, to pass values to the cursor.
- ✓ These values are passed to the cursor, when it is opened, and they are used in the query when it is executed.
- ✓ The use of parameters, it can open and close a cursor many times with different values.
- ✓ The cursor with different values will then return different active sets each time it is opened.
- ✓ The ***general syntax*** is,

```
CURSOR cursorname
        [(parameter1 datatype, parameter2 datatype,…..)]
IS
SELECT query;
```

- ✓ parameter1, parameter2, and so on are formal parameters passed to the cursor.
- ✓ datatype is any scalar data type assigned to the parameter.
- ✓ The parameters are assigned only data types, they are not assigned size.
- ✓ A cursor is opened, values are passed too the cursor.
- ✓ Each value must match the positional order of the parameters in a cursor's declaration.
- ✓ For example, the cursor employee_cur is declared with a parameter dept_num, it is also used in the cursor SELECT statement's Where clause.
- ✓ To input a value for department number with substitution variable DEPARTMENT_ID, it is assigned to variable D_ID.
- ✓ The formal parameter DEPT_NUM gets value of parameter D_ID.
- ✓ The active set is created based on DEPTID = DEPT_NUM.
- ✓ The cursor loop prints all employees for department number 10.
- ✓ The parameter can be passed a value with a literal, a bind variable, or an expression.

CURSOR WITH PARAMETER

```
SQL>SET SERVEROUTPUT ON
SQL>DECLARE
  2     V_LAST EMPLOYEE.LNAME%TYPE
  3     V_FIRST EMPLOYEE.FNAME%TYPE;
  4     D_ID NUMBER(2) := &DEPARTMENT_ID;
  5     CURSOR EMPLOYEE_CUR(DEPT_NUM EMPLOYEE.DEPTID%TYPE) IS
  6     SELECT LNAME, FNAME
  7     FROM EMPLOYEE
  8     WHERE DEPID = DEPT_NUM;
  9  BEGIN
 10       OPEN EMPLOYEE_CUR(D_ID);
 11        DBMS_OUTPUT.PUT_LINE('EMPLOYEE IN DEPARTMENT' ||
 12                                                    TO_CHAR(D_ID));
 13     LOOP
 14          FETCH EMPLOYEE_CUR INTO V_LAST, V_FIRST);
 15               EXIT WHRN EMPLOYEE_CUR%NOTFOUND;
 16               DBMS_OUTPUT.PUT_LINE(V_LAST || ',' || V_FIRST);
 17          END LOOP;
 18             CLOSE EMPLOYEE_CUR;
 19          END;
 20          /

Enter value for department_id:10
EMPLOYEE IN DEPARTMENT 10
SMITH,
JOHN
SUNNY
ROBERTS

PL/SQL procedure successfully completed.

SQL>
```

CURSOR VARIABLE: AN INTRODUCTION:

- ✓ A cursor is based on one specific query, whereas a cursor variable can be opened with different queries within a program.
- ✓ A static cursor is like a constant, and a cursor variable is like a pointer to that cursor.
- ✓ It also uses the action statements OPEN, FETCH, and CLOSE with cursor variables.
- ✓ The cursor attributes %ISOPEN, %FOUND, %NOTFOUND, and %ROWCOUNT are available for cursor variables.
- ✓ The cursor variable, used in an assignment statement.
- ✓ A cursor variable can also be assigned to another cursor variable.

(a) REF CURSOR TYPE:

- ✓ Cursor variables create two steps:
  - ➢ To create a referenced cursor type.
  - ➢ To declare an actual cursor variable with the referenced cursor type.

The general syntax is,

TYPE cursortypename IS REF CURSOR [RETURN returntype];
Cursorvarname cursortypename;

cursortypename is the name of the type of cursor.

- ✓ The RETURN clause is optional.
- ✓ The returntype is the RETURN data type and any valid data structure.
- ✓ **For example,**

```
TYPE any_cursor_type IS REF CURSOR;
Any_cursor_var          any_cursor_type;
TYPE employee_cursor_type IS REF CURSOR
RETURN employee%ROWTYPE;
Employee_cursor_var          employee_cursor_type;
```

- ✓ The first cursor type any_cursor_type is called the **weak type**, because its RETURN clause is missing.
- ✓ The second cursor type declared with the RETURN clause is called the **strong type,** because it links a row type to the cursor type at the declaration time.

(b) Opening a Cursor Variable:

- ✓ The ***general syntax*** is,

OPEN cursorname/cursorname FOR SELECT query;

- ✓ The cursor type is declared with the RETURN clause, the structure form the SELECT query must match the structure specified in the REF CURSOR declaration.

For example,

```
OPEN exployee_cursor_var For SELECT * FROM employee;
```

- ✓ The structure returned by the select query matches the RETURN type employee%ROWTYPE.
- ✓ The other cursor type, any_cursor_type, is declared without the RETURN clause.
- ✓ Some OPEN statement foro the weak cursor variable:

```
OPEN any_cursor_var FOR SELECT * FROM dept;

OPEN any_cursor_var FOR SELECT * FROM employee;

OPEN any_cursor_var FOR SELECT Deptid * FROM Dept;
```

(c) Fetching from a CURSOR Variable:

- ✓ The compiler checks the data structure type after the INTO clause.
- ✓ The ***general syntax*** is,

FETCH cuursorvarname INTO recordname / variablelist

EXCEPTIONS:

- ✓ In PL/SQL error are known as exceptions.
- ✓ Exceptions can result from a system error, a user error, or an application error.
- ✓ An exception occurs, control of the current program block shifts to another section of the program, known as the exception section, to handle exception.
- ✓ PL/SQL provides ways to trap and handle errors and it is possible to create PL/SQL programs with full protection against errors.
- ✓ The program may have more than one exception handler, WHEN…THEN statements like an ELSIF or CASE structure.
- ✓ **For example,**

```
DECLARE
    Declaration of constants, variables, cursors, and exceptions
BEGIN
    /* Exception is raised here. */
EXCEPTION
    /* Exception is trapped here.*/
END;
```

- ➢ The general syntax of exception section is,

```
EXCEPTION
    WHEN exceptionname1 [OR exceptionname2…] THEN
        Exception statements
    [WHEN exceptionname3 [OR exceptionname4…] THEN
        Executable statements]
[WHEN OTHERS THEN
    Executable statements]
```

- ✓ An exception is handled when the exception name matches the name of the raised exception.
- ✓ The exceptions are trapped by name.
- ✓ IF an exception is raised by no handler for it is present, the WHEN OHTERS clause is performed.

(a) TYPES OF EXCEPTIONS:

- ✓ There are three types of exception in PL/SQL:

1. Predefined Oracle server exceptions:

- ✓ Exceptions that are given name by PL/SQL are declared in a PL/SQL package called STANDARD.
- ✓ The exception-handling routine is also defined.

Predefined/named system exceptions

Exception Name	Error-Number	Brief Description
NO_DATA_FOUND	ORA-01403	Single-row SELECT returned no data.
TOO_MANY_ROWS	ORA-01422	Single-row SELECT returned more than one row.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero.
VALUE_ERROR	ORA-06502	Arithmetic, conversion, truncation, or size constraint error occurred.
STORAGE_ERROR	ORA-06500	PL/SQL ran out of memory, or memory is corrupted.
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal program.

- ✓ Suppose a program block generates an error message for exception error number ORA-01403 that is not handled by the exception section.
- ✓ The error has occurred because of a SELECT statement that did not return any data.

Handling Named Exceptions (Source)

- ✓ Two named exceptions – NO_DATA_FOUND and TOO_MANY_ROWS are handled.
- ✓ The NO_DATA_FOUND exception occurs when a SELECT…INTO statement does not retrieve a row.
- ✓ The TOO_MANY_ROWS exception occurs when a SELECT…INTO statement retrieves more then one row.

```
SQL.; SET SERVEROUTPUT ON
SQL>DECLARE
  2      V_LAST EMPLOYEE.LNAME%TYPE
  3      V_FIRST EMPLOYEE.FNAME%TYPE;
  4      D_ID NUMBER(2) :=  &DEPARTMENT_ID;
  5   BEGIN
  6      SELECT LNAME, FNAME
  7       INTO V_LAST, V_FIRST
  8      FROM EMPLOYEE
  9      WHERE DEPID = D_ID;
 10         DBMS_OUTPUT.PUT_LINE(‘ ’);
 11         DBMS_OUTPUT.PUT_LINE(V_LAST || ‘,’ || V_FIRST);
 12   EXCEPTION
 13         WHEN NO_DATA_FOUND THEN
 14            DBMS_OUTPUT.PUT_LINE(‘NO SUCH DEPARTMENT WITH
 15                                                  EMPLOYEE’);
 16         WHEN TOO_MANY_ROWS THEN
 17            DBMS_OUTPUT.PUT_LINE(‘MORE THAN ONE EMPLOYEE
 18                                                IN DEPT’ || D_ID);
 19   END;
 20   /
```

Handling Named Exceptions (Output)

```
Enter value for department_id: 10
MORE THAN ONE EMPLOYEE IN DEPT10

PL/SQL procedure successfully completed.

SQL>/
Enter value for department_id: 50
NO SUCH DEPARTMENT WITH EMPLOYEES

PL/SQL procedure successfully completed.

SQL>/
Enter value for department_id: 40
Houston, Larry

PL/SQL procedure successfully completed.

SQL>
```

2. Nonpredefined Oracle Server Exceptions:

- ✓ A nonpredefined Oracle server exception has an attached Oracle error code, but it is not named by Oracle.
- ✓ It can trap exception with a WHEN OTHERS clause or by declaring them with names in the DECLARE section.
- ✓ The declared exception is raised implicitly by Oracle, or it can raise it explicitly.

3. Pragma Exception_Init:

- ✓ PRAGMA is a compiler directive that associates an exception name with an internal Oracle error code.
- ✓ The PRAGMA directive is not processed with the execution of a PL/SQL block, but it directs the PL/SQL compiler to associate a name with the error code.
- ✓ It can use more than one PRAGMA EXCEPTION_INIT directive in DECLARE section to assign names to different error codes.
- ✓ Naming and associating are two separate statements in the declaration section.
- ✓ First, an exception name is declared are an EXCEPTION.
- ✓ Second, the declared name is associated with an internal error code returned by SQLCODE with the PRAGMA directive.
- ✓ The ***general syntax*** is,

  Exceptionname EXCEPTION;
  PRAGMA EXCEPTION_INIT (exceptionname, errornumber);

- ✓ exceptionname is user supplied and errornumber is Oracle's internal error code. The error code is a numeric literal with a negative sign (-).

4. Nonpredefined Oracle exception

Exception-Trapping functions:

- ✓ The two functions to identify the error code and error message are:

1. SQLCODE: The SQLCODE function returns a negative error code number. The number can be assigned to a variable of NUMBER type.
2. SQLERRM: The SQLERRM function returns the error message associated with the error code.
The maximum length of error message is 512 bytes. It can be assigned to a VARCHAR2-type variable.

SQLCODE and SQLERRM

```
SQL> DECLARE
  2       emp_remain EXCEPTION;
  3       PRAGMA EXCEPTION_INIT ( emp_remain, -2292);
  4       v_deptid dept.Deptid%TYPE := &p_deptnum;
  5  BEGIN
  6        DELETE FROM dept
  7        WHERE Deptid = v_deptid
  8        COMMIT;
  9   EXCEPTION
 10         WHEN emp_remain THEN
 11          DBMS_OUTPUT.PUT('DEPARTMENT' ||TO_CHAR(v_deptid));
 12          DBMS_OUTPUT.PUT('cannoto be removed – ');
 13           DBMS_OUTPUT.PUT_LINE('Employee is department');
 14  END;
 15  /
Enter value for p_deptnum:10
DEPARTMENT 10 cannoto be removed – Employee in department

PL/SQL procedure successfully completed.
SQL>/
Enter value for p_deptnum:60
PL/SQL procedure successfully completed.

SQL>
```

```
.; SET SERVEROUTPUT ON
SQL>DECLARE
  2     V_FIRST       EMPLOYEE.FNAME%TYPE
  3     V_LAST        EMPLOYEE.LNAME%TYPE;
  4     D_ID          NUMBER(2) :=  &DEPARTMENT_ID;
  5     V_CODE        NUMBER;
  6      V_MSG        VARCHAR2(255);
  7   BEGIN
  8     SELECT LNAME, FNAME
  9      INTO V_LAST, V_FIRST
 10     FROM EMPLOYEE
 11     WHERE DEPID = D_ID;
 12       DBMS_OUTPUT.PUT_LINE(' ');
 13       DBMS_OUTPUT.PUT_LINE(V_LAST || ',' || V_FIRST);
 14  EXCEPTION
 15       WHEN OTHERRS THEN
 16          V_CODE := SQLCODE;
 17          V_MSG := SQLERRM;
 18           DBMS_OUTPUT.PUT_LINE('ERROR CODE: ' || SQLCODE);
 19           DBMS_OUTPUT.PUT_LINE(SQLERRM);
 20   END;
 21   /
Enter value for department_id:10
ERROR CODE:-1422
ORA-01422: exact fetch returns more than requested number of rows

PL/SQL peocedure successfully completed.

SQL>
```

5. User-Defined Exceptions:

- ✓ PL/SQL defines three steps for exceptions.
  - ❖ **Declare** the exception in the **DECLARE** section. There is no need to use a PRAGMA directive, because there is no standard error number to associate.
  - ❖ **Raise** the exception in the execution section of the program with an explicit **RAISE** statement.
  - ❖ **Write** the handler for the exception.
- ✓ The user-defined exceptions invalid_commission and no_commission.
- ✓ The invalid_commission exception is raised when the commission value is negative.
- ✓ The no_commission exception is raised when the commission value is NULL>

User-defined exception (source)

```
SQL> DECLARE
  2         invalid_commission EXCEPTION;
  3         no_commission  EXCEPTION;
  4         v_comm employee.Commission%TYPE;
  5    BEGIN
  6        SELECT Commission
  7         INTO V_comm
  8         FROM employee
  9         WHERE EmployeeId = &emp_id;
 10         IF V_comm < 0  then
 11              RAISE invalid_commission;
 12         ELSIF v_comm IS NUMM THEN
 13              RAISE no_commission;
 14         ELSE
 15             DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_comm));
 16         END IF;
 17  EXCEPTION
 18         WHEN invalid_commission THEN
 19              DBMS_OUTPUT.PUT_LINE ('Commission is negative.');
 20         WHEN no_commission THEN
 21              DBMS_OUTPUT.PUT_LINE('Commission is value');
 22         WHEN OTHERS THEN
 23               DBMS_OUTPUT.PUT_LINE('No such ID');
 24   END;
 25    /
```

User-defined exception (output)

```
Enter value for emp_id:111
35000
PL/SQL procedure successfully completed.
SQL>/
Enter value for emp_id:123
No commission value
PL/SQL procedure successfully completed.
SQL>/
Enter value foro emp_id:546
Commission is negative.
PL/SQL procedure successfully completed.
SQL>/
Enter value foro emp_id:321
No such ID

PL/SQL procedure successfully completed.
```

RAISE_APPLICATION_ERROR Procedure:

- ✓ The **RAISE_APPLICATION_ERROR** Procedure allows displaying nonstandard error codes and user-defined error messages from a stored subprogram.
- ✓ The ***general syntax*** is,

RAISE_APPLICATION_ERROR (error_code, error_message [, TRUE/ FALSE];

- ✓ The error_code is a user-specified number between -20,000 and -20,999 and error_message is a user-supplied message that can be up to 512 bytes long.
- ✓ TRUE means; place the error on stack of other errors.
- ✓ FALSE is the default value, and it replaces all previous errors.
- ✓ For example,

```
EXCEPTION
 WHEN NO_DATA_FOUND THEN
 RAISE_APPLICATION_ERROR
     (-20001, 'Department does not exist,);
```

UNIT IV
QUESTION BANK

5MARK:

1. What are the control structures?
2. Write about the decode function?
3. What is the difference between case and searched case statement?
4. Give four differences between the basic loop and the while loop?
5. Write about the data manipulation in PL/SQL?
6. What are the selection statements in PL/SQL?
7. What are exceptions?
8. Where are they handling?
9. What is the difference between implicit cursor and explicit cursor?
10. What is the difference between predefined Oracle server exception and user-defined exception?
11. Can you use cursor attributes with implicit cursor?
12. Explain about types of exception?
13. What is the difference between non predefined Oracle server exception and user-defined exception?
14. What is a cursor FOR loop? What are its benefits?

10MARK:

1. Write about the selection structure
2. Write about the simple if with multiple conditions? What is the difference between if..then..Else…endif and if..then..elsif…end if statement?
3. Explain about the three types looping statements?
4. What actions can be performed on an explicit cursor? Give an example of each statement's use?
5. What are four cursor attributes? State their use.
6. Name the error-trapping functions. How are they useful?
7. How are the three types of exception declared, raised, and handled?
8. What is the difference between static cursor and dynamic cursor?

UNIT-V

PL/SQL COMPOSITE DATA TYPES

COMPOSITE DATA TYPES.

- The PL/SQL has composite data types, which are data types like scalar data types. The composite data type consists of groups or collections. This data types include the following:
  - **Records**
  - **Tables**
  - **VArrays**

PL/SQL RECORDS

- The PL/SQL records are similar in structure to rows in the database table. They consist of components of any scalar type, PL/SQL record type, or PL/SQL table type. These components are known as fields, and they have their own values.
- It is based on a cursor, a table's row, or a user-defined record type. A record can be explicitly declared based on a cursor or a table.

```
CURSOR Cursorname IS
    SELECT query;
Recordname CursorName%ROWTYPE;
```

- A record can also be based on another composite data type called TABLE.

Creating a PL/SQL Record:

- To create a user-defined record by the following ways:
  - Create a RECORD type
  - Declare a record with that RECORD type

General Syntax:

```
TYPE recordtypename IS RECORD
    (fieldname1 datatype|variable%TYPE|table.column%TYPE|
      table%ROWTYPE[[NOT NULL]:=|DEFAULT Expression]
    [, fieldname2…
      , FieldName3…);
recordname recordtypename;
```

Example 1:

```
TYPE employee_rectype IS RECORD
    (elast VARCHAR2(15),
    (esal NUMBER(8,2));
employee_rec employee_rectype;
```

- In the above declaration employee_rectype is the user-defined RECORD type. The record employee_rec is a record declared with the user-defined record type employee_rectype.

Example 2:

- It is also declared with the %TYPE attribute.

```
TYPE employee_rectype IS RECORD
    (eid NUMBER(4) NOT NULL:=123,
    (esal employee.salary%TYPE);
employee_rec employee_rectype;
```

- The not null constraint can be used for any field to prevent Null values, but that field must be initialized with a value.

Referencing Fields in a Record:

- The fields in a record are referenced with the record name as a qualifier.
  - **recordname.fieldname**

- The recordname and field name are joined by a dot(.)
  - **employee_rec.esal;**

Working With Records:

- To assign values to a record from columns in a row by using the SELECT statement or the FETCH statement. The order of fields in a record must match the order of columns in the row. A record can be assigned to another record if both records have the same structure.
- A record can be set to NULL, and all fields will be set to NULL.
  - **Example:** Employee_rec:=NULL;
- A record declared with %ROWTYPE has the same structure as the table's row.
  - **emp_rec employee%ROWTYPE;**
- The emp_rec assumes the structure of the EMPLOYEE table. The fields in emp_rec take their column names and their data types from the table.

Nested Records:

- To create a nested record by including a record into another record as a field. A nested record is a record used as a field in another record. The record containing another record is called the enclosing record.

Example:

```
DECLARE
    TYPE address_rectype IS RECORD
      (first VARCHAR(15),
       street VARCHAR2(25));
  TYPE all_address_rectype IS RECORD
      (home_address address_rectype,
       address_rec all_address_rectype;
```

- In the above example all_address_rectype nests address_rectype as a field type. The nesting record makes code more readable and easier to maintain.

PL/SQL TABLES

- A PL/SQL table is another composite data type. It is a single-dimensional structure with a collection of elements that store the same type of values. A table is like an array, but a table is unbounded.

Declaring a PL/SQL Table:

- A PL/SQL TABLE declaration is done in two steps:
  - Declare a PL/SQL table type with a TYPE statement. The Structure could use any of the scalar data types.
  - Declare an actual table based on the type declared in the previous step.

General Syntax:

```
TYPE tabletypename IS TABLE OF
    datatype|variablename%TYPE|tablename.columnname%TYPE
      [NOT NULL] INDEX BY BINARY_INTEGER;
```

Example:

```
TYPE deptname_table_type IS TABLE OF
        dept.DeptName%TYPE
          INDEX BY BINARY_INTEGER;
```

- To declare a table type with a scalar data type (VARCHAR2, DATE, BOOLEAN, or POSITIVE) with the declaration attribute %TYPE.
- The indexing speeds up the search process from the table. The table consists of two columns, a primary key column and a data column. The primary key is of type BINARY_INTEGER.

Referencing Table Elements/Rows:

✶The rows in a table are referenced in the same way that an element in an array is referenced.

tablename(primarykeyvalue)

deptname_table(5):='Human Resources';

Assigning values to rows in a PL/SQL Table:

✶To assign values to the rows in a table in three ways:

- **Direct Assignment**
- **Assignment in a Loop**
- **Aggregate Assignment**

Direct Assignment:

✶ To assign a value to the rows with an assignment statement. This is preferable if only a few assignments are to be made.

Assignment in a Loop:

✶ The users prefer or use any of the three PL/SQL loops to assign values to rows in a table.

Aggregate Assignment:

✶ To assign a table's values to another table, the data types of both tables must be compatible.

Built-In Table Methods:

✶ These methods are procedures or functions that provide information about a PL/SQL table.

General Syntax:

tablename.methodname[(index1[,index2])]

Built-in Method	Use
FIRST	Returns the smallest index number in a PL/SQL table.
LAST	Returns the largest index number in a PL/SQL table.
COUNT	Returns the total number of elements in a PL/SQL table.
PRIOR(n)	Returns the index number that is before index number n.
NEXT(n)	Returns the index number that is after index number n.
EXISTS(n)	Returns TRUE if index n exists in the table.
TRIM	Removes one element from end of the table.
TRIM(n)	Removes n elements from end of the table.
DELETE	Removes all elements from a PL/SQL table.
DELETE(n)	Removes the n-th element from a table.
DELETE(m,n)	Removes all elements in the range m…n from a table.
EXTEND	Appends a null element to a table.
EXTEND(n)	Appends n null elements to a table.
EXTEND(n,x)	Appends n copies of the x-th element to a table.

Example:

```
Student.DELETE(7,10);          /*delete elements 7 to 10 */
Student.EXISTS(11) THEN …      /*true, if match 11 exists */
```

Table of Records:

✶ The PL/SQL table type is declared with a data type. To use the record type as a table's data type. The %ROWTYPE declaration attribute can be used to define the record type.

Example:

✶ A PL/SQL table type based on a programmer-defined record:

```
TYPE student_record_type IS
      RECORD(stu_id NUMBER(3),
                stu_name VARCHAR2(30));
TYPE student_table_type IS TABLE OF student_record_type
      INDEX BY BINARY_INTEGER;
```

student_table student_table_type;

- A PL/SQL table type based on a database table:

TYPE employee_table_type IS TABLE OF employee%ROWTYPE
INDEX BY BINARY_INTEGER;
employee_table employee_table_type;

- The %ROWTYPE attribute is not used when the table is based on a user-defined record. Use the %ROWTYPE attribute when the table is based on a database table or a cursor.
- The fields of a PL/SQL table based on a record are referenced with the following syntax:

tablename(index).fieldname
Example: student_table(10).stu_name:='kalpana';

PL/SQL VARRAYS

- A Varray is another composite data type or collection type in PL/SQL. It stands for variable-size array.
- They are single-dimensional, bounded collections of elements with the same type. They are similar to a PL/SQL table, and each element is assigned a subscript/index starting with 1. A PL/SQL Varray declaration is done in two steps:
  - Declare a PL/SQL Varray type with a TYPE statement. The TYPE declaration includes a size to set the upper bound of a Varray. The lower bound is always one.
  - Declare an actual Varray based on the type declared in the previous step.

General Syntax:

```
DECLARE
    TYPE Varraytypename IS VARRAY(size) OF ElementType[NOT NULL];
    varrayname varraytypename;
```

Example:

```
DECLARE
    TYPE SS IS VARRAY(5) OF employee.lname%TYPE;
    SS1 SS:=SS();
```

- When a varray is declared, it is NULL. It must be initialized before referencing its elements. The **EXTEND** method is used before adding a new element to a varray. The **COUNT** method returns the number of elements, the **LIMIT** method the upper bound, the **FIRST** method the first subscript, and the **LAST** method the last subscript.
- In oracle9i, it is possible to create a collection of a collection (multilevel collection) like a varray of varrays.

Example:

```
DECLARE
    TYPE V1 IS VARRAY(3) OF NUMBER;
    TYPE V2 IS VARRAY(2) OF V1;
```

- In the above example V1 is an array, and V2 is a varray of varray V1. The varray V1 contains three elements, and varray V2 contains six elements (2.3=6). The elements of varray V1 are referenced with one subscript, but elements of varray V2 are referenced with two subscripts.

PL/SQL NAMED BLOCKS

PROCEDURES

- A procedure is a named PL/SQL program block that can perform one or more tasks. A procedure is the building block of modular programming.

General Syntax:

```
CREATE [OR REPLACE] PROCEDURE Procedurename
   [(Parameter1,[,Parameter2…])]
IS
```

[Constant/variable declarations]
BEGIN
Executable statements
[EXCEPTION
exception handling statement]
END [procedurename];

- The above syntax procedurename is a user-supplied name. The parameter list has the names of parameters passed to the procedure by the calling program as well as the information passed from the procedure to the calling program. The local constants and variables are declared after the reserved word IS.
- The executable statements are written after BEGIN and before EXCEPTION or END. There must be at least one executable statement in the body. The reserved word EXCEPTION and the exception-handling statements are optional.

Calling a Procedure:

- A call to the procedure is made through an executable PL/SQL statement. The procedure is called by specifying its name along with the list of parameters (if any) in parenthesis. The call statement ends with a semicolon.

General Syntax: procedurename[(parameter1,….)];

Procedure Header:

- The procedure definition that comes before the reserved word **IS** is called the procedure header. The procedure header contains the name of the procedure and the parameter list with data types.

Example:

```
CREATE OR REPLACE PROCEDURE mon_sal
(vsal IN employee.salary%TYPE)
```

- The parameter list in the header contains the name of a parameter along with its type.

Procedure Body:

- It contains declaration, executable, and exception-handling sections. The declaration and exception-handling sections are optional. The executable section contains action statements, and it must contain at least one.
- It stats after the reserved word IS. If there is no local declaration, IS is followed by the reserved word BEGIN. The body ends with the reserved word END.

Parameters:

- It is used to pass values back and forth from the calling environment to the oracle server. The values passed are processed and/or returned with a procedure execution. These are three types of parameters: IN, OUT, and IN OUT.

Parameter Type	Use
IN	Passes a value into the program; read-only type of value; it cannot be changed; default parameter type. **Example:** constants, literals and expressions.
OUT	Passes a value back from the program; write-only type of value; cannot assign a default value. If a program is successful value is assigned. **Example:** variable
IN OUT	Passes a value in and returns a value back; value is read from and then written to. **Example:** variable

Actual and Formal Parameters:

- The parameters passed in a call statement are called the actual parameters. The parameter names in the header of a module are called the formal parameters. The actual parameters and their matching formal parameters must have the same data type.
- In a procedure call, the parameters are passed without data types. The procedure header contains formal parameters with data types, but the size of the data type is not required.

Procedure Call: Search_emp(543, last)

Procedure Header: **PROCDURE SEARCH_EMP(EMPNO IN NUMBER, LAST OUT VARCHAR2)**

Actual and Formal Parameters

Matching Actual and Formal Parameters:

- There are two different ways in PL/SQL to link formal and actual parameters:
  - In **positional notation**, the formal parameter is linked with an actual parameter implicitly by position. Positional notation is more commonly used for parameter matching.
  - In **named notation,** the formal parameter is linked with actual parameters explicitly by name. The formal parameter and actual parameters are linked the call statement with the symbol =>.

General Syntax:

- **formalparametername => argumentvalue**

Example:

- **EMPNO => 543**

- To execute this procedure from the SQL*Plus environment (SQL> prompt) with the EXECUTE command.
  Example: SQL> EXECUTE dependent_info

Example:

```
SQL> CREATE OR REPLACE PROCEDURE SEARCH_EMP (EMPID IN NUMBER, LAST
OUT VARCHAR2, FIRST OUT VARCHAR2)
IS
BEGIN
      SELECT LNAME,FNAME INTO LAST,FIRST FROM EMPLOYEE5 WHERE
      EMPLOYEEID=EMPID;
EXCEPTION
      WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE(EMPID);
END SEARCH_EMP;
```

Procedure with Parameters

```
SQL>  DECLARE
          VLAST EMPLOYEE5.LNAME%TYPE;
          VFIRST EMPLOYEE5.FNAME%TYPE;
          VID EMPLOYEE5.EMPLOYEEID%TYPE :=&EMP_ID;
      BEGIN
          SEARCH_EMP(VID,VLAST,VFIRST);
          IF VLAST IS NOT NULL THEN
              DBMS_OUTPUT.PUT_LINE(VID);
              DBMS_OUTPUT.PUT_LINE(VLAST||'  ' ||VFIRST);
            END IF ;
       END;
```

FUNCTIONS

- A function is a named PL/SQL block. It is also a stored block. The main difference between a function and a procedure is that a function always returns a value to the calling block.

Characteristics of Functions:

- A function can be passed zero or more parameters of IN, OUT and IN OUT types.
- A function must have an explicit RETURN statement in the executable section to return a value.
- The data type of the return value must be declared in the function header.
- A function cannot be executed as a stand-alone program.

General syntax:

```
CREATE [OR REPLACE] FUNCTION functionname
        [(parameter1 [, parameter2…])]
        RETURN Datatype
IS
        [Constant |Variable declarations]
BEGIN
        executable statements
        RETURN returnvalue
[EXCEPTION
        exception_handling statements
        RETURN returnvalue]
END [functionname];
```

- The RETURN statement does not have to be the last statement in the body of a function. The body may contain more than one RETURN statement, but only one is executed with each function call.

Function Header:

- The function header comes before the reserved word **IS**. The header contains the name of the function, the list of parameters (if any), and the RETURN data type.

Function Body:

- The body of a function must contain at least one executable statement.

RETURN Data Types:

- A function can return a value with a scalar data type, such as VARCHAR2, NUMBER, BINARY_INTEGER, or BOOLEAN. It can also return a composite or complex data type, such as PL/SQL table, a PL/SQL record, a nested table, VARRAY or LOB.

Calling a Function:

- A function is called by mentioning its name along with its parameters (if any).

  Example:

  vsalary:=get_salary(&emp_id);
- In the above example, the function call, the function get_salary is called from an assignment statement with the substitution variable emp_id as its actual parameter. The function returns the employee's salary, which is assigned to the variable vsalary.

Example:

```
SQL> CREATE OR REPLACE FUNCTION GET_DEPTNAME(DEPTID1 IN NUMBER)
        RETURN VARCHAR2 IS
      VDEPTNAME VARCHAR(12);
      BEGIN
         SELECT DEPTNAME INTO VDEPTNAME FROM DEPT5 WHERE
DEPTID=DEPTID1;
         RETURN VDEPTNAME;
    END GET_DEPTNAME;
```

Functions with Parameters

Function Call:

```
   SQL> DECLARE
            VDEPTID EMPLOYEE5.DEPTID%TYPE;
            VDEPTNAME VARCHAR2(12);
            VEMPID EMPLOYEE5.EMPLOYEEID%TYPE:=&EMP_ID;
         BEGIN
```

```
SELECT DEPTID INTO VDEPTID FROM EMPLOYEE5 WHERE
EMPLOYEEID=VEMPID;
VDEPTNAME:=GET_DEPTNAME(VDEPTID);
DBMS_OUTPUT.PUT_LINE(VEMPID);
DBMS_OUTPUT.PUT_LINE(VDEPTNAME);
EXCEPTION
        WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE(VEMPID);
END;
```

- ♦ The function returns the department name back to the calling block. The calling block then prints the employee's information along with the department name.

Calling a Function from an SQL Statement:

- ♦ A stored function block can be called from an SQL statement.

Example: SELECT get_deptname(10) FROM dual;

PACKAGES

- ➲ A package is a collection of PL/SQL objects. The objects in a package are grouped within BEGIN and END blocks. A package may contain objects from the following list:
    - o **Cursors**
    - o **Scalar variables**
    - o **Composite variables**
    - o **Constants**
    - o **Exception names**
    - o **TYPE declarations for records and tables**
    - o **Procedures**
    - o **Functions**
- ➲ Oracle has many built-in packages. Example: DBMS_OUTPUT. The objects in a package can be declared as public objects, which can be referenced from outside or as private objects, which are known only to the package.
- ➲ When an object in the package is referenced for the first time, the entire package is loaded into memory. All package elements are available from that point on, because the entire package stays in memory. This one-time loading improves performance and is very useful when the functions and procedures in it are accessed frequently.

Structure of a Package:

- ➲ A package has a specification and a body. The package specification tells us how to call different modules within a package.

Package Specification:

- ➲ A package specification does not contain any code, but it does contain information about the elements of the package.
- ➲ It contains definitions of functions and procedures declarations of global or public variables, and anything else that can be declared in a PL/SQL block's declaration section. The objects in the specification section of a package are called public objects.

General Syntax:

```
CREATE [OR REPLACE] PACKAGE packagename
IS
        [constant, variable and type declarations]
        [exception declarations]
        [cursor specification]
        [function specification]
        [procedure specification]
```

END [packagename];

Example1:

```
Package team
IS players constant integer:=12;
player_on EXCEPTION;
FUNCTION team_average(points IN NUMBER, players IN NUMBER) RETURN NUMBER;
End team;
```

Example2:

```
CREATE OR REPLACE PACKAGE COURSEINFO
AS
    PROCEDURE FINDTITLE
        (ID IN COURSE.COURSEID%TYPE,
        TITLE OUT COURSE.TITLE%TYPE);
FUNCTION HASPREREG
    (ID IN COURSE.COURSEID%TYPE)
    RETURN BOOLEAN;
FUNCTION FINDPREREG
    (ID IN COURSE.COURSEID%TYPE)
    RETURN VARCHAR2;
END COURSEINFO;
/
Package Created.
```

- The package specification for the courseinfo package is shown above contains the specification of a procedure called findtitle and functions hasprereg and findprereg.

Package Body:

- It contains actual programming code for the modules described in the specification section. It also contains code for the modules not described in the specification section.
- The module code in the body without a description in the specification is called a private module, or a hidden module, and it is not visible outside the body of the package.

General Syntax:

```
Package body packagename
IS
    [variable and type declarations]
    [cursor specifications and SELECT queries]
    [header and body of functions]
    [header and body of procedures]
[BEGIN
    executable statements]
[EXCEPTION
    exception handlers]
END [packagename];
```

- To reference an object is a package use packagename.objectname notation. If you do not use dot notation to reference an object, the compilation will fail.
- Within the body of package, you do not have to use dot notation for that packages objects, but you definitely have to use dot notation to reference an object from another package.

Example:

If team.player <10 then

- There is a set of rules that you must follow in writing a package's body:

- ♦ The variables, constants, exceptions, and so on declared in the specification must not be declared again in the package body.
- ♦ The number of cursor and module definitions in the specification must match the number of cursor and module header in the body.
- ♦ Any element declared in the specification and be referenced in the body.

➲ The example for package specification and body is shown below:

```
SQL> CREATE OR REPLACE PACKAGE BODY COURSEINFO AS
     PROCEDURE FINDTITLE
        (ID IN COURSE.COURSEID%TYPE, TITLE OUT COURSE.TITLE%TYPE) IS
     BEGIN
        SELECT TITLE INTO TITLE1 FROM COURSE WHERE COURSEID=ID;
     EXCEPTION
           WHEN OTHERS THEN
              DBMS_OUTPUT.PUT_LINE(ID||'NOT FOUND');
           END FINDTITLE;
     FUNCTION HASPREREG
        (ID IN COURSE.COURSEID%TYPE) RETURN BOOLEAN IS
        VPREREG VARCHAR2(6);
     BEGIN
        SELECT PREREG INTO VPREREG FROM COURSE WHERE COURSEID=ID;
        IF VPREREG='NONE' THEN
           DBMS_OUTPUT.PUT_LINE('NO PREREQUISITE');
           RETURN FALSE;
      ELSE
           RETURN TRUE;
        END IF;
        EXCEPTION
           WHEN NO_DATA_FOUND THEN
           DBMS_OUTPUT.PUT_LINE(ID||'DOES NOT EXISTS');
                RETURN FALSE;
      END HASPREREG;
```

Package body

➲ A call is made to the procedure findtitle of the courseinfo package

```
DECLARE
        VCOURSEID COURSE.COURSEID%TYPE:='&PCOURSEID';
        VTITLE COURSE.TITLE%TYPE;
 BEGIN
     COURSEINFO.FINDTITLE(VCOURSEID,VTITLE);
        IF VTITLE IS NOT NULL THEN
                DBMS_OUTPUT.PUT_LINE(VCOURSEID||'  '||VTITLE);
     END IF;
 END;
```

➲ To use the EXECUTE command to run a package's procedure:

```
EXECUTE packagename.procedurename
```

TRIGGERS

- A **database trigger**, known simply as a **trigger**, is a PL/SQL block. It is stored in the database and is called automatically when a triggering event occurs. A user cannot call a trigger.
- The triggering event is based on a Data Manipulation Language (DML) statement, such as INSERT, UPDATE, or DELETE. It can be created to fire before or after the triggering event. The execution of a trigger is also known as **firing the trigger**.

General Syntax:

```
Create [or replace] trigger triggername
Before|After|instead of triggeringevent ON table|view
        [For each row]
        [When condition]
DECLARE
        Declaration statements
BEGIN
        Executable statements
EXCEPTION
        Exception-handling statements
END;
```

- The above syntax the CREATE is used for creating a new trigger and REPLACE helps to replace an existing trigger. The keyword REPLACE is optional, and you should only use it to modify a trigger. If a trigger already exists in one table, you cannot replace it and associate it with another table.
- A trigger is very useful in generating values for derived columns, keeping track of table access, preventing invalid entries, performing validity checks, or maintaining security.

Restrictions on Triggers:

- A trigger cannot use a Transaction Control Language (DCL) statement, such as COMMIT, ROLLBACK, or SAVEPOINT.
- A procedure or function called by a trigger cannot perform Transaction Control Language statements.
- A variable in a trigger cannot be declared with LONG or LONG RAW data type.

BEFORE Triggers:

- It is fired before execution of a DML statement. The BEFORE trigger is useful when you want to plug into some values in a new row, insert a calculated column into a new row, or validate a value in the INSERT query with a lookup in another table.

Example:

```
SQL> CREATE OR REPLACE TRIGGER BEFORETRIGGER
     BEFORE INSERT ON EMPLOYEE5
         FOR EACH ROW
     DECLARE
         VEMPID EMPLOYEE5.EMPLOYEEID%TYPE;
     BEGIN
      SELECT EMPLOYEE5_EMPLOYEEID_SEQ.NEXTVAL INTO VEMPID FROM    DUAL;
      :NEW.EMPLOYEEID:=VEMPID;
     END;
```

- It fires before a new row is inserted into a table. The 'for each row' is used such trigger is known as a row trigger. A trigger uses a pseudorecord called :NEW, which allows you to access the currently processed row. The type of record :NEW is tablename%TYPE. The columns in this :NEW record are referenced with dot notation. (:NEW.Employeeid).

- The trigger beforetrigger provides values of employeeid so you need not include those values in your INSERT statement.

Example:

SQL> insert into employee5(lname,fname,salary,deptid) values('xxx','yyy',2000,2);
SQL>select * from employee5 where lname='xxx';

After Trigger:

- An AFTER trigger fires after a DML statement is executed. It uses the built-in Boolean functions INSERTING, UPDATING and DELETING.

Example:

```
SQL> CREATE OR REPLACE TRIGGER EMPLOYEEATRIGGER
     AFTER DELETE OR UPDATE ON EMPLOYEE5
     DECLARE
              VTYPE VARCHAR2(6);
     BEGIN
              IF DELETING THEN
             VTYPE:='DELETE';
           ELSIF UPDATING THEN
             VTYPE:='UPDATE';
           END IF;
              INSERT INTO XXX VALUES('EMPLOYEE', VTYPE);
     END;
```

- In this example, we did not use a FOR EACH ROW clause. Such a trigger is known as a **statement trigger.**
- The trigger uses the transaction type based on the last DML statement. It also plugs in the user name and today's date. The information is then inserted in the xxx table.

Example1:

```
SQL> delete from employee5 where lname='yyy';
SQL> select * form xxx;
SQL> update employee5 set commission= salary* 0.10 where employeeid=547;
SQL> select * from xxx;
```

- The above shows rows inserted in the xxx table on use of DELETE and UPDATE statement by trigger.

INSTEAD of Trigger:

- The BEFORE and AFTER triggers are based on database tables. From version 8i onward, oracle provides another type of trigger called INSTEAD OF Trigger, which is not based on a table but is based on a view.
- The INSTEAD OF trigger is a row trigger. If a view is based on a SELECT query that contains set operators, group functions, GROUP BY and HAVING clauses, DISTINCT function, join, and/or a ROWNUM pesudocolumn, data manipulation is not possible through it.
- It is used to modify a table that cannot be modified through a view. This trigger fires "instead of" triggering DML statements, such as DELETE, UPDATE, or INSERT.

Example1:

```
SQL> CREATE OR REPLACE VIEW STUDFACULTY
     AS
          SELECT S.STUDENTID, S.LAST, S.FIRST,F.FACULTYID, F.NAME
          FROM STUDENT S, FACULTY F
         WHERE S.FACULTYID(+) =F.FACULTYID;
     View created
```

- The above example creates a view with select queries and an outer join.

Example2:

```
     Delete from studfaculty where facultyid=235;
     ERROR: cannot delete from view without exactly one key-preserved table.
```

- The delete statement to delete facultyid 235 is shown above return an error message. We will accomplish deletion of row by creating an INSTEAD OF trigger.

```
SQL> CREATE OR REPLACE TRIGGER FACULTYDELETE
     INSTEAD OF DELETE ON STUDFACULTY
     FOR EACH ROW
     BEGIN
          DELETE FROM FACULTY WHERE FACULTYID =:OLD.FACULTYID;
     END;
SQL> delete from studfaculty where facultyid=235;
```

Data Manipulation and the INSTEAD OF Trigger

- In the above example INSTEAD OF DELETE trigger is created on the studfaculty view. Now, when the DELETE statement is issued to delete a faculty member with the complex view, the trigger is fired, and the faculty member is deleted without any error messages.
- The use of pseudo row called :OLD in this trigger, which gets the value of FacultyId 235 from the DELETE statement that the user had issued.

DATA DICTIONARY VIEWS

1. EXPLAIN ABOUT DATA DICTIONARY VIEWS. (PART-B)

- Oracle maintains a very informative Data Dictionary. A few Data Dictionary Views are useful for getting information about stored PL/SQL blocks.
- The following are example of queries to USER_PROCEDURES (for named blocks), USER_TRIGGERS (for trigger only), USER_SOURCE (for all source codes), USER_OBJECTS (for any object), and USER_ERRORS(for current errors) views:

```
Select ObjectName, Procedure_Name FROM USER_PROCEDURES;
Select Name, Type, Line, Text FROM USER_SOURCE;
Select object_Name, Object_Type FROM USER_OBJECTS;
```

- Use the DESCRIBE command to find out the names of columns in each Data Dictionary view, and issue SELECT queries according to the information desired.

UNIT V

5 MARK:

1. Explain about PL/SQL recods.
2. Explain about PL/SQL recods.
3. Explain about PL/SQL recods.
4. Explain about procedure.
5. Explain about function.
6. Explain about package.
7. Explain about triggers.

10 MARK:

8. Explain briefly about PL/SQL composite data type.
9. Explain briefly about Pl/SQL named blocks.
10. Define trigger. Briefly about its types.