7BCA3C1 - CORE COURSE - V – DATABASE MANGEMENT SYSTEMS

Unit I

Introduction: Database System Applications – Purpose of Database Systems – View of Data– Database Languages – Relational Databases – Database Design – Object based and Semi Structured Databases – Data Storage and Querying – Database Users and Administrators– Transaction Management – Database users and Architectures – History of Database System.

Entity-Relationship Model: E-R model – constraints – E-R diagrams – E-R Design Issues – Weak Entity Sets – Extended E-R features.

Unit II

Relational Database Design: Features of good Relational Designs – Atomic Domains and First Normal Form – Decomposition using Functional Dependencies – Functional Dependency Theory – Decomposition using Functional – Decomposition using Multivalued Dependencies – more Normal forms – Database Design Process – Modeling Temporal Data.

Unit III

Database System Architecture: Centralized and Client-Server architecture – Server System Architecture – Parallel Systems – Distributed Systems – Network Types.
Parallel Databases: I/O parallelism – Interquery Parallelism – Intraquery Parallelism.
Distributed Databases: Homogeneous and Heterogeneous Databases – Distributed Data Storage – Distributed Transactions – Distributed Query Processing.

Unit IV

Schema Objects: Data Integrity – Creating and Maintaining Tables – Indexes – Sequences – Views – Users Privileges and Roles – Synonyms.

Unit V

PL/SQL: PL/SQL – Triggers – Stored Procedures and Functions – Package – Cursors –Transaction

Text Books:

1. Database System Concepts – Silberschatz Korth Sudarshan, International (6^{th} Edition) McGraw Hill Higher Education, 2011.
2. Jose A.Ramalho – Learn ORACLE 8i BPB Publications 2007

Books for Reference:

1. "Oracle 9i The complete reference", Kevin Loney and George Koch, Tata McGraw Hill, 2004.
2. "Database Management Systems", Ramakrishnan and Gehrke, McGraw Hill, Third Edition, 2003.
3. "Oracle 9i PL/SQL Programming "Scott Urman, Oracle Press, Tata McGraw Hill, 2002.

♣♣♣♣♣♣♣♣♣

UNIT I

INTRODUCTION

Database is a collection of related data and data is a collection of facts and figures that can be processed to produce information. A **database management system** stores data in such a way that it becomes easier to retrieve, manipulate, and produce information.

1.1 DATABASE SYSTEM APPLICATIONS

Applications where we use Database Management Systems are:

- **Telecom**: There is a database to keeps track of the information regarding calls made, network usage, customer details etc. Without the database systems it is hard to maintain that huge amount of data that keeps updating every millisecond.
- **Industry**: Where it is a manufacturing unit, warehouse or distribution centre, each one needs a database to keep the records of ins and outs. For example distribution centre should keep a track of the product units that supplied into the centre as well as the products that got delivered out from the distribution centre on each day; this is where DBMS comes into picture.
- **Banking System**: For storing customer info, tracking day to day credit and debit transactions, generating bank statements etc. All this work has been done with the help of Database management systems.
- **Education sector**: Database systems are frequently used in schools and colleges to store and retrieve the data regarding student details, staff details, course details, exam details, payroll data, attendance details, fees details etc. There is a hell lot amount of inter-related data that needs to be stored and retrieved in an efficient manner.
- **Online shopping**: You must be aware of the online shopping websites such as Amazon, Flipkart etc. These sites store the product information, your addresses and preferences, credit details and provide you the relevant list of products based on your query. All this involves a Database management system.

1.2 PURPOSE OF DBMS

DBMS was developed to overcome the following drawbacks of File System.

Drawbacks of File system:

- Data Isolation: Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.
- Duplication of data – Redundant data
- Dependency on application programs – Changing files would lead to change in application programs.

Advantage of DBMS over file system

There are several advantages of Database management system over file system. Few of them are as follows:

- No redundant data – Redundancy removed by data normalization
- Data Consistency and Integrity – data normalization takes care of it too
- Secure – Each user has a different set of access
- Privacy – Limited access
- Easy access to data
- Easy recovery
- Flexible

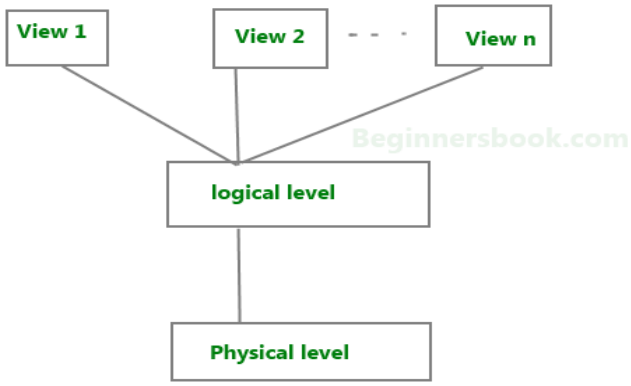
Disadvantages of DBMS:

- DBMS implementation cost is high compared to the file system
- Complexity: Database systems are complex to understand
- Performance: Database systems are generic, making them suitable for various applications. However this feature affect their performance for some applications

1.3 VIEW OF DATA

1.3.1 DATA ABSTRACTION

Database systems are made-up of complex data structures. To ease the user interaction with database, the developers hide internal irrelevant details from users. This process of hiding relevant details from user is called data abstraction.



Three Levels of data abstraction

We have three levels of abstraction:

Physical level: This is the lowest level of data abstraction. It describes how data is actually stored in database. You can get the complex data structure details at this level.

Logical level: This is the middle level of 3-level data abstraction architecture. It describes what data is stored in database.

View level: Highest level of data abstraction. This level describes the user interaction with database system.

Example: Let's say we are storing customer information in a customer table. At **physical level** these records can be described as blocks of storage (bytes, gigabytes, terabytes etc.) in memory. These details are often hidden from the programmers.

At the **logical level** these records can be described as fields and attributes along with their data types, their relationship among each other can be logically implemented. The programmers generally work at this level because they are aware of such things about database systems.

At **view level**, user just interact with system with the help of GUI and enter the details at the screen, they are not aware of how the data is stored and what data is stored; such details are hidden from them.

1.3.2 INSTANCE AND SCHEMA

Definition of schema: Design of a database is called the schema. Schema is of three types: Physical schema, logical schema and view schema.The design of a database at physical level is called **physical schema**, how the data stored in blocks of storage is described at this level.

Design of database at logical level is called **logical schema**, programmers and database administrators work at this level, at this level data can be described as certain types of data records gets stored in data structures, however the internal details such as implementation of data structure is hidden at this level (available at physical level).

Design of database at view level is called **view schema**. This generally describes end user interaction with database systems.

Definition of instance: The data stored in database at a particular moment of time is called instance of database. Database schema defines the variable declarations in tables that belong to a particular database; the value of these variables at a moment of time is called the instance of that database.

1.4 DATABASE LANGUAGES

Database languages are used for read, update and store data in a database. There are several such languages that can be used for this purpose; one of them is SQL (Structured Query Language).

Types of DBMS languages:

Data Definition Language (DDL): DDL is used for specifying the database schema.
Let's take SQL for instance to categorize the statements that comes under DDL.

- To create the database instance – CREATE
- To alter the structure of database – ALTER

- To drop database instances – DROP
- To delete tables in a database instance – TRUNCATE
- To rename database instances – RENAME

All these commands specify or update the database schema that's why they come under Data Definition language.

Data Manipulation Language (DML): DML is used for accessing and manipulating data in a database.

- To read records from table(s) – SELECT
- To insert record(s) into the table(s) – INSERT
- Update the data in table(s) – UPDATE
- Delete all the records from the table – DELETE

Data Control language (DCL): DCL is used for granting and revoking user access on a database –

- To grant access to user – GRANT
- To revoke access from user – REVOKE

1.5 DATA MODELS in DBMS

A **Data Model** is a logical structure of Database. It describes the design of database to reflect entities, attributes, relationship among data, constrains etc.

Types of Data Models:
Object based logical Models – Describe data at the conceptual and view levels.

1. E-R Model
2. Object oriented Model

Record based logical Models – Like Object based model, they also describe data at the conceptual and view levels. These models specify logical structure of database with records, fields and attributes.

1. Relational Model
2. Hierarchical Model
3. Network Model – Network Model is same as hierarchical model except that it has graph-like structure rather than a tree-based structure. Unlike hierarchical model, this model allows each record to have more than one parent record.

Physical Data Models – These models describe data at the lowest level of abstraction.

1.6 DATABASE DESIGN

The design of a DBMS depends on its architecture. It can be centralized or decentralized or hierarchical. The architecture of a DBMS can be seen as either single tier or multi-tier.

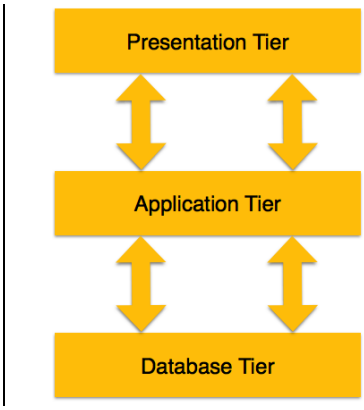
An n-tier architecture divides the whole system into related but independent **n** modules, which can be independently modified, altered, changed, or replaced.

In 1-tier architecture, the DBMS is the only entity where the user directly sits on the DBMS and uses it. Any changes done here will directly be done on the DBMS itself. It does not provide handy tools for end-users. Database designers and programmers normally prefer to use single-tier architecture.

If the architecture of DBMS is 2-tier, then it must have an application through which the DBMS can be accessed. Programmers use 2-tier architecture where they access the DBMS by means of an application. Here the application tier is entirely independent of the database in terms of operation, design, and programming.

3-tier Architecture

A 3-tier architecture separates its tiers from each other based on the complexity of the users and how they use the data present in the database. It is the most widely used architecture to design a DBMS



- **Database (Data) Tier** − At this tier, the database resides along with its query processing languages. We also have the relations that define the data and their constraints at this level.
- **Application (Middle) Tier** − At this tier reside the application server and the programs that access the database. For a user, this application tier presents an abstracted view of the database. End-users are unaware of any existence of the database beyond the application. At the other end, the database tier is not aware of any other user beyond the application tier. Hence, the application layer sits in the middle and acts as a mediator between the end-user and the database.
- **User (Presentation) Tier** − End-users operate on this tier and they know nothing about any existence of the database beyond this layer. At this layer, multiple views of the database can be provided by the application. All views are generated by applications that reside in the application tier. Multiple-tier database architecture is highly modifiable, as almost all its components are independent and can be changed independently.

1.7 DATABASE USERS AND ADMINISTRATORS

Database Users

Database users are the one who really use and take the benefits of database. There will be different types of users depending on their need and way of accessing the database.

1. **Application Programmers -** They are the developers who interact with the database by means of DML queries. These DML queries are written in the application programs like C, C++, JAVA, Pascal etc. These queries are converted into object code to communicate with the database. For example, writing a C program to generate the report of employees who are working in particular department will involve a query to fetch the data from database. It will include a embedded SQL query in the C Program.
2. **Sophisticated Users -** They are database developers, who write SQL queries to select/insert/delete/update data. They do not use any application or programs to request the database. They directly interact with the database by means of query language like SQL. These users will be scientists, engineers, analysts who thoroughly study SQL and DBMS to apply the concepts in their requirement. In short, we can say this category includes designers and developers of DBMS and SQL.
3. **Specialized Users -** These are also sophisticated users, but they write special database application programs. They are the developers who develop the complex programs to the requirement.
4. **Stand-alone Users -** These users will have stand –alone database for their personal use. These kinds of database will have readymade database packages which will have menus and graphical interfaces.
5. **Native Users -** these are the users who use the existing application to interact with the database. For example, online library system, ticket booking systems, ATMs etc which has existing application and users use them to interact with the database to fulfill their requests.

Database Administrators

The life cycle of database starts from designing, implementing to administration of it. A database for any kind of requirement needs to be designed perfectly so that it should work without any issues. Once all the design is complete, it needs to be installed. Once this step is complete, users start using the database. The database grows as the data grows in the database. When the database becomes huge, its performance comes down. Also accessing the data from the database becomes challenge. There will be unused memory in database, making the memory inevitably huge. These administration and maintenance of database is taken care by database Administrator – DBA.
A DBA has many responsibilities. A good performing database is in the hands of DBA.

- **Installing and upgrading the DBMS Servers: -** DBA is responsible for installing a new DBMS server for the new projects. He is also responsible for upgrading these servers as there are new versions comes in the market or requirement. If there is any failure in upgradation of the existing servers, he should be able revert the new changes back to the older version, thus maintaining the DBMS working. He is

also responsible for updating the service packs/ hot fixes/ patches to the DBMS servers.

- **Design and implementation: -** Designing the database and implementing is also DBA's responsibility. He should be able to decide proper memory management, file organizations, error handling, log maintenance etc for the database.
- **Performance tuning: -** Since database is huge and it will have lots of tables, data, constraints and indices, there will be variations in the performance from time to time. Also, because of some designing issues or data growth, the database will not work as expected. It is responsibility of the DBA to tune the database performance. He is responsible to make sure all the queries and programs works in fraction of seconds.
- **Migrate database servers: -** Sometimes, users using oracle would like to shift to SQL server or Netezza. It is the responsibility of DBA to make sure that migration happens without any failure, and there is no data loss.
- **Backup and Recovery: -** Proper backup and recovery programs needs to be developed by DBA and has to be maintained him. This is one of the main responsibilities of DBA. Data/objects should be backed up regularly so that if there is any crash, it should be recovered without much effort and data loss.
- **Security: -** DBA is responsible for creating various database users and roles, and giving them different levels of access rights.
- **Documentation: -** DBA should be properly documenting all his activities so that if he quits or any new DBA comes in, he should be able to understand the database without any effort. He should basically maintain all his installation, backup, recovery, security methods. He should keep various reports about database performance.

1.8 TRANSACTION MANAGEMENT

A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further.

`A's Account

Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)

B's Account

Open_Account(B)

Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)

ACID Properties

A transaction is a very small unit of a program and it may contain several lowlevel tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

- **Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency** − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability** − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

1.9 History of DBMS

1960's-1970's: The emergence of the first type of DBMS, the hierarchical DBMS. **IBM** had the first model, developed on IBM 360 and their (DBMS) was called IMS, originally it was written for the Apollo program**.** This type of DBMS was based on binary trees, where the shape was like a tree and relations were only limited between parent and child records. The benefits were numerous; less redundant data, data independence, security and integrity, which all lead to efficient searches. Nonetheless; there were some disadvantages such as; complex implementation, was hard to manage because of the absence of standards, which made it harder to handle many relationships.(see image below) (1)(3)

1960's-1970's: The emergence of the network DBMS. **Charles Bachmann** developed first DBMS at Honeywell, Integrated Data Store (IDS) then a group called CODASYL who is responsible for the creation of COBOL, had that system standardized. However; the CODASYL group invented what they call the "**CODASYL APPROACH**. Based on that approach many systems using network DBMS were developed for business use(2). In this model, each record can have multiple parents in comparison with one in the hierarchical DBMS. It is made of sets of relationships where a set represents a one to many relationship between the owner and the member. The main and unfortunate disadvantage was that the System was complex and there was difficulty in design and maintenance, it is believed that the Lack of structural independence was the main cause. (2)(3)

1970's- 1990's: The emergence of the relational DBMS on the hands of **Edgar Codd**. He worked at IBM, and he was unhappy with the navigational model of the CODASYL APPROACH. To him, a tool for searching, such as a search facility was very useful, and it

was absent . In 1970, he proposed a new approach to database construction, which made the creation of a Relational DBMS intended for Large Shared Data Banks, possible and easy to grab (3). Moreover; This was a new system for entering data and working with big databases, where the idea was to use a table of records. All tables will be then linked by either one to one relationships, one to many, or many to many(2). when elements took space and were not useful, it was easy to remove them from the original table, and all the other "entries" in other tables linked to this record were removed. Worth mentioning, is that two initial projects were launched, the **R** program at IBM, and **INGRES** program at the university of California. In 1985, the object oriented DBMS was developed, but it did not have any booming commercial profit because of the high unjustified costs to change systems, and format. In 1990, the DBMS took on a new object oriented approach joint with relational DBMS . In this approach, text, multimedia, internet and web use in conjunction with DBMS were available and possible.(3)(1)

Past and present

- In the early years of computing, a **punch card** was used in unit record machines for input, data storage and processing this data. Data was entered offline and for both data, and computer programs input. This input method is similar to voting machines nowadays (2). This was the only method, where it was fast to enter data, and retrieve it, but not to manipulate or edit it.
  After that era, there was the introduction of the file type entries for data, then the DBMS as hierarchical, network, and relational.

2. ENTITY RELATIONSHIP MODEL

2.1 ER MODEL BASIC CONCEPTS

The ER model defines the conceptual view of a database. It works around real-world entities and the associations among them. At view level, the ER model is considered a good option for designing databases.

Entity

An entity can be a real-world object, either animate or inanimate, that can be easily identifiable. For example, in a school database, students, teachers, classes, and courses offered can be considered as entities. All these entities have some attributes or properties that give them their identity.An entity set is a collection of similar types of entities. An entity set may contain entities with attribute sharing similar values. For example, a Students set may contain all the students of a school; likewise a Teachers set may contain all the teachers of a school from all faculties. Entity sets need not be disjoint.

Attributes

Entities are represented by means of their properties, called **attributes**. All attributes have values. For example, a student entity may have name, class, and age as attributes.There exists a domain or range of values that can be assigned to attributes. For example, a student's name cannot be a numeric value. It has to be alphabetic. A student's age cannot be negative, etc.

Types of Attributes

- **Simple attribute** − Simple attributes are atomic values, which cannot be divided further. For example, a student's phone number is an atomic value of 10 digits.
- **Composite attribute** − Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first_name and last_name.
- **Derived attribute** − Derived attributes are the attributes that do not exist in the physical database, but their values are derived from other attributes present in the database. For example, average_salary in a department should not be saved directly in the database, instead it can be derived. For another example, age can be derived from data_of_birth.
- **Single-value attribute** − Single-value attributes contain single value. For example − Social_Security_Number.
- **Multi-value attribute** − Multi-value attributes may contain more than one values. For example, a person can have more than one phone number, email_address, etc.

These attribute types can come together in a way like −

- simple single-valued attributes
- simple multi-valued attributes
- composite single-valued attributes
- composite multi-valued attributes

Entity-Set and Keys

Key is an attribute or collection of attributes that uniquely identifies an entity among entity set.

For example, the roll_number of a student makes him/her identifiable among students.

- **Super Key** − A set of attributes (one or more) that collectively identifies an entity in an entity set.
- **Candidate Key** − A minimal super key is called a candidate key. An entity set may have more than one candidate key.
- **Primary Key** − A primary key is one of the candidate keys chosen by the database designer to uniquely identify the entity set.

Relationship

The association among entities is called a relationship. For example, an employee **works_at** a department, a student **enrolls** in a course. Here, Works_at and Enrolls are called relationships.

Relationship Set

A set of relationships of similar type is called a relationship set. Like entities, a relationship too can have attributes. These attributes are called **descriptive attributes**.

Degree of Relationship

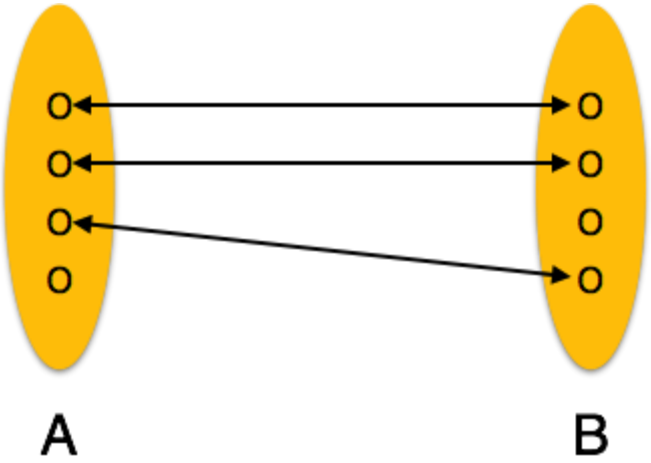
The number of participating entities in a relationship defines the degree of the relationship.

- Binary = degree 2
- Ternary = degree 3
- n-ary = degree

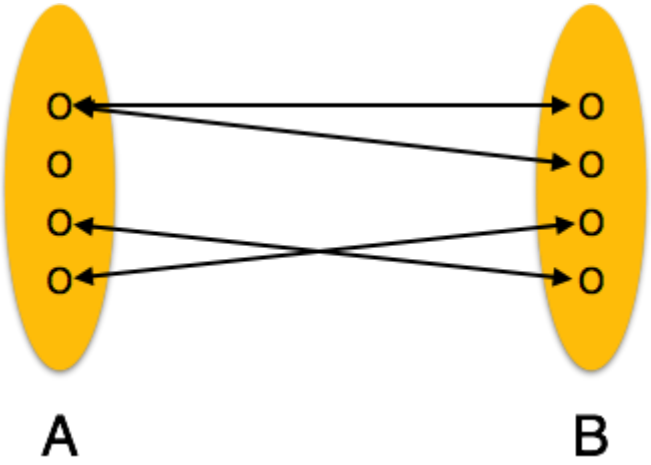
2.2 E-R CONSTRAINTS: Mapping Cardinalities

Cardinality defines the number of entities in one entity set, which can be associated with the number of entities of other set via relationship set.

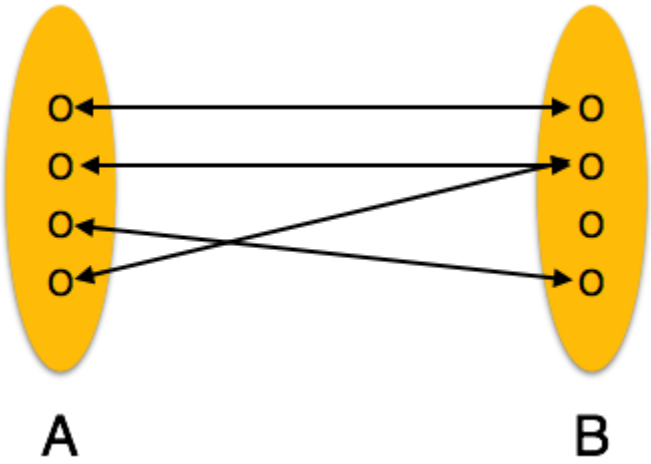
- **One-to-one** − One entity from entity set A can be associated with at most one entity of entity set B and vice versa.



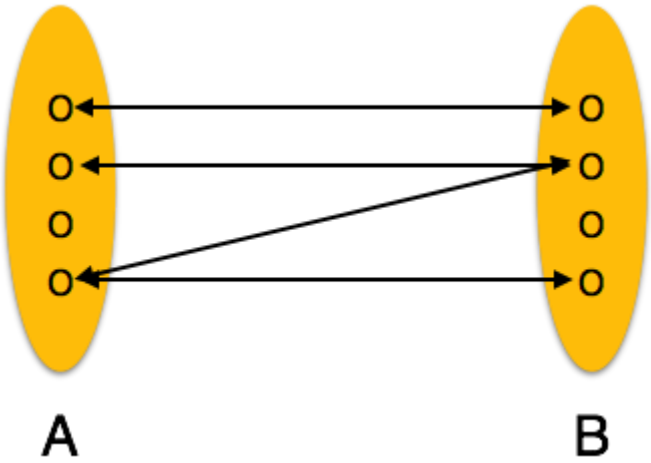
- **One-to-many** − One entity from entity set A can be associated with more than one entities of entity set B however an entity from entity set B, can be associated with at most one entity.



- **Many-to-one** − More than one entities from entity set A can be associated with at most one entity of entity set B, however an entity from entity set B can be associated with more than one entity from entity set A.



- **Many-to-many** − One entity from A can be associated with more than one entity from B and vice versa.



2.3 E-R DIAGRAMS

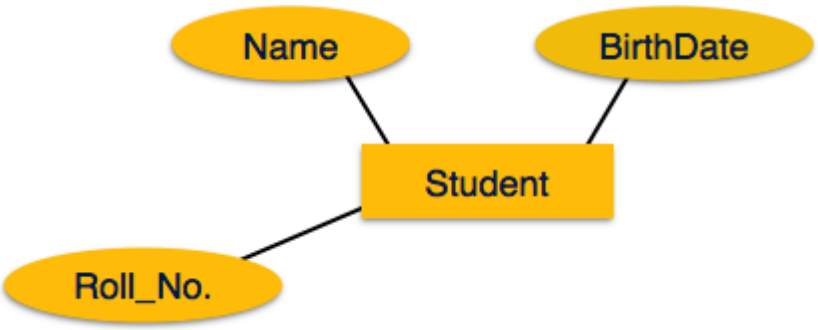
ER Model is represented by means of an ER diagram. Any object, for example, entities, attributes of an entity, relationship sets, and attributes of relationship sets, can be represented with the help of an ER diagram.

Entity

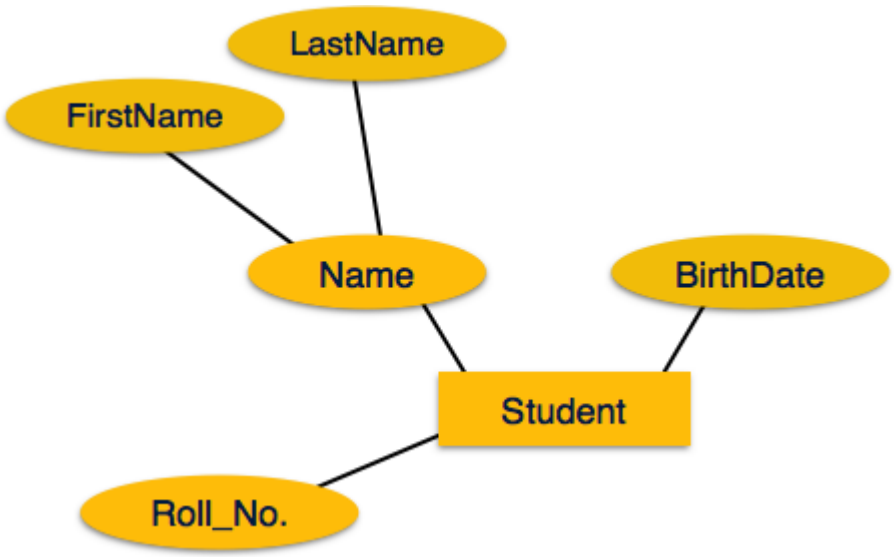
Entities are represented by means of rectangles. Rectangles are named with the entity set they represent.

Attributes

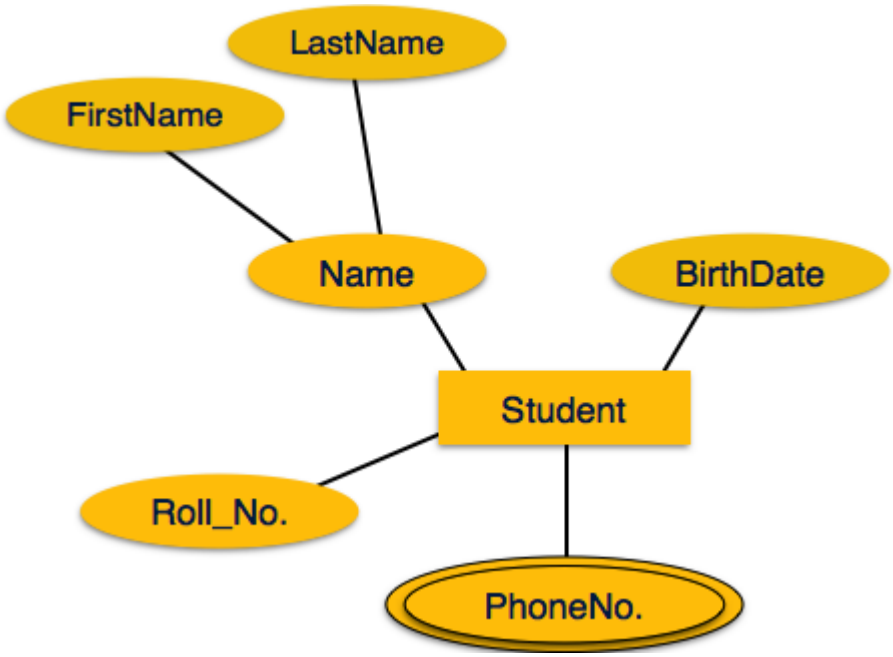
Attributes are the properties of entities. Attributes are represented by means of ellipses. Every ellipse represents one attribute and is directly connected to its entity (rectangle).



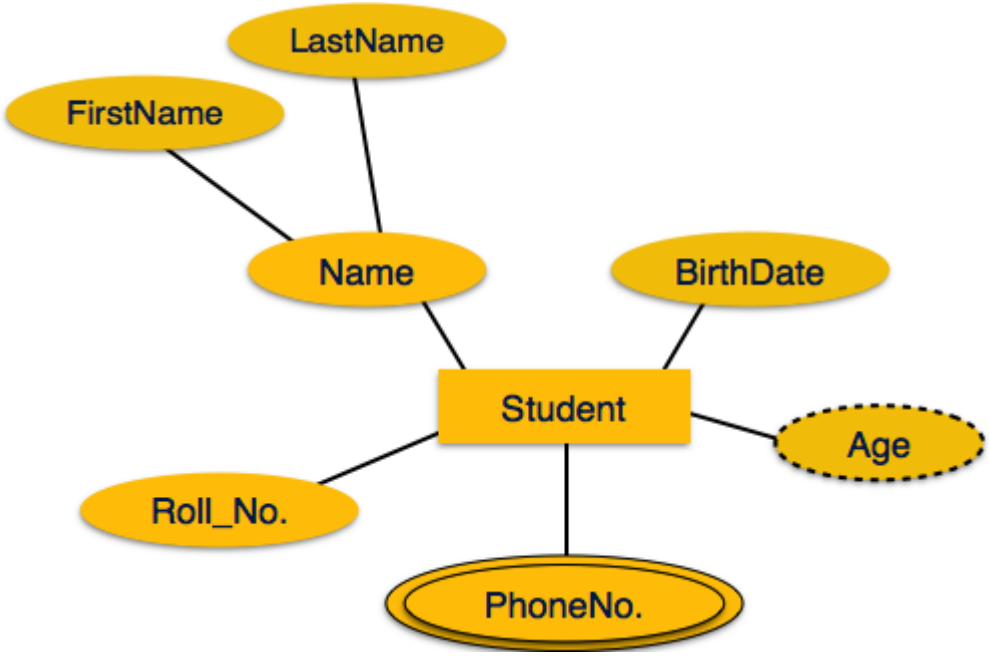
If the attributes are **composite**, they are further divided in a tree like structure. Every node is then connected to its attribute. That is, composite attributes are represented by ellipses that are connected with an ellipse.



Multivalued attributes are depicted by double ellipse.



Derived attributes are depicted by dashed ellipse.



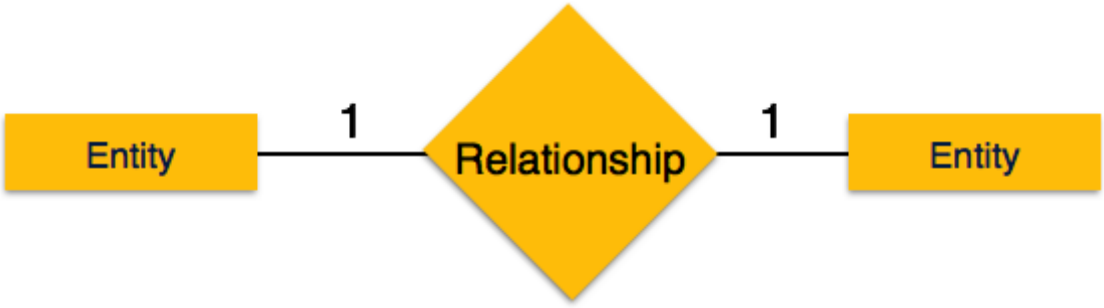
Relationship

Relationships are represented by diamond-shaped box. Name of the relationship is written inside the diamond-box. All the entities (rectangles) participating in a relationship, are connected to it by a line.

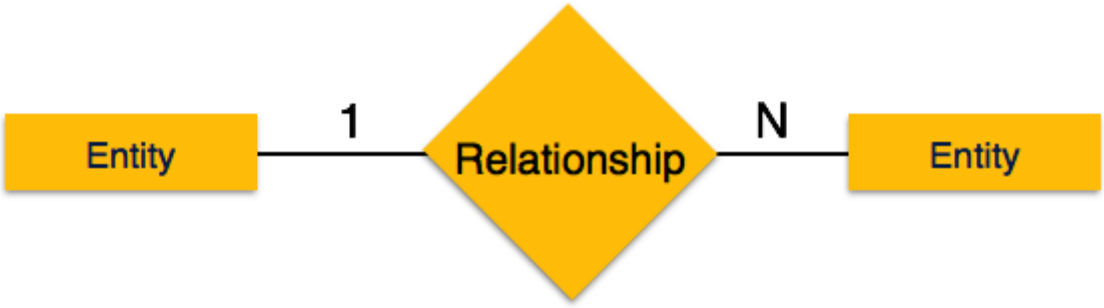
Binary Relationship and Cardinality

A relationship where two entities are participating is called a **binary relationship**. Cardinality is the number of instance of an entity from a relation that can be associated with the relation.

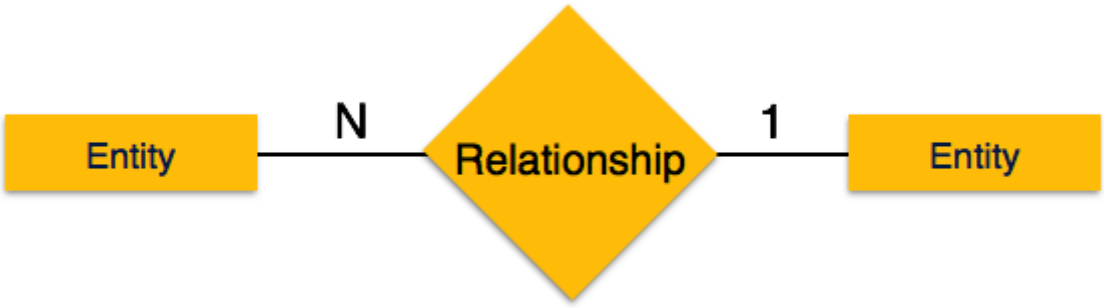
- **One-to-one** − When only one instance of an entity is associated with the relationship, it is marked as '1:1'. The following image reflects that only one instance of each entity should be associated with the relationship. It depicts one-to-one relationship.



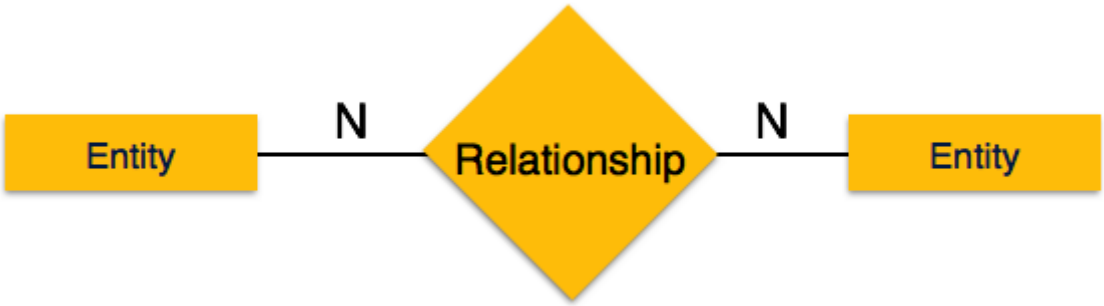
- **One-to-many** − When more than one instance of an entity is associated with a relationship, it is marked as '1:N'. The following image reflects that only one instance of entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts one-to-many relationship.



- **Many-to-one** − When more than one instance of entity is associated with the relationship, it is marked as 'N:1'. The following image reflects that more than one instance of an entity on the left and only one instance of an entity on the right can be associated with the relationship. It depicts many-to-one relationship.

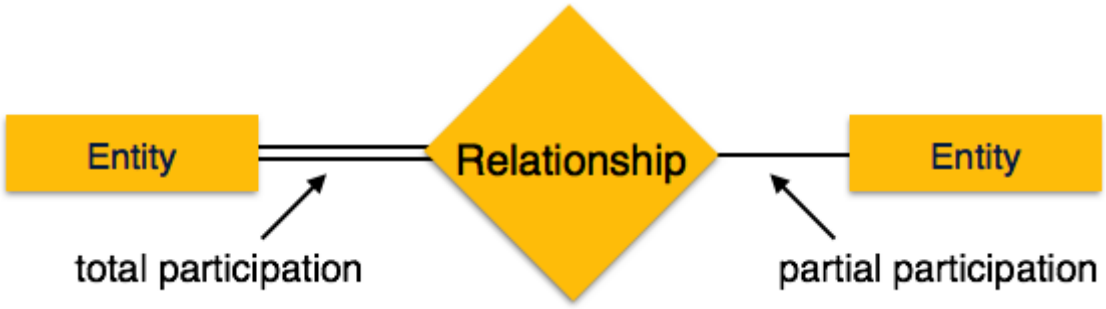


- **Many-to-many** − The following image reflects that more than one instance of an entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts many-to-many relationship.



Participation Constraints

- **Total Participation** − Each entity is involved in the relationship. Total participation is represented by double lines.
- **Partial participation** − Not all entities are involved in the relationship. Partial participation is represented by single lines.



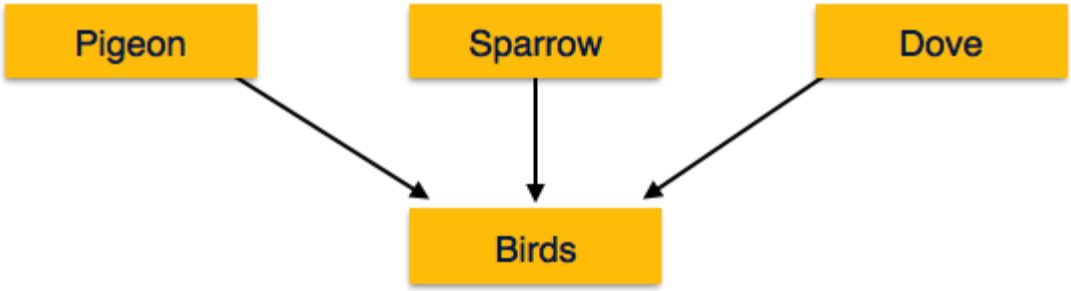
2.4 EXTENDED E-R FEATURES

The ER Model has the power of expressing database entities in a conceptual hierarchical manner. As the hierarchy goes up, it generalizes the view of entities, and as we go deep in the hierarchy, it gives us the detail of every entity included.

Going up in this structure is called **generalization**, where entities are clubbed together to represent a more generalized view. For example, a particular student named Mira can be generalized along with all the students. The entity shall be a student, and further, the student is a person. The reverse is called **specialization** where a person is a student, and that student is Mira.

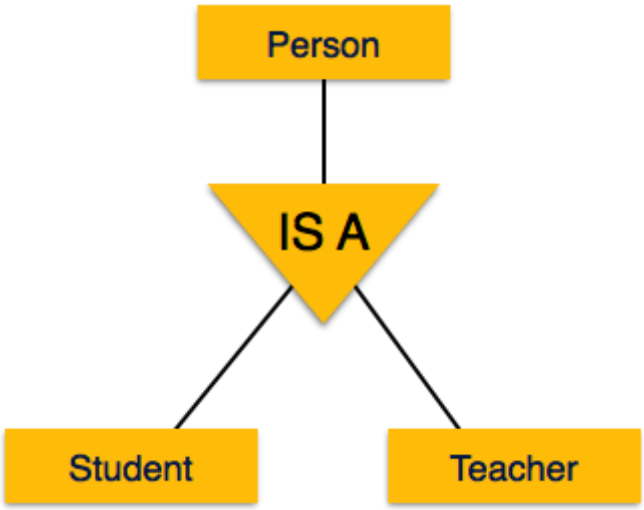
Generalization

As mentioned above, the process of generalizing entities, where the generalized entities contain the properties of all the generalized entities, is called generalization. In generalization, a number of entities are brought together into one generalized entity based on their similar characteristics. For example, pigeon, house sparrow, crow and dove can all be generalized as Birds.



Specialization

Specialization is the opposite of generalization. In specialization, a group of entities is divided into sub-groups based on their characteristics. Take a group 'Person' for example. A person has name, date of birth, gender, etc. These properties are common in all persons, human beings. But in a company, persons can be identified as employee, employer, customer, or vendor, based on what role they play in the company.

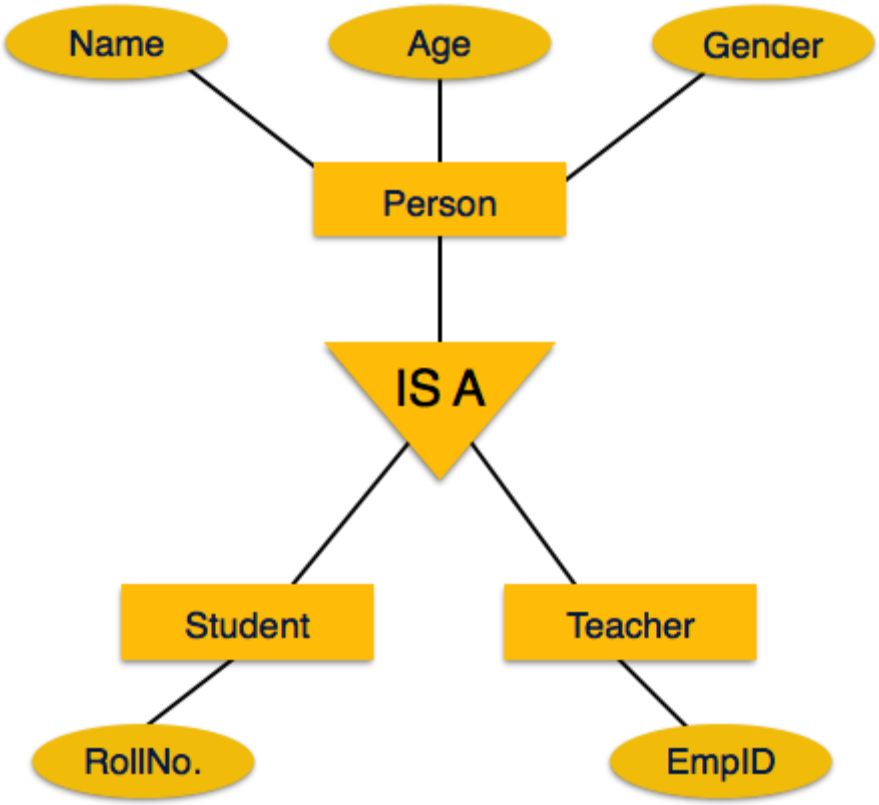


Similarly, in a school database, persons can be specialized as teacher, student, or a staff, based on what role they play in school as entities.

Inheritance

We use all the above features of ER-Model in order to create classes of objects in object-oriented programming. The details of entities are generally hidden from the user; this process known as **abstraction**.

Inheritance is an important feature of Generalization and Specialization. It allows lower-level entities to inherit the attributes of higher-level entities.



For example, the attributes of a Person class such as name, age, and gender can be inherited by lower-level entities such as Student or Teacher.

UNIT – II

3. RELATIONAL DATABASE DESIGN

INTRODUCTION

Relational database was proposed by Edgar Codd (of IBM Research) around 1969. It has since become the dominant database model for commercial applications (in comparison with other database models such as hierarchical, network and object models). Today, there are many commercial Relational Database Management System (RDBMS), such as Oracle, IBM DB2 and Microsoft SQL Server. There are also many free and open-source RDBMS, such as MySQL, mSQL (mini-SQL) and the embedded JavaDB (Apache Derby).

A relational database organizes data in tables (or relations). A table is made up of rows and columns. A row is also called a record (or tuple). A column is also called a field (or attribute). A database table is similar to a spreadsheet. However, the relationships that can be created among the tables enable a relational database to efficiently store huge amount of data, and effectively retrieve selected data.

3.1 FEATURES OF GOOD REALTIONAL DB DESIGN

A language called SQL (Structured Query Language) was developed to work with relational databasesData in the relational database must be represented in tables, with values in columns within rows.

Data within a column must be accessible by specifying the table name, the column name, and the value of the primary key of the row.

The DBMS must support missing and inapplicable information in a systematic way, distinct from regular values and independent of data type.

The DBMS must support an active on-line catalogue.

The DBMS must support at least one language that can be used independently and from within programs, and supports data definition operations, data manipulation, constraints, and transaction management.

Views must be updatable by the system.

The DBMS must support insert, update, and delete operations on sets.

The DBMS must support logical data independence.

The DBMS must support physical data independence.

Integrity constraints must be stored within the catalogue, separate from the application.

The DBMS must support distribution independence. The existing application should run when the existing data is redistributed or when the DBMS is redistributed.

If the DBMS provides a low level interface (row at a time), that interface cannot bypass the integrity constraints.

3.2 ATOMIC DOMAIN AND FIRST NORMAL FORM

A database is in first normal form if it satisfies the following conditions:

- Contains only atomic values
- There are no repeating groups

An atomic value is a value that cannot be divided. For example, in the table shown below, the values in the [Color] column in the first row can be divided into "red" and "green", hence [TABLE_PRODUCT] is not in 1NF.

A repeating group means that a table contains two or more columns that are closely related. For example, a table that records data on a book and its author(s) with the following columns: [Book ID], [Author 1], [Author 2], [Author 3] is not in 1NF because [Author 1], [Author 2], and [Author 3] are all repeating the same attribute.

1st Normal Form Example

How do we bring an un normalized table into first normal form? Consider the following example:

TABLE_PRODUCT

Product ID	Color	Price
1	red, green	15.99
2	yellow	23.99
3	green	17.50
4	yellow, blue	9.99
5	red	29.99

This table is not in first normal form because the [Color] column can contain multiple values. For example, the first row includes values "red" and "green."

To bring this table to first normal form, we split the table into two tables and now we have the resulting tables:

TABLE_PRODUCT_PRICE

Product ID	Price
1	15.99
2	23.99
3	17.50
4	9.99
5	29.99

TABLE_PRODUCT_COLOR

Product ID	Color
1	red
1	green
2	yellow
3	green
4	yellow
4	blue
5	red

Now first normal form is satisfied, as the columns on each table all hold just one value.

3.3 FUNCTIONAL DEPENDENCY THEORY – DECOMPOSITION USING FUNCTIONAL DEPENDENCY

In relational database theory, a **functional dependency** is a **constraint** between two sets of attributes in a relation from a database. In other words, functional dependency is a constraint that describes the relationship between attributes in a relation.

Employee Department Model

A classic example of functional dependency is the employee, department model. The following table

Employee ID	Employee Name	Department ID	Department Name
0001	John Doe	1	Human Resources
0002	Jane Doe	2	Marketing
0003	John Smith	1	Human Resources
0004	Jane Goodall	3	Sales

This case represents an example where multiple functional dependencies are embedded in a single representation of data. Note that because an employee can only be a member of one department, the unique ID of that employee determines the department.

- Employee ID → Employee Name
- Employee ID → Department ID

In addition to this relationship, the table also has a functional dependency through a non-key attribute

- Department ID → Department Name

This example demonstrates that even though there exists a FD Employee ID → Department ID - the employee ID would not be a logical key for determination of the department ID. The process of normalization of the data would recognize all FD's and

allow the designer to construct tables and relationships that are more logical based on the data.

Properties and axiomatization of functional dependencies

Given that *X*, *Y*, and *Z* are sets of attributes in a relation *R*, one can derive several properties of functional dependencies. Among the most important are the following, usually called Armstrong's axioms:

Reflexivity: If *Y* is a subset of *X*, then $X \rightarrow Y$

- **Augmentation**: If $X \rightarrow Y$, then $XZ \rightarrow YZ$
- **Transitivity**: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

"Reflexivity" can be weakened to just , i.e. it is an actual axiom, where the other two are proper inference rules, more precisely giving rise to the following rules of syntactic consequence:These three rules are a sound and complete axiomatization of functional dependencies. This axiomatization is sometimes described as finite because the number of inference rules is finite,[5] with the caveat that the axiom and rules of inference are all schemata, meaning that the *X*, *Y* and *Z* range over all ground terms (attribute sets).

From these rules, we can derive these secondary rules:[3]

- **Union**: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
- **Decomposition**: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
- **Pseudotransitivity**: If $X \rightarrow Y$ and $WY \rightarrow Z$, then $WX \rightarrow Z$

The union and decomposition rules can be combined in a logical equivalence stating that $X \rightarrow YZ$, holds iff $X \rightarrow Y$ and $X \rightarrow Z$. This is sometimes called the splitting/combining rule.[6]

Another rule that is sometimes handy is:[7]

- **Composition**: If $X \rightarrow Y$ and $Z \rightarrow W$, then $XZ \rightarrow YW$

Closure of Functional Dependency

The closure is essentially the full set of values that can be determined from a set of known values for a given relationship using its functional dependencies. You use Armstrong's axioms to provide a proof - i.e. Reflexivity, Augmentation, Transitivity.

Given and a set of FD's that holds in : The closure of in (denoted $^{+}$) is the set of all FD's in that are logically implied by

Closure of a set of attributes

Closure of a set of attributes X with respect to is the set X^+ of all attributes that are functionally determined by X using $^+$.

Example

Imagine the following list of FD's. We are going to calculate a closure for A from this relationship.

1. $A \rightarrow B$
2. $B \rightarrow C$
3. $AB \rightarrow D$

The closure would be as follows:

a) A → A (by Armstrong's reflexivity)
b) A → AB (by 1. and (a))
c) A → ABD (by (b), 3, and Armstrong's transitivity)
d) A → ABCD (by (c), and 2)

The closure is therefore A → ABCD. By calculating the closure of A, we have validated that A is also a good candidate key as its closure is every single data value in the relationship.

3.4 DECOMPOSITION USING MULTIVALUED DEPENDENCY

In database theory, a **multivalued dependency** is a full constraint between two sets of attributes in a relation. In contrast to the functional dependency, the multivalued **dependency** requires that certain tuples be present in a relation. Therefore, a multivalued dependency is a special case of *tuple-generating dependency*. The multivalued dependency plays a role in the 4NF database normalization.

A multivalued dependency is a special case of a join dependency, with only two sets of values involved, i.e. it is a binary join dependency. A multivalued dependency exists when there are at least 3 attributes (like X,Y and Z) in a relation and for value of X there is a well defined set of values of Y and a well defined set of values of Z. However, the set of values of Y is independent of set Z and vice versa.

Multivalued Dependencies

A consequence of 1NF: no multivalued attributes. Now have multiple entries to record the multivalued dependency.If you have two MVDs in a relation (several dependents and several projects for an employee), all combinations of project and dependents have to be listed.

In general, in **R(A,B,C)** if each A values has associated with it a set of B values and a set of C values such that the B and C values are independent of each other, then A multi-determines B and A multi-determines C.Multi-valued dependencies occur in pairs.

Example: **JointAppoint(facId, dept, committee)** assuming a faculty member can belong to more than one department and belong to more than one committee.Table must list all combinations of values of department and committee for each facId

MVD Formal definition

Let R be a relation scheme and X and Y are subsets of R,
X ->> Y
holds if for all pairs of tuples s and t in r, such that
s[X] = t[X] there also exist tuples u and v where
s[X] = t[X] = u[X] = v[X]
s[Y] = u[Y]
t[R-XY] = u[R-XY]
s[R-XY] = v[R-XY]
t[Y] = v[Y]

MVD Example

Course ->> Instructor

Course ->> Text

Course(Y)	Instructor(X)	Text(R-XY)
Prin of IT	Kruse	Intro to IT
Prin of IT	Wright	Intro to IT
CS1	Thomas	Intro to Java
CS1	Thomas	CS Theory
CS2	Rhodes	Java Data Structures
CS2	Rhodes	Unix
CS2	Kruse	Java Data Structures

3.5 NORMAL FORMS

Normalization rule are divided into following normal form.

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF

First Normal Form (1NF)

As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.

The **Primary key** is usually a single column, but sometimes more than one column can be combined to create a single primary key. For example consider a table which is not in First normal form

Student Table :

Student	Age	Subject
Adam	15	Biology, Maths
Alex	14	Maths
Stuart	17	Maths

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

Student Table following 1NF will be :

Student	Age	Subject
Adam	15	Biology
Adam	15	Maths
Alex	14	Maths
Stuart	17	Maths

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

Second Normal Form (2NF)

As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column depends only on one part of the concatenated key, then the table fails **Second normal form**.

In example of First Normal Form there are two rows for Adam, to include multiple subjects that he has opted for. While this is searchable, and follows First normal form, it is an inefficient use of space. Also in the above Table in First Normal Form, while the candidate key is {**Student**, **Subject**}, **Age** of Student only depends on Student column, which is incorrect as per Second Normal Form. To achieve second normal form, it would be helpful to split out the subjects into an independent table, and match them up using the student names as foreign keys.

New Student Table following 2NF will be :

Student	Age
Adam	15
Alex	14
Stuart	17

In Student Table the candidate key will be **Student** column, because all other column i.e **Age** is dependent on it.

New Subject Table introduced for 2NF will be :

Student Subject
Adam Biology
Adam Maths
Alex Maths
Stuart Maths

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

Third Normal Form (3NF)

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

Student_Detail Table :

Student_id Student_name DOB Street city State Zip

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

New Student_Detail Table :

Student_id Student_name DOB Zip

Address Table :

Zip Street city state

The advantage of removing transtive dependency is,

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

Boyce and Codd Normal Form (BCNF)

Boyce and Codd Normal Form is a higher version of the Third Normal form. This form deals with certain type of anamoly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:

- R must be in 3rd Normal Form
- and, for each functional dependency (X -> Y), X should be a super Key.

Consider the following relationship : **R (A,B,C,D)**

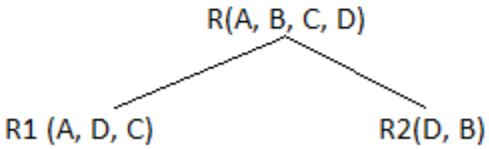
and following dependencies :

A -> BCD
BC -> AD
D -> B

Above relationship is already in 3rd NF. Keys are **A** and **BC.**

Hence, in the functional dependency, **A -> BCD**, A is the super key.
in second relation, **BC -> AD**, BC is also a key.
but in, **D -> B**, D is not a key.

Hence we can break our relationship R into two relationships **R1** and **R2.**



Breaking, table into two tables, one with A, D and C while the other with D and B.

3.6 MODELING TEMPORAL DATA

Temporal database stores data relating to time instances. It offers temporal data types and stores information relating to past, present and future time.More specifically the temporal aspects usually include valid time and transaction time. These attributes can be combined to form bitemporal data.

- **Valid time** is the time period during which a fact is true in the real world.
- **Transaction time** is the time period during which a fact stored in the database was known.
- **Bitemporal data** combines both Valid and Transaction Time.

Features

Temporal databases support managing and accessing temporal data by providing one or more of the following features:

- A time period datatype, including the ability to represent time periods with no end (infinity or forever)
- The ability to define valid and transaction time period attributes and bitemporal relations
- System-maintained transaction time
- Temporal primary keys, including non-overlapping period constraints
- Temporal constraints, including non-overlapping uniqueness and referential integrity
- Update and deletion of temporal records with automatic splitting and coalescing of time periods
- Temporal queries at current time, time points in the past or future, or over durations
- Predicates for querying time periods, often based on Allen's interval relations

UNIT III

4. DATABASE SYSTEM STRUCTURES

4.1 Centralized and client/ server architecture

Centralized database system: The centralized database system consist of a single processor together with it associated data storage devices and other peripherals. It is physically confined to a single location. The data can be accessed from the multiple sites with the use of a computer network while the database is maintained at the central site.

Disadvantages of centralized database system:

- When the central site computer or database system goes down, then everyone is blocked from using the system until the system comes back.
- Communication costs from the terminals to the central site can be expensive.

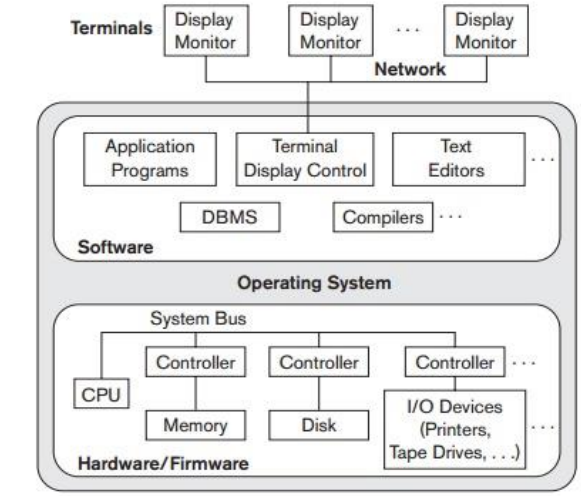
Client server dbms:- Client / server architecture of database systems has two logical components namely client and server. Client is generally personal computer or workstations whereas server is large workstations, mini range computer system or a mainframe computer system. The application and toals of DBMS run on one or more client platforms, while the DBMS softwares reside on the server. The server computer is called front end. These server and client computer are connected into a network. The application and tools act as client of the DBMS making requests for its services; client/ server architecture handles the graphical user interface (GUI) and does computations and other programming of interest to the end user.

ADVANTAGES:

1. CLIENT / SERVER system has less expensive platforms to support applications that had previously been running only on large and expensive mini or main frame computers.
2. Client/ server environment facilities in more productive work by the users and making better use of existing data.
3. It is more flexible as compared to the centralized systems.
4. Responsive time 7 throughput is high.
5. A single dg(on server) can be shared across several distinct client (application) system.

DISADVANTAGE:

1. Programming cost is high in client / server environment, particularly in initial phases.
2. There's a lack of management tools for diagnosis, performance monitoring and tuning and security control, for the DBMS client and OS and networking environments.



4.2 Parallel Database System

Parallel database system architecture consists of a multiple Central Processing Units (CPUs) and data storage disk in parallel. Hence, they improve processing and Input/Output (I/O) speeds. Parallel database systems are used in the application that have to query extremely large databases or that have to process an extremely large number of transactions per second.

Advantages of a Parallel Database System

• Parallel database systems are very useful for the applications that have to query extremely large databases (of the order of terabytes, for example, 1012 bytes) or that have to process an extremely large number of transactions per second (of the order of thousands of transactions per second).

• In a parallel database system, the throughput (that is, the number of tasks that can be completed in a given time interval) and the response time (that is, the amount of time it takes to complete a single task from the time it is· submitted) are very high.

Disadvantages of a Parallel Database System

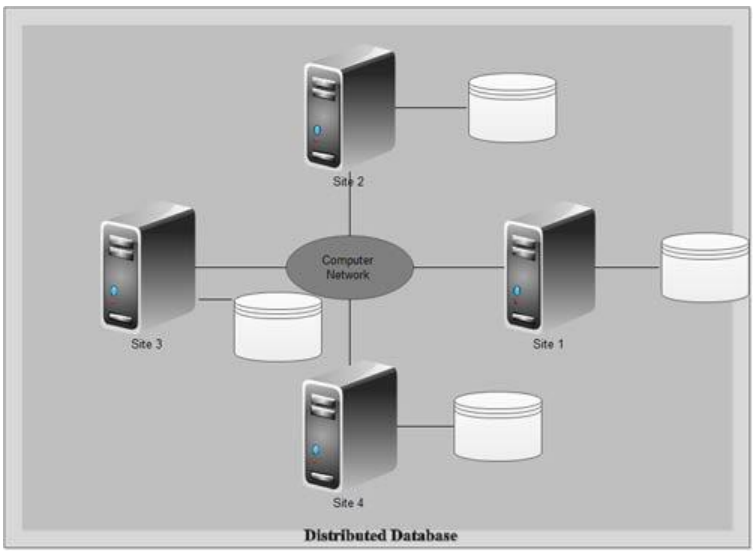
• In a parallel database system, there· is a startup cost associated with initiating a single process and the startup-time may overshadow the actual processing time, affecting speedup adversely.

• Since process executing in a parallel system often access shared resources, a slowdown may result from interference of each new process as it completes with existing processes for commonly held resources, such as shared data storage disks, system bus and so on.

4.3 Distributed Database System

A logically interrelated collection of shared data physically distributed over a computer network is called as distributed database and the software system that permits the management of the distributed database and makes the distribution transparent to users is called as Distributed DBMS.

It consists of a single logical database that is split into a number of fragments. Each fragment is stored on one or more computers under the control of a separate DBMS, with the computers connected by a communications network. As shown, in distributed database system, data is spread across a variety of different databases. These are managed by a variety of different DBMS software running on a variety of different operating systems. These machines are spread (or distributed) geographically and connected together by a variety of communication networks.

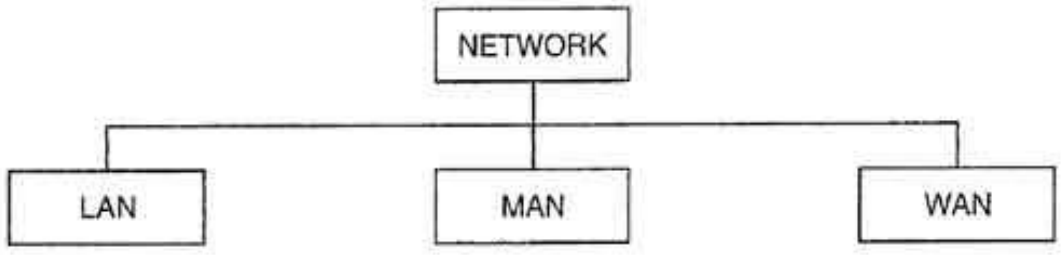


Advantages of Distributed Database System

• Distributed database architecture provides greater efficiency and better performance.
• A single database (on server) can be shared across several distinct client (application) systems.
• As data volumes and transaction rates increase, users can grow the system incrementally.
• It causes less impact on ongoing operations when adding new locations.
• Distributed database system provides local autonomy.

4.4 NETWORK TYPES

Computer Networks fall into three classes regarding the size, distance and the structure namely: LAN (Local Area Network), MAN (Metropolitan Area Network), WAN (Wide Area Network).



Types of Networks

LAN (Local Area Network

Now-a-days LANs are being installed using wireless tech. A **Local Area Network** is a privately owned computer **network** covering a **small Networks geographical area,** like a home, office, or groups of buildings e.g. a school Network. A LAN is used to connect the computers and other network devices so that the devices can communicate with each other to share the resources. The resources to be shared can be a hardware device like printer, software like an application program or data. The size of LAN is usually small. The various devices in LAN are connected to central devices called Hub or Switch using a cable.

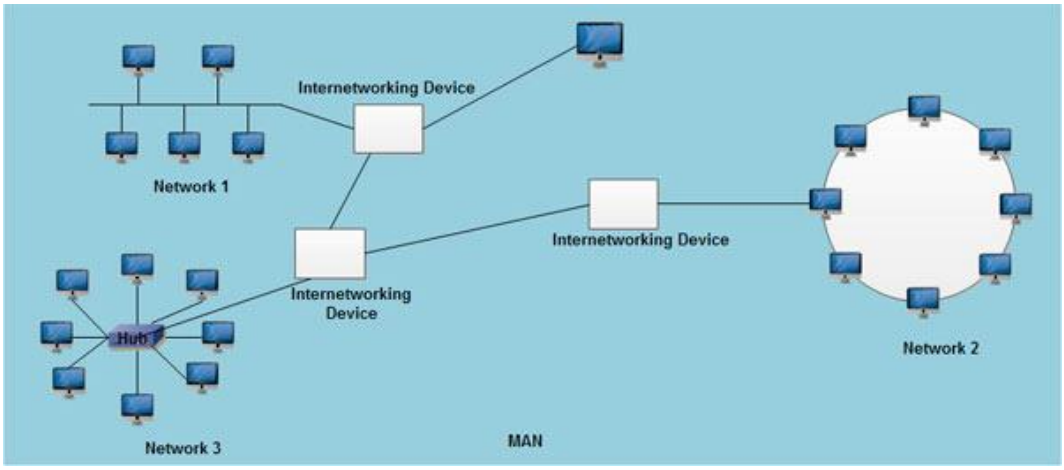
Such a system makes use of access point or APs to transmit and receive data. One of the computers in a network can become a server serving all the remaining computers called Clients.

For example, a library will have a wired or wireless LAN **Network** for users to interconnect local networking devices e.g., printers and servers to connect to the internet.

LAN offers high speed communication of data rates of 4 to 16 megabits per second (Mbps). **IEEE** has projects investigating the standardization of 100 Gbit/s, and possibly 40 Gbit/s. LANs **Network** may have connections with other LANs **Network** via leased lines, leased services.

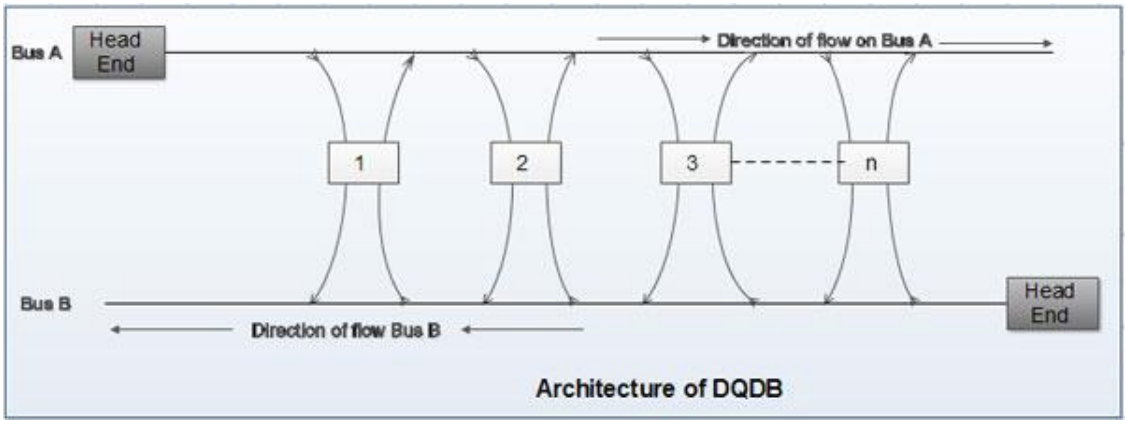
MAN (Metropolitan Area Networks)

MAN stands for Metropolitan Area Networks is one of a number of types of networks. A MAN is a relatively new class of network. MAN is larger than a local area network and as its name implies, covers the area of a single city. MANs rarely extend beyond 100 KM and frequently comprise a combination of different hardware and transmission media. It can be single network such as a cable TV network, or it is a means of connecting a number of LANs into a larger network so that resources can be shared LAN to LAN as well as device to device.



A MAN can be created as a single network such as Cable TV Network, covering the entire city or a group of several Local Area Networks (LANs). It this way resource can be shared from LAN to LAN and from computer to computer also. MANs are usually owned by large organizations to interconnect its various branches across a city.

MAN is based on IEEE 802.6 standard known as DQDB (Distributed Queue Dual Bus). DQDB uses two unidirectional cables (buses) and all the computers are connected to these two buses. Each bus has a specialized device that initiates the transmission activity. This device is called head end. Data that is to be sent to the computer on the right hand side of the sender is transmitted on upper bus. Data that is to be sent to the left hand side of the sender is transmitted on lower bus.



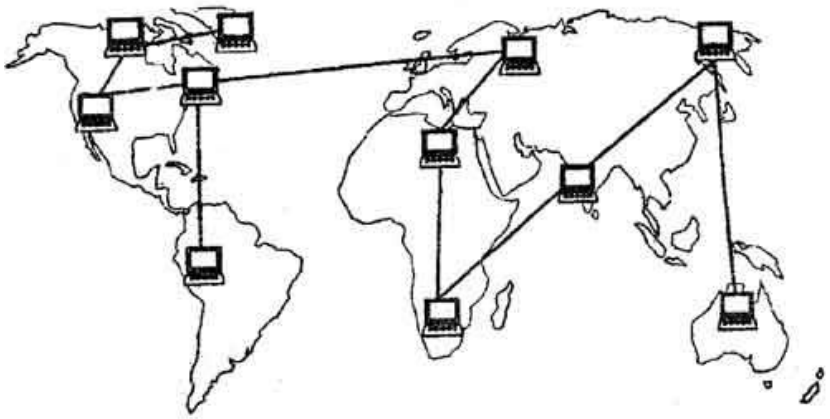
The two most important components of MANs are security and standardization. Security is important because information is being shared between dissimilar systems. Standardization is necessary to ensure reliable data communication.

A MAN usually interconnects a number of local area networks using a high-capacity backbone technology, such as fiber-optical links, and provides up-link services to wide area networks and the Internet.The Metropolitan Area Networks (MAN) protocols

are mostly at the data link level (layer 2 in the OSI model), which are defined by IEEE, ITU-T, etc.

WAN (Wide Area Networks)

A wide area network (WAN) is a telecommunication network. A wide area network is simply a LAN of LANs or Network of Networks. WANs connect LANs that may be on opposite sides of a building, across the country or around the world. WANS are characterized by the slowest data communication rates and the largest distances. WANs can be of two types: an enterprise WAN and Global WAN.



WAN

Computers connected to a Wide Area Networks are often connected through public networks, such as the telephone system. They can also be connected through leased lines or satellites. The largest WAN in existence is the Internet. Some segments of the Internet, like VPN based extranets, are also WANs in themselves. Finally, many WANs are corporate or research networks that utilize leased lines.

5 PARALLEL DATABASE SYSTEMS

Introduction

The transaction requirements of organizations have grown with increasing use of computers. Moreover, the growth of the World Wide Web has created many sites with millions of viewers, and the increasing amounts of data collected from these viewers has produced extremely large databases at many companies.Organizations are using these increasingly large volumes of data such as data about what items people buy, what Web links users clicked on, or when people make telephone calls to plan their activities and pricing. Queries used for such purposes are called decision-support queries, and the data requirements for such queries may run into terabytes. Single-processor systems are not capable of handling such large volumes of data at the required rates.The set-oriented nature of database queries naturally lends itself to parallelization. A number of commercial and research systems have demonstrated the power and scalability of parallel query processing.As microprocessors have become cheap, parallel machines have become common and relatively inexpensive.

As we discussed in Database System Architecture, parallelism is used to provide speedup, where queries are executed faster because more resources, such as processors and disks, are provided. Parallelism is also used to provide scaleup, where increasing workloads are handled without increased response time, via an increase in the degree of parallelism.

We outlined in Database System Architecture the different architectures for parallel database systems: shared-memory, shared-disk, shared-nothing, and hierarchical architectures. Briefly, in shared-memory architectures, all processors share a common memory and disks; in shared-disk architectures, processors have independent memories, but share disks; in shared-nothing architectures, processors share neither memory nor disks; and hierarchical architectures have nodes that share neither memory nor disks with each other, but internally each node has a shared-memory or a shared-disk architecture.

5.1 I/O Parallelism

In it simplest form, I/O parallelism refers to reducing the time required to retrieve relations from disk by partitioning the relations on multiple disks. The most common form of data partitioning in a parallel database environment is horizontal partitioning. In horizontal partitioning, the tuples of a relation are divided (or declustered) among many disks, so that each tuple resides on one disk. Several partitioning strategies have been proposed.

Partitioning Techniques

We present three basic data-partitioning strategies. Assume that there are n disks, D0,D1, . . .,Dn−1, across which the data are to be partitioned.

Round-robin. This strategy scans the relation in any order and sends the ith tuple to disk number Di mod n. The round-robin scheme ensures an even distribution of tuples across disks; that is, each disk has approximately the same number of tuples as the others.Hash partitioning. This declustering strategy designates one or more attributes from the given relation's schema as the partitioning attributes. A hash function is chosen whose range is {0, 1, . . . , n − 1}. Each tuple of the original relation is hashed on the partitioning attributes. If the hash function returns i, then the tuple is placed on disk Di.Range partitioning. This strategy distributes contiguous attribute-value ranges to each disk. It chooses a partitioning attribute, A, as a partitioning vector. The relation is partitioned as follows. Let [v0, v1, . . . , vn−2] denote the partitioning vector, such that, if $i < j$, then $v_i < v_j$. Consider a tuple t such that $t[A] = x$. If $x < v_0$, then t goes on disk D0. If $x \geq v_{n-2}$, then t goes on disk Dn−1. If $v_i \leq x < v_{i+1}$, then t goes on disk Di+1.

For example, range partitioning with three disks numbered 0, 1, and 2 may assign tuples with values less than 5 to disk 0, values between 5 and 40 to disk 1, and values greater than 40 to disk 2.Comparison of Partitioning Techniques

Once a relation has been partitioned among several disks, we can retrieve it in parallel, using all the disks. Similarly, when a relation is being partitioned, it can be written to multiple disks in parallel. Thus, the transfer rates for reading or writing an entire relation are much faster with I/O parallelism than without it. However, reading an entire

relation, or scanning a relation, is only one kind of access to data. Access to data can be classified as follows:

Scanning the entire relation Locating a tuple associatively (for example, employee-name = “Campbell”); these queries, called point queries, seek tuples that have a specified value for a specific attributeLocating all tuples for which the value of a given attribute lies within a specified range (for example, 10000 < salary < 20000); these queries are called range queries.The different partitioning techniques support these types of access at different levels of efficiency:

Round-robin. The scheme is ideally suited for applications that wish to read the entire relation sequentially for each query. With this scheme, both point queries and range queries are complicated to process, since each of the n disks must be used for the search.Hash partitioning. This scheme is best suited for point queries based on the partitioning attribute. For example, if a relation is partitioned on the telephone number attribute, then we can answer the query “Find the record of the employee with telephone-number = 555-3333” by applying the partitioning hash function to 555-3333 and then searching that disk. Directing a query to a single disk saves the startup cost of initiating a query on multiple disks, and leaves the other disks free to process other queries.

Hash partitioning is also useful for sequential scans of the entire relation. If the hash function is a good randomizing function, and the partitioning attributes form a key of the relation, then the number of tuples in each of the disks is approximately the same, without much variance. Hence, the time taken to scan the relation is approximately 1/n of the time required to scan the relation in a single disk system.

The scheme, however, is not well suited for point queries on nonpartitioning attributes. Hash-based partitioning is also not well suited for answering range queries, since, typically, hash functions do not preserve proximity within a range. Therefore, all the disks need to be scanned for range queries to be answered.

Range partitioning. This scheme is well suited for point and range queries on the partitioning attribute. For point queries, we can consult the partitioning vector to locate the disk where the tuple resides. For range queries, we consult the partitioning vector to find the range of disks on which the tuples may reside. In both cases, the search narrows to exactly those disks that might have any tuples of interest.

An advantage of this feature is that, if there are only a few tuples in the queried range, then the query is typically sent to one disk, as opposed to all the disks. Since other disks can be used to answer other queries, range partitioning results in higher throughput while maintaining good response time. On the other hand, if there are many tuples in the queried range (as there are when the queried range is a larger fraction of the domain of the relation), many tuples have to be retrieved from a few disks, resulting in an I/O bottleneck (hot spot) at those disks. In this example of execution skew, all processing occurs in one or only a few partitions. In contrast, hash partitioning and round-robin partitioning would engage all the disks for such queries, giving a faster response time for approximately the same throughput.

The type of partitioning also affects other relational operations, such as joins . Thus, the choice of partitioning technique also depends on the operations that need to be executed. In general, hash partitioning or range partitioning are preferred to round-robin

partitioning. In a system with many disks, the number of disks across which to partition a relation can be chosen in this way: If a relation contains only a few tuples that will fit into a single disk block, then it is better to assign the relation to a single disk. Large relations are preferably partitioned across all the available disks. If a relation consists of m disk blocks and there are n disks available in the system, then the relation should be allocated min(m, n) disks.

Handling of Skew

When a relation is partitioned (by a technique other than round-robin), there may be a skew in the distribution of tuples, with a high percentage of tuples placed in some partitions and fewer tuples in other partitions. The ways that skew may appear are classified as:

Attribute-value skewPartition skew

Attribute-value skew refers to the fact that some values appear in the partitioning attributes of many tuples. All the tuples with the same value for the partitioning attribute end up in the same partition, resulting in skew. Partition skew refers to the fact that there may be load imbalance in the partitioning, even when there is no attribute skew.

Attribute-value skew can result in skewed partitioning regardless of whether range partitioning or hash partitioning is used. If the partition vector is not chosen carefully, range partitioning may result in partition skew. Partition skew is less likely with hash partitioning, if a good hash function is chosen.

For example, if a relation of 1000 tuples is divided into 10 parts, and the division is skewed, then there may be some partitions of size less than 100 and some partitions of size more than 100; if even one partition happens to be of size 200, the speedup that we would obtain by accessing the partitions in parallel is only 5, instead of the 10 for which we would have hoped. If the same relation has to be partitioned into 100 parts, a partition will have 10 tuples on an average. If even one partition has 40 tuples (which is possible, given the large number of partitions) the speedup that we would obtain by accessing them in parallel would be 25, rather than 100. Thus, we see that the loss of speedup due to skew increases with parallelism.

A balanced range-partitioning vector can be constructed by sorting: The relation is first sorted on the partitioning attributes. The relation is then scanned in sorted order. After every 1/n of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partition vector. Here, n denotes the number of partitions to be constructed. In case there are many tuples with the same value for the partitioning attribute, the technique can still result in some skew. The main disadvantage of this method is the extra I/O overhead incurred in doing the initial sort.

The I/O overhead for constructing balanced range-partition vectors can be reduced by constructing and storing a frequency table, or histogram, of the attribute values for each attribute of each relation. Figure shows an example of a histogram for an integer-valued attribute that takes values in the range 1 to 25. A histogram takes up only a little space, so histograms on several different attributes can be stored in the system catalog. It is straightforward to construct a balanced range-partitioning function given a histogram on

the partitioning attributes. If the histogram is not stored, it can be computed approximately by sampling the relation, using only tuples from a randomly chosen subset of the disk blocks of the relation.

Example of histogram.

Another approach to minimizing the effect of skew, particularly with range partitioning, is to use virtual processors. In the virtual processor approach, we pretend there are several times as many virtual processors as the number of real processors. Any of the partitioning techniques and query evaluation techniques that we study later in this chapter can be used, but they map tuples and work to virtual processors instead of to real processors. Virtual processors, in turn, are mapped to real processors, usually by round-robin partitioning.

The idea is that even if one range had many more tuples than the others because of skew, these tuples would get split across multiple virtual processor ranges. Round robin allocation of virtual processors to real processors would distribute the extra work among multiple real processors, so that one processor does not have to bear all the burden.

5.2 Interquery Parallelism

In interquery parallelism, different queries or transactions execute in parallel with one another. Transaction throughput can be increased by this form of parallelism. However, the response times of individual transactions are no faster than they would be if the transactions were run in isolation. Thus, the primary use of interquery parallelism is to scaleup a transaction-processing system to support a larger number of transactions per second.

Interquery parallelism is the easiest form of parallelism to support in a database system particularly in a shared-memory parallel system. Database systems designed for single-processor systems can be used with few or no changes on a shared-memory parallel architecture, since even sequential database systems support concurrent processing. Transactions that would have operated in a time-shared concurrent manner on a sequential machine operate in parallel in the shared-memory parallel architecture.

Supporting interquery parallelism is more complicated in a shared-disk or sharednothing architecture. Processors have to perform some tasks, such as locking and logging, in a coordinated fashion, and that requires that they pass messages to each other. A parallel database system must also ensure that two processors do not update the same data independently at the same time. Further, when a processor accesses or updates data, the database system must ensure that the processor has the latest version of the data in its buffer pool. The problem of ensuring that the version is the latest is known as the cache-coherency problem.

Various protocols are available to guarantee cache coherency; often, cache-coherency protocols are integrated with concurrency-control protocols so that their overhead is reduced. One such protocol for a shared-disk system is this:

5.3 Intraquery Parallelism

Intraquery parallelism refers to the execution of a single query in parallel on multiple processors and disks. Using intraquery parallelism is important for speeding up long-running queries. Interquery parallelism does not help in this task, since each query is run sequentially. To illustrate the parallel evaluation of a query, consider a query that requires a relation to be sorted. Suppose that the relation has been partitioned across multiple disks by range partitioning on some attribute, and the sort is requested on the partitioning attribute. The sort operation can be implemented by sorting each partition in parallel, then concatenating the sorted partitions to get the final sorted relation.

Thus, we can parallelize a query by parallelizing individual operations. There is another source of parallelism in evaluating a query: The operator tree for a query can contain multiple operations.We can parallelize the evaluation of the operator tree by evaluating in parallel some of the operations that do not depend on one another. The two operations can be executed in parallel on separate processors, one generating output that is consumed by the other, even as it is generated.

In summary, the execution of a single query can be parallelized in two ways:

Intraoperation parallelism. We can speed up processing of a query by parallelizing the execution of each individual operation, such as sort, select, project, and join.Interoperation parallelism.We can speed up processing of a query by executing in parallel the different operations in a query expression.

The two forms of parallelism are complementary, and can be used simultaneously on a query. Since the number of operations in a typical query is small, compared to the number of tuples processed by each operation, the first form of parallelism can scale better with increasing parallelism. However,with the relatively small number of processors in typical parallel systems today, both forms of parallelism are important.

In the following discussion of parallelization of queries, we assume that the queries are read only. The choice of algorithms for parallelizing query evaluation depends on the machine architecture. Rather than presenting algorithms for each architecture separately, we use a shared-nothing architecture model in our description. Thus, we explicitly describe when data have to be transferred from one processor to another.

We can simulate this model easily by using the other architectures, since transfer of data can be done via shared memory in a shared-memory architecture, and via shared disks in a shared-disk architecture. Hence, algorithms for shared-nothing architectures can be used on the other architectures too. We mention occasionally how the algorithms can be further optimized for shared-memory or shared-disk systems.

To simplify the presentation of the algorithms, assume that there are n processors, P0, P1, . . . , Pn−1, and n disks D0,D1, . . .,Dn−1, where disk Di is associated with processor Pi. A real system may have multiple disks per processor. It is not hard to extend the algorithms to allow multiple disks per processor: We simply allow Di to be a set of disks. However, for simplicity, we assume here that Di is a single disk.

Intraoperation Parallelism

Since relational operations work on relations containing large sets of tuples, we can parallelize the operations by executing them in parallel on different subsets of the relations. Since the number of tuples in a relation can be large, the degree of parallelism is potentially enormous. Thus, intraoperation parallelism is natural in a database system.

6. DISTRIBUTED DATABASE

Distributed Databases

Unlike parallel systems, in which the processors are tightly coupled and constitute a single database system, a distributed database system consists of loosely coupled sites that share no physical components. Furthermore, the database systems that run on each site may have a substantial degree of mutual independence. We discussed the basic structure of distributed systems in Database System Architecture.

Each site may participate in the execution of transactions that access data at one site, or several sites. The main difference between centralized and distributed database systems is that, in the former, the data reside in one single location, whereas in the latter, the data reside in several locations. This distribution of data is the cause of many difficulties in transaction processing and query processing. In this chapter, we address these difficulties.

We start by classifying distributed databases as homogeneous or heterogeneous. We then address the question of how to store data in a distributed database.

6.1 Homogeneous and Heterogeneous Databases

In a homogeneous distributed database, all sites have identical database management system software, are aware of one another, and agree to cooperate in processing users' requests. In such a system, local sites surrender a portion of their autonomy in terms of their right to change schemas or database management system software.

That software must also cooperate with other sites in exchanging information about transactions, to make transaction processing possible across multiple sites. In contrast, in a heterogeneous distributed database, different sites may use different schemas, and different database management system software. The sites may not be aware of one another, and they may provide only limited facilities for cooperation in transaction processing. The differences in schemas are often a major problem for query processing, while the divergence in software becomes a hindrance for processing transactions that access multiple sites.

In this chapter, we concentrate on homogeneous distributed databases. Transaction processing issues in such systems are covered later.

Distributed Data Storage

Consider a relation r that is to be stored in the database. There are two approaches to storing this relation in the distributed database:

Replication The system maintains several identical replicas (copies) of the relation, and stores each replica at a different site. The alternative to replication is to store only one copy of relation r.Fragmentation. The system partitions the relation into several fragments, and stores each fragment at a different site.

Fragmentation and replication can be combined: A relation can be partitioned into several fragments and there may be several replicas of each fragment. In the following subsections, we elaborate on each of these techniques.

Data Replication

If relation r is replicated, a copy of relation r is stored in two or more sites. In the most extreme case, we have full replication, in which a copy is stored in every site in the system. There are a number of advantages and disadvantages to replication.

Availability. If one of the sites containing relation r fails, then the relation r can be found in another site. Thus, the system can continue to process queries involving r, despite the failure of one site.Increased parallelism. In the case where the majority of accesses to the relation r result in only the reading of the relation, then several sites can process queries involving r in parallel. The more replicas of r there are, the greater the chance that the needed data will be found in the site where the transaction is executing. Hence, data replication minimizes movement of data between sites.Increased overhead on update. The system must ensure that all replicas of a relation r are consistent; otherwise, erroneous computations may result. Thus, whenever r is updated, the update must be propagated to all sites containing replicas. The result is increased overhead. For example, in a banking system, where account information is replicated in various sites, it is necessary to ensure that the balance in a particular account agrees in all sites.

Data Fragmentation

If relation r is fragmented, r is divided into a number of fragments r1, r2, . . . , rn. These fragments contain sufficient information to allow reconstruction of the original relation r. There are two different schemes for fragmenting a relation: horizontal fragmentation and vertical fragmentation. Horizontal fragmentation splits the relation by assigning each tuple of r to one or more fragments. Vertical fragmentation splits the relation by decomposing the scheme R of relation r.

We shall illustrate these approaches by fragmenting the relation account, with the schema

Account-schema = (account-number, branch-name, balance)

In horizontal fragmentation, a relation r is partitioned into a number of subsets, r1, r2, . . . , rn. Each tuple of relation r must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed.As an illustration, the account relation can

be divided into several different fragments, each of which consists of tuples of accounts belonging to a particular branch.

If the banking system has only two branches Hillside and Valley view then there are two different fragments:

account1 = σbranch-name = “Hillside” (account) account2 = σbranch-name = “Valleyview” (account)

Horizontal fragmentation is usually used to keep tuples at the sites where they are used the most, to minimize data transfer. In general, a horizontal fragment can be defined as a selection on the global relation r. That is,we use a predicate Pi to construct fragment ri:

$r_i = \sigma_{P_i}(r)$

We reconstruct the relation r by taking the union of all fragments; that is,

$r = r_1 \cup r_2 \cup \cdots \cup r_n$

In our example, the fragments are disjoint. By changing the selection predicates used to construct the fragments, we can have a particular tuple of r appear in more than one of the ri. In its simplest form, vertical fragmentation is the same as decomposition . Vertical fragmentation of r(R) involves the definition of several subsets of attributes R1, R2, . . .,Rn of the schema R so that

$R = R_1 \cup R_2 \cup \cdots \cup R_n$

Each fragment ri of r is defined by

$r_i = \Pi_{R_i}(r)$

The fragmentation should be done in such a way that we can reconstruct relation r from the fragments by taking the natural join

$r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \cdots \bowtie r_n$

One way of ensuring that the relation r can be reconstructed is to include the primary-key attributes of R in each of the Ri. More generally, any superkey can be used. It is often convenient to add a special attribute, called a tuple-id, to the schema R. The tuple-id value of a tuple is a unique value that distinguishes the tuple from all other tuples. The tuple-id attribute thus serves as a candidate key for the augmented schema, and is included in each of the Ris. The physical or logical address for a tuple can be used as a tuple-id, since each tuple has a unique address.

To illustrate vertical fragmentation, consider a university database with a relation employee-info that stores, for each employee, employee-id, name, designation, and salary. For privacy reasons, this relation may be fragmented into a relation employee-privateinfo containing employee-id and salary, and another relation employee-public-info containing

attributes employee-id, name, and designation. These may be stored at different sites, again for security reasons.

The two types of fragmentation can be applied to a single schema; for instance, the fragments obtained by horizontally fragmenting a relation can be further partitioned vertically. Fragments can also be replicated. In general, a fragment can be replicated, replicas of fragments can be fragmented further, and so on.

Transparency

The user of a distributed database system should not be required to know either where the data are physically located or how the data can be accessed at the specific local site. This characteristic, called data transparency, can take several forms:

Fragmentation transparency

Users are not required to know how a relation has been fragmented.Replication transparency. Users view each data object as logically unique. The distributed system may replicate an object to increase either system performance or data availability. Users do not have to be concerned with what data objects have been replicated, or where replicas have been placed.Location transparency. Users are not required to know the physical location of the data. The distributed database system should be able to find any data as long as the data identifier is supplied by the user transaction.

Data items such as relations, fragments, and replicas must have unique names. This property is easy to ensure in a centralized database. In a distributed database, however, we must take care to ensure that two sites do not use the same name for distinct data items.

One solution to this problem is to require all names to be registered in a central name server. The name server helps to ensure that the same name does not get used for different data items. We can also use the name server to locate a data item, given the name of the item. This approach, however, suffers from two major disadvantages.

First, the name server may become a performance bottleneck when data items are located by their names, resulting in poor performance. Second, if the name server crashes, it may not be possible for any site in the distributed system to continue to run.

A more widely used alternative approach requires that each site prefix its own site identifier to any name that it generates. This approach ensures that no two sites generate the same name (since each site has a unique identifier). Furthermore, no central control is required. This solution, however, fails to achieve location transparency, since site identifiers are attached to names. Thus, the account relation might be referred to as site17.account, or account@site17, rather than as simply account. Many database systems use the internet address of a site to identify it.

To overcome this problem, the database system can create a set of alternative names or aliases for data items. A user may thus refer to data items by simple names that are translated by the system to complete names. The mapping of aliases to the real names can be stored at each site. With aliases, the user can be unaware of the physical location of a data item. Furthermore, the user will be unaffected if the database administrator decides to move a data item from one site to another.

Users should not have to refer to a specific replica of a data item. Instead, the system should determine which replica to reference on a read request, and should update all replicas on a write request.We can ensure that it does so by maintaining a catalog table, which the system uses to determine all replicas for the data item.

6.2 Distributed Transactions

Access to the various data items in a distributed system is usually accomplished through transactions, which must preserve the ACID properties . There are two types of transaction that we need to consider. The local transactions are those that access and update data in only one local database; the global transactions are those that access and update data in several local databases. Ensuring the ACID properties of the local transactions can be done However, for global transactions, this task is much more complicated, since several sites may be participating in execution. The failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations.

System Structure

Each site has its own local transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that site. The various transaction managers cooperate to execute global transactions. To understand how such a manager can be implemented, consider an abstract model of a transaction system, in which each site contains two subsystems:

The transaction manager manages the execution of those transactions (or subtransactions) that access data stored in a local site. Note that each such transaction may be either a local transaction (that is, a transaction that executes at only that site) or part of a global transaction (that is, a transaction that executes at several sites).The transaction coordinator coordinates the execution of the various transactions (both local and global) initiated at that site.

UNIT IV

7. SCHEMA OBJETS

7.1 DATA INTEGRITY

The preservation of the integrity of a database system is concerned with the maintenance of the correctness and consistency of the data In a multi-user database environment this is a major task, since integrity violations may arise from many different sources, such as typing errors by data entry clerks, logical errors in application programs, or errors in system software which result in data corruption.

Many commercial database management systems have an integrity subsystem, which is responsible for monitoring transactions, which update the database and detecting integrity violations. In the event of an integrity violation, the system then takes appropriate action, which should involve rejecting the operation, reporting the violation, and if necessary returning the database to a consistent state.

Integrity rules may be divided into three 'broad categories:

• Domain integrity rules

• Entity integrity rules

• Referential integrity rules

Domain integrity rules

Domain integrity rules are concerned with maintaining the correctness of attribute values within relations. A domain integrity rule therefore, is simply a definition of the type of the domain, and domain integrity is closely related to the familiar concept of type checking in programming languages. The definition of the type of a domain must be as precise as possible in order to avoid violations of domain integrity.

Example: If we have an attribute AGE, it is not sufficient to describe its type as INTEGER since this does not prevent unrealistic values for AGE (e.g. negative values) being entered into the database. At the very least we should be able to specify that the domain type for attribute AGE is POSITIVE_INTEGER, and ideally it should be possible to specify upper and lower bounds for values of AGE. Unfortunately commercial database management 'systems typically provide only simple types for domains. For example, the ORACLE database management system provides the domain types: NUMBER, CHAR (variable length character strings), DATE and TIME. INGRES and DB2 provide similar restricted domain types.

Entity integrity rules

Entity integrity rules relate to the correctness of relationships among attributes of the same relation (e.g. function: dependencies) and to the preservation of key uniqueness.

Requirement of entity integrity rules: All entries are unique and no null entries in a primary key.

Purpose of Entity identity rules: Guarantees that each entity will have a unique

Referential integrity rules: Referential integrity rules are concerned with maintaining the correctness and consistency of relationships between relations.

Requirement of Referential integrity rules: Foreign key must have either a null entry or an entry that matches the primary key value in a table to which it is related.

Purpose of Referential integrity rules: Makes it possible for an attribute NOT to have a corresponding value, but it will be impossible to have an invalid entry. The enforcement of the referential integrity rule makes it impossible to delete a row in one table whose primary key has mandatory matching foreign key values in another table.

Example: A customer might not (yet) have an assigned sales representative (number), but it will be impossible to have an invalid sales representative (number).

Example of Entity integrity rule and referential integrity rule:

Consider the following Database Table features:

Table Name: CUSTOMER

Table Fields: CUS_CODE, CUS_NAME, CUS_RENEW _DATE, AGENT_CODE

Primary Name: CUS_CODE

Foreign key: AGENT_CODE

Table Name: AGENT

Table Fields: AGENT _CODE, AGENT _AREA _CODE,AGENT _PHONE, AGENT LNAME

Primary Name: AGENT CODE

Foreign key: None

Entity integrity: The CUSTOMER table's primary key is CUS_CODE. The CUSTOMER primary key column has no null entries and all entries are unique. Similarly, the AGENT table's primary key is AGENT_CODE, and this primary key column also is free of null entries.

Referential integrity: The CUSTOMER table contains a foreign key AGENT_CODE, which links entries in the CUSTOMER table to the AGENT table.

7.2 CREATING AND MAINTAINING TABLES

Creating a basic table involves naming the table and defining its columns and each column's data type.The SQL **CREATE TABLE** statement is used to create a new table.

Syntax

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

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

CREATE TABLE is the keyword telling the database system what you want to do. In this case, you want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement.

Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with the following example.

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. You can check the complete details at Create Table Using another Table.

Example

The following code block is an example, which creates a CUSTOMERS table with an ID as a primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table −

```
SQL> CREATE TABLE CUSTOMERS(
   ID   INT              NOT NULL,
   NAME VARCHAR (20)     NOT NULL,
   AGE  INT              NOT NULL,
   ADDRESS  CHAR (25) ,
   SALARY   DECIMAL (18, 2),
   PRIMARY KEY (ID)
);
```

You can verify if your table has been created successfully by looking at the message displayed by the SQL server, otherwise you can use the **DESC** command as follows –

```
SQL> DESC CUSTOMERS;
+--------- +--------------+------ +-----+---------+-------+
| Field   |    Type         | Null |  Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| ID        | int(11)        | NO   | PRI |         |       |
| NAME    | varchar(20)   | NO   |     |         |       |
| AGE     | int(11)        | NO   |     |         |       |
| ADDRESS | char(25)       | YES  |     | NULL    |       |
| SALARY  | decimal(18,2) | YES  |     | NULL    |       |
+---------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

Now, you have CUSTOMERS table available in your database which you can use to store the required information related to customers.

SQL - ALTER TABLE Command

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

Syntax

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

ALTER TABLE table_name ADD column_name datatype;

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

ALTER TABLE table_name DROP COLUMN column_name;

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

ALTER TABLE table_name MODIFY COLUMN column_name datatype;

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

ALTER TABLE table_name MODIFY column_name datatype NOT NULL;

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

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

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

```
ALTER TABLE table_name
ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);
```

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

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

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

```
ALTER TABLE table_name
DROP CONSTRAINT MyUniqueConstraint;
```

If you're using MySQL, the code is as follows −

```
ALTER TABLE table_name
DROP INDEX MyUniqueConstraint;
```

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

```
ALTER TABLE table_name
DROP CONSTRAINT MyPrimaryKey;
```

If you're using MySQL, the code is as follows −

```
ALTER TABLE table_name
DROP PRIMARY KEY;
```

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is the example to ADD a **New Column** to an existing table −

ALTER TABLE CUSTOMERS ADD SEX char(1);

Now, the CUSTOMERS table is changed and following would be output from the SELECT statement.

```
+----+---------+-----+-----------+----------+------+
| ID | NAME    | AGE | ADDRESS   | SALARY   | SEX  |
+----+---------+-----+-----------+----------+------+
|  1 | Ramesh  |  32 | Ahmedabad |  2000.00 | NULL |
|  2 | Ramesh  |  25 | Delhi     |  1500.00 | NULL |
|  3 | kaushik |  23 | Kota      |  2000.00 | NULL |
|  4 | kaushik |  25 | Mumbai    |  6500.00 | NULL |
|  5 | Hardik  |  27 | Bhopal    |  8500.00 | NULL |
|  6 | Komal   |  22 | MP        |  4500.00 | NULL |
|  7 | Muffy   |  24 | Indore    | 10000.00 | NULL |
+----+---------+-----+-----------+----------+------+
```

Following is the example to DROP sex column from the existing table.

ALTER TABLE CUSTOMERS DROP SEX;

Now, the CUSTOMERS table is changed and following would be the output from the SELECT statement.

```
+----+---------+-----+-----------+----------+
| ID | NAME    | AGE | ADDRESS   | SALARY   |
+----+---------+-----+-----------+----------+
|  1 | Ramesh  |  32 | Ahmedabad |  2000.00 |
|  2 | Ramesh  |  25 | Delhi     |  1500.00 |
|  3 | kaushik |  23 | Kota      |  2000.00 |
|  4 | kaushik |  25 | Mumbai    |  6500.00 |
|  5 | Hardik  |  27 | Bhopal    |  8500.00 |
|  6 | Komal   |  22 | MP        |  4500.00 |
|  7 | Muffy   |  24 | Indore    | 10000.00 |
+----+---------+-----+-----------+----------
```

SQL - DROP or DELETE Table

The SQL DROP TABLE statement is used to remove a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.

NOTE − You should be very careful while using this command because once a table is deleted then all the information available in that table will also be lost forever.

Syntax

The basic syntax of this DROP TABLE statement is as follows −

DROP TABLE table_name;

Example

Let us first verify the CUSTOMERS table and then we will delete it from the database as shown below −

```
SQL> DESC CUSTOMERS;
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| ID      | int(11)      | NO   | PRI |         |       |
| NAME    | varchar(20)  | NO   |     |         |       |
| AGE     | int(11)      | NO   |     |         |       |
| ADDRESS | char(25)     | YES  |     | NULL    |       |
| SALARY  | decimal(18,2) | YES  |     | NULL    |       |
+---------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

This means that the CUSTOMERS table is available in the database, so let us now drop it as shown below.

```
SQL> DROP TABLE CUSTOMERS;
Query OK, 0 rows affected (0.01 sec)
```

Now, if you would try the DESC command, then you will get the following error −

```
SQL> DESC CUSTOMERS;
ERROR 1146 (42S02): Table 'TEST.CUSTOMERS' doesn't exist
```

7.3 INDEXES

Indexes are **special lookup tables** that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.

An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

The CREATE INDEX Command

The basic syntax of a **CREATE INDEX** is as follows.

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes

A single-column index is created based on only one table column. The basic syntax is as follows.

```
CREATE INDEX index_name
ON table_name (column_name);
```

Unique Indexes

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows.

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

Composite Indexes

A composite index is an index on two or more columns of a table. Its basic syntax is as follows.

```
CREATE INDEX index_name
on table_name (column1, column2);
```

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

The DROP INDEX Command

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because the performance may either slow down or improve.

The basic syntax is as follows −

DROP INDEX index_name;

7.4 SQL - Using Views

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

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

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

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

Creating Views

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

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

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

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

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

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

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

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

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

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

This would produce the following result.

```
+----------+-----+
| name     | age |
+----------+-----+
| Ramesh   |  32 |
| Khilan   |  25 |
| kaushik  |  23 |
| Chaitali |  25 |
| Hardik   |  27 |
| Komal    |  22 |
| Muffy    |  24 |
+----------+-----+
```

The WITH CHECK OPTION

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

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

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

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

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

Updating a View

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

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

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

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

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

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Deleting Rows into a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE = 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW
  WHERE age = 22;
```

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

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below −

```
DROP VIEW view_name;
```

7.5 SQL - SEQUENCES

A sequence is a set of integers 1, 2, 3, ... that are generated in order on demand. Sequences are frequently used in databases because many applications require each row in a table to contain a unique value and sequences provide an easy way to generate them.

This chapter describes how to use sequences in MySQL.

Using AUTO_INCREMENT column

The simplest way in MySQL to use sequences is to define a column as AUTO_INCREMENT and leave the rest to MySQL to take care.

Example

Try out the following example. This will create a table and after that it will insert a few rows in this table where it is not required to give a record ID because its auto-incremented by MySQL.

```
mysql> CREATE TABLE INSECT
   -> (
   -> id INT UNSIGNED NOT NULL AUTO_INCREMENT,
   -> PRIMARY KEY (id),
   -> name VARCHAR(30) NOT NULL, # type of insect
   -> date DATE NOT NULL, # date collected
   -> origin VARCHAR(30) NOT NULL # where collected
);
Query OK, 0 rows affected (0.02 sec)
mysql> INSERT INTO INSECT (id,name,date,origin) VALUES
   -> (NULL,'housefly','2001-09-10','kitchen'),
   -> (NULL,'millipede','2001-09-10','driveway'),
   -> (NULL,'grasshopper','2001-09-10','front yard');

Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT * FROM INSECT ORDER BY id;
+----+-------------+------------+------------+
| id | name        | date       | origin     |
+----+-------------+------------+------------+
|  1 | housefly    | 2001-09-10 | kitchen    |
|  2 | millipede   | 2001-09-10 | driveway   |
|  3 | grasshopper | 2001-09-10 | front yard |
+----+-------------+------------+------------+
3 rows in set (0.00 sec)
```

Obtain AUTO_INCREMENT Values

The LAST_INSERT_ID( ) is an SQL function, so you can use it from within any client that understands how to issue SQL statements. Otherwise PERL and PHP scripts provide exclusive functions to retrieve auto-incremented value of last record.

PERL Example

Use the **mysql_insertid** attribute to obtain the AUTO_INCREMENT value generated by a query. This attribute is accessed through either a database handle or a statement handle, depending on how you issue the query. The following example references it through the database handle.

```
$dbh->do ("INSERT INTO INSECT (name,date,origin)
VALUES('moth','2001-09-14','windowsill')");
my $seq = $dbh->{mysql_insertid};
```

PHP Example

After issuing a query that generates an AUTO_INCREMENT value, retrieve the value by calling the **mysql_insert_id( )** function.

```
mysql_query ("INSERT INTO INSECT (name,date,origin)
VALUES('moth','2001-09-14','windowsill')", $conn_id);
$seq = mysql_insert_id ($conn_id);
```

Renumbering an Existing Sequence

There may be a case when you have deleted many records from a table and you want to re-sequence all the records. This can be done by using a simple trick, but you should be very careful to do this and check if your table is having a join with another table or not.

If you determine that resequencing an AUTO_INCREMENT column is unavoidable, the way to do it is to drop the column from the table, then add it again.

The following example shows how to renumber the id values in the insect table using this technique.

```
mysql> ALTER TABLE INSECT DROP id;
```

```
mysql> ALTER TABLE insect
  -> ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT FIRST,
  -> ADD PRIMARY KEY (id);
```

Starting a Sequence at a Particular Value

By default, MySQL will start the sequence from 1, but you can specify any other number as well at the time of table creation.

The following code block has an example where MySQL will start sequence from 100.

```
mysql> CREATE TABLE INSECT
  -> (
  -> id INT UNSIGNED NOT NULL AUTO_INCREMENT = 100,
  -> PRIMARY KEY (id),
  -> name VARCHAR(30) NOT NULL, # type of insect
  -> date DATE NOT NULL, # date collected
  -> origin VARCHAR(30) NOT NULL # where collected
);
```

Alternatively, you can create the table and then set the initial sequence value with ALTER TABLE.

7.6 USER PRIVILEGS AND ROLES

SQL GRANT REVOKE Commands

DCL commands are used to enforce database security in a multiple user database environment. Two types of DCL commands are GRANT and REVOKE. Only Database Administrator's or owner's of the database object can provide/remove privileges on a database object.

SQL GRANT Command

SQL GRANT is a command used to provide access or privileges on the database objects to the users.

The Syntax for the GRANT command is:

```
GRANT                                                    privilege_name
ON                                                         object_name
TO              {user_name              |PUBLIC          |role_name}
[WITH GRANT OPTION];
```

- **privilege_name** is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
- **object_name** is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
- **user_name** is the name of the user to whom an access right is being granted.
- **user_name** is the name of the user to whom an access right is being granted.
- **PUBLIC** is used to grant access rights to all users.
- **ROLES** are a set of privileges grouped together.

- **WITH GRANT OPTION** - allows a user to grant access rights to other users.

For Example: GRANT SELECT ON employee TO user1; This command grants a SELECT permission on employee table to user1.You should use the WITH GRANT option carefully because for example if you GRANT SELECT privilege on employee table to user1 using the WITH GRANT option, then user1 can GRANT SELECT privilege on employee table to another user, such as user2 etc. Later, if you REVOKE the SELECT privilege on employee from user1, still user2 will have SELECT privilege on employee table.

SQL REVOKE Command:

The REVOKE command removes user access rights or privileges to the database objects.

The Syntax for the REVOKE command is:

```
REVOKE                                    privilege_name
ON                                          object_name
FROM {user_name |PUBLIC |role_name}
```

For Example: REVOKE SELECT ON employee FROM user1;This command will REVOKE a SELECT privilege on employee table from user1.When you REVOKE SELECT privilege on a table from a user, the user will not be able to SELECT data from that table anymore. However, if the user has received SELECT privileges on that table from more than one users, he/she can SELECT from that table until everyone who granted the permission revokes it. You cannot REVOKE privileges if they were not initially granted by you.

Privileges and Roles:

Privileges: Privileges defines the access rights provided to a user on a database object. There are two types of privileges.

1) System privileges - This allows the user to CREATE, ALTER, or DROP database objects.
2) Object privileges - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.

Few CREATE system privileges are listed below:

System Privileges	Description
CREATE object	allows users to create the specified object in their own schema.
CREATE ANY object	allows users to create the specified object in any schema.

The above rules also apply for ALTER and DROP system privileges.

Few of the object privileges are listed below:

Object Privileges	Description
INSERT	allows users to insert rows into a table.
SELECT	allows users to select data from a database object.
UPDATE	allows user to update data in a table.
EXECUTE	allows user to execute a stored procedure or a function.

Roles: Roles are a collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles, you can grant or revoke privileges to users, thereby automatically granting or revoking privileges. You can either create Roles or use the system roles pre-defined by oracle.

Some of the privileges granted to the system roles are as given below:

System Role	Privileges Granted to the Role
CONNECT	CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE SEQUENCE, CREATE SESSION etc.
RESOURCE	CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER etc. The primary usage of the RESOURCE role is to restrict access to database objects.
DBA	ALL SYSTEM PRIVILEGES

Creating Roles:

The Syntax to create a role is:

```
CREATE ROLE role_name
[IDENTIFIED BY password];
```

For Example: To create a role called "developer" with password as "pwd",the code will be as follows

```
CREATE ROLE testing
[IDENTIFIED BY pwd];
```

It's easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user. If a role is identified by a password, then, when you GRANT or REVOKE privileges to the role, you definitely have to identify it with the password.

We can GRANT or REVOKE privilege to a role as below.

For example: To grant CREATE TABLE privilege to a user by creating a testing role:

First, create a testing Role

CREATE ROLE testing

Second, grant a CREATE TABLE privilege to the ROLE testing. You can add more privileges to the ROLE.

GRANT CREATE TABLE TO testing;

Third, grant the role to a user.

GRANT testing TO user1;

To revoke a CREATE TABLE privilege from testing ROLE, you can write:

REVOKE CREATE TABLE FROM testing;

The Syntax to drop a role from the database is as below:

DROP ROLE role_name;

For example: To drop a role called developer, you can write:

DROP ROLE testing;

7.7 SYNONYS

This Oracle tutorial explains how to **create and drop synonyms** in Oracle with syntax and examples.

Description

A **synonym** is an alternative name for objects such as tables, views, sequences, stored procedures, and other database objects.

You generally use synonyms when you are granting access to an object from another schema and you don't want the users to have to worry about knowing which schema owns the object.

Create Synonym (or Replace)

You may wish to create a synonym so that users do not have to prefix the table name with the schema name when using the table in a query.

Syntax

The syntax to create a synonym in Oracle is:

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema .] synonym_name
  FOR [schema .] object_name [@ dblink];
```

OR REPLACE

Allows you to recreate the synonym (if it already exists) without having to issue a DROP synonym command.

PUBLIC

It means that the synonym is a public synonym and is accessible to all users. Remember though that the user must first have the appropriate privileges to the object to use the synonym.

schema

The appropriate schema. If this phrase is omitted, Oracle assumes that you are referring to your own schema.

object_name

The name of the object for which you are creating the synonym. It can be one of the following:

- table
- view
- sequence
- stored procedure
- function
- package
- materialized view
- java class schema object
- user-defined object
- synonym

Example

Let's look at an example of how to create a synonym in Oracle.

For example:

```
CREATE PUBLIC SYNONYM suppliers
FOR app.suppliers;
```

This first CREATE SYNONYM example demonstrates how to create a synonym called *suppliers*. Now, users of other schemas can reference the table called *suppliers* without having to prefix the table name with the schema named *app*. For example:

```
SELECT *
FROM suppliers;
```

If this synonym already existed and you wanted to redefine it, you could always use the *OR REPLACE* phrase as follows:

```
CREATE OR REPLACE PUBLIC SYNONYM suppliers
FOR app.suppliers;
```

Drop synonym

Once a synonym has been created in Oracle, you might at some point need to drop the synonym.

Syntax

The syntax to drop a synonym in Oracle is:

```
DROP [PUBLIC] SYNONYM [schema .] synonym_name [force];
```

PUBLIC

Allows you to drop a public synonym. If you have specified *PUBLIC*, then you don't specify a *schema*.

force

It will force Oracle to drop the synonym even if it has dependencies. It is probably not a good idea to use *force* as it can cause invalidation of Oracle objects.

```
DROP PUBLIC SYNONYM suppliers;
```

UNIT V

8. PL/SQL

8.1 PL/SQL - TRIGGERS

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes −

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is −

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
   Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
  sal_diff number;
BEGIN
  sal_diff := :NEW.salary  - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
```

/

When the above code is executed at the SQL prompt, it produces the following result −

Trigger created.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

8.2 PL/SQL - PROCEDURES

A procedure is a **subprogram. It** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created −

- At the schema level
- Inside a package
- Inside a PL/SQL block

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the

database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter **'PL/SQL - Packages'**.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

- **Functions** − These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

This chapter is going to cover important aspects of a **PL/SQL procedure**. We will discuss **PL/SQL function** in the next chapter.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts −

S.No	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	**Executable Part** This is a mandatory part and contains statements that perform the designated action.
3	**Exception-handling** This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
  dbms_output.put_line('Hello World!');
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

Procedure created.

Executing a Standalone Procedure

A standalone procedure can be called in two ways −

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as –

```
EXECUTE greetings;
```

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

The procedure can also be called from another PL/SQL block −

```
BEGIN
  greetings;
```

```
END;
/
```

The above call will display −

```
Hello World
```

PL/SQL procedure successfully completed.

Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

```
DROP PROCEDURE procedure-name;
```

You can drop the greetings procedure by using the following statement −

```
DROP PROCEDURE greetings;
```

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms −

S.No	Parameter Mode & Description
1	**IN** An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.
2	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
3	**IN OUT** An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
DECLARE
  a number;
  b number;
  c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
  IF x < y THEN
    z:= x;
  ELSE
    z:= y;
  END IF;
END;
BEGIN
  a:= 23;
  b:= 45;
  findMin(a, b, c);
  dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
  Minimum of (23, 45) : 23

  PL/SQL procedure successfully completed.
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
DECLARE
  a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
 x := x * x;
END;
BEGIN
  a:= 23;
  squareNum(a);
  dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Square of (23): 529

PL/SQL procedure successfully completed.
```

8.3 PL/SQL - FUNCTIONS

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows −

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
  < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the PL/SQL Variables chapter –

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
  total number(2) := 0;
BEGIN
  SELECT count(*) into total
  FROM customers;

  RETURN total;
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

```
Function created.
```

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block −

```
DECLARE
  c number(2);
BEGIN
  c := totalCustomers();
  dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Total no. of Customers: 6

PL/SQL procedure successfully completed.
```

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
  a number;
  b number;
  c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
   z number;
BEGIN
  IF x > y THEN
    z:= x;
  ELSE
    Z:= y;
  END IF;
  RETURN z;
END;
BEGIN
  a:= 23;
  b:= 45;
  c := findMax(a, b);
  dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Maximum of (23,45): 45

PL/SQL procedure successfully completed.
```

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

```
n! = n*(n-1)!
  = n*(n-1)*(n-2)!
     ...
  = n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively −

```
DECLARE
  num number;
  factorial number;

FUNCTION fact(x number)
RETURN number
IS
  f number;
BEGIN
  IF x=0 THEN
    f := 1;
  ELSE
    f := x * fact(x-1);
  END IF;
RETURN f;
END;

BEGIN
  num:= 6;
  factorial := fact(num);
  dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

8.4 PL/SQL - PAKAGES

Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms.

A package will have two mandatory parts −

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just **DECLARES** the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
  PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Package created.
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from the code outside the package.

The **CREATE PACKAGE BODY** Statement is used for creating the package body. The following code snippet shows the package body declaration for the ***cust_sal*** package created above. I assumed that we already have CUSTOMERS table created in our database as mentioned in the PL/SQL - Variables chapter.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS

  PROCEDURE find_sal(c_id customers.id%TYPE) IS
  c_sal customers.salary%TYPE;
  BEGIN
    SELECT salary INTO c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line('Salary: '|| c_sal);
  END find_sal;
END cust_sal;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

Package body created.

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax −

package_name.element_name;

Consider, we already have created the above package in our database schema, the following program uses the ***find_sal*** method of the ***cust_sal*** package −

```
DECLARE
  code customers.id%type := &cc_id;
BEGIN
  cust_sal.find_sal(code);
END;
/
```

When the above code is executed at the SQL prompt, it prompts to enter the customer ID and when you enter an ID, it displays the corresponding salary as follows −

```
Enter value for cc_id: 1
Salary: 3000

PL/SQL procedure successfully completed.
```

Example

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records −

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  3000.00 |
|  2 | Khilan   |  25 | Delhi     |  3000.00 |
|  3 | kaushik  |  23 | Kota      |  3000.00 |
|  4 | Chaitali |  25 | Mumbai    |  7500.00 |
|  5 | Hardik   |  27 | Bhopal    |  9500.00 |
|  6 | Komal    |  22 | MP        |  5500.00 |
+----+----------+-----+-----------+----------+
```

The Package Specification

```
CREATE OR REPLACE PACKAGE c_package AS
  -- Adds a customer
  PROCEDURE addCustomer(c_id   customers.id%type,
  c_name  customerS.No.ame%type,
  c_age  customers.age%type,
  c_addr customers.address%type,
  c_sal  customers.salary%type);

  -- Removes a customer
  PROCEDURE delCustomer(c_id  customers.id%TYPE);
  --Lists all customers
  PROCEDURE listCustomer;

END c_package;
/
```

When the above code is executed at the SQL prompt, it creates the above package and displays the following result −

Package created.

Creating the Package Body

```
CREATE OR REPLACE PACKAGE BODY c_package AS
  PROCEDURE addCustomer(c_id  customers.id%type,
    c_name customerS.No.ame%type,
    c_age  customers.age%type,
    c_addr  customers.address%type,
    c_sal   customers.salary%type)
  IS
  BEGIN
    INSERT INTO customers (id,name,age,address,salary)
      VALUES(c_id, c_name, c_age, c_addr, c_sal);
  END addCustomer;

  PROCEDURE delCustomer(c_id   customers.id%type) IS
  BEGIN
    DELETE FROM customers
    WHERE id = c_id;
  END delCustomer;

  PROCEDURE listCustomer IS
  CURSOR c_customers is
    SELECT  name FROM customers;
  TYPE c_list is TABLE OF customerS.No.ame%type;
  name_list c_list := c_list();
  counter integer :=0;
  BEGIN
```

```
    FOR n IN c_customers LOOP
    counter := counter +1;
    name_list.extend;
    name_list(counter) := n.name;
    dbms_output.put_line('Customer(' ||counter|| ')'||name_list(counter));
    END LOOP;
  END listCustomer;

END c_package;
/
```

The above example makes use of the **nested table**. We will discuss the concept of nested table in the next chapter.

When the above code is executed at the SQL prompt, it produces the following result −

```
Package body created.
```

Using The Package

The following program uses the methods declared and defined in the package *c_package*.

```
DECLARE
  code customers.id%type:= 8;
BEGIN
  c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
  c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
  c_package.listcustomer;
  c_package.delcustomer(code);
  c_package.listcustomer;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
Customer(8): Subham
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
```

Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish

PL/SQL procedure successfully completed

8.5 PL/SQL - CURSORS

In this chapter, we will discuss the cursors in PL/SQL. Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes −

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or

DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

%ISOPEN

3 Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

%ROWCOUNT

4 Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example

We will be using the CUSTOMERS table we had created and used in the previous chapters.

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customers
  SET salary = salary + 500;
  IF sql%notfound THEN
    dbms_output.put_line('no customers selected');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers selected ');
  END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

6 customers selected

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated −

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is −

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

```
CURSOR c_customers IS
   SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
  c_id customers.id%type;
  c_name customerS.No.ame%type;
  c_addr customers.address%type;
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
  FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

8.6 PL/SQL - TRANSACTIONS

In this chapter, we will discuss the transactions in PL/SQL. A database **transaction** is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed, i.e., made permanent to the database or rolled back (undone) from the database.

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

Starting and Ending a Transaction

A transaction has a **beginning** and an **end**. A transaction starts when one of the following events take place −

- The first SQL statement is performed after connecting to the database.
- At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place −

- A **COMMIT** or a **ROLLBACK** statement is issued.
- A **DDL** statement, such as **CREATE TABLE** statement, is issued; because in that case a COMMIT is automatically performed.
- A **DCL** statement, such as a **GRANT** statement, is issued; because in that case a COMMIT is automatically performed.
- User disconnects from the database.
- User exits from **SQL*PLUS** by issuing the **EXIT** command, a COMMIT is automatically performed.
- SQL*Plus terminates abnormally, a **ROLLBACK** is automatically performed.
- A **DML** statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

Committing a Transaction

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is −

COMMIT;

For example,

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );

COMMIT;
```

Rolling Back Transactions

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

The general syntax for the ROLLBACK command is −

ROLLBACK [TO SAVEPOINT < savepoint_name>];

When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If you are not using **savepoint**, then simply use the following statement to rollback all the changes −

ROLLBACK;

Savepoints

Savepoints are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting savepoints within a long transaction, you can roll back to a checkpoint if required. This is done by issuing the **SAVEPOINT** command.

The general syntax for the SAVEPOINT command is −

SAVEPOINT < savepoint_name >;

For example

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Rajnish', 27, 'HP', 9500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (8, 'Riddhi', 21, 'WB', 4500.00 );
SAVEPOINT sav1;

UPDATE CUSTOMERS
SET SALARY = SALARY + 1000;
ROLLBACK TO sav1;

UPDATE CUSTOMERS
```

```
SET SALARY = SALARY + 1000
WHERE ID = 7;
UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 8;

COMMIT;
```

ROLLBACK TO sav1 − This statement rolls back all the changes up to the point, where you had marked savepoint sav1.

After that, the new changes that you make will start.

Automatic Transaction Control

To execute a **COMMIT** automatically whenever an **INSERT, UPDATE** or **DELETE** command is executed, you can set the **AUTOCOMMIT** environment variable as −

```
SET AUTOCOMMIT ON;
```

You can turn-off the auto commit mode using the following command –

```
SET AUTOCOMMIT OFF;
```