CSS3C11 – ADVANCED DATABASE MANAGEMENT SYSTEM

Unit I: Introduction - purpose of database systems, views of data - data abstraction,instances and schemas, data independence, data models - hierarchical data model, network data model, relational data model, ER model. Database languages - DDL, DML, transaction management, storage management, database administrator, database users, overall system structure. Relational data model - relational model concepts, keys, integrity constraints - domain constraints, key constraints, entity integrity constraints, referential integrity constraints. ER data model - basic concepts, constraints, keys, design issues, entity relationship diagram, weak entity sets, extended ER features, design of an ER database schema, reduction of an ER schema to tables. Relational algebra and calculus - relational algebra - selection and projection, set operations, renaming, joins, division. Relational calculus - tuple relational calculus, domain relational calculus. Expressive power of algebra and calculus

Unit II: Relational database design - anomalies in a database - functional dependency -lossless join and dependency- preserving decomposition - normalization - normal forms -first, second and third normal form - Boyce Codd normal form - multivalued, dependency - fourth normal form - join dependency - project join normal form - domain key normal form.

Unit III: Relational database query languages - basics of QBE and SQL. Data definition in SQL data types, creation, insertion, viewing, updation, deletion of tables, mo difying the structure of the tables, renaming, dropping of tables. Data constraints - I/O constraints, primary key, foreign key, unique key constraints, ALTER TABLE command database manipulation in SQL - computations done on table data - SELECT command, logical operators, range searching, pattern matching, grouping data from tables in SQL, GROUP BY, HAVING clauses. Joins - joining multiple tables, joining a table to it. DELETE -UPDATE. Views - creation, renaming the column of a view, destroys view. Program with SQL - data types Using SET and SELECT commands, procedural flow, IF, IF /ELSE, WHILE, GOTO, global variables. Security - locks, types of locks, levels of locks. Cursors -working with cursors, error handling, developing stored procedures,

CREATE, A LTER and DROP, passing and returning data to stored procedures, using stored procedures within queries, building user defined functions, creating and calling a scalar function, implementing triggers, creating triggers, multiple trigger interaction (Use MySQL as the RDBMS).

Unit IV: Transaction management, concurrency control and query processing - concept,definition and states of transactions, ACID properties - concurrency control, serializability -conflict serializability, view serializability, recoverability-recoverable schedules,non-cascading schedules, strict schedules. Concurrency control schemes - lockingtwo phase locking, deadlock, granularity, timestamp ordering protocol. Basics of query processing.

Unit V: Object Oriented Database Management Systems (OODBMS) - concepts, need for OODBMS, composite objects, issues in OODBMSs, advantages and disadvantages of OODBMS. Distributed databases - motivation - distributed database concepts, types of distribution, architecture of distributed databases, the design of distributed databases, distributed transactions, commit protocols for distributed databases.

MODULE 1

DBMS – PURPOSE

It is a collection of programs that enables the user to create and maintain a database. In other words, it is general-purpose software that provides the users with the processes of defining, constructing and manipulating the database for various applications.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Disadvantages in File Processing

A database management system (DBMS) is a collection of interrelated data and a set of programs to access those data. This is a collection of related data with an implicit meaning and hence is a database. The collection of data, usually referred to as the database, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient. By data, we mean known facts that can be recorded and that have implicit meaning.

For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book, or you may have stored

it on a diskette, using a personal computer and software such as DBASE IV or V, Microsoft ACCESS, or EXCEL.

The following are the main disadvantages of DBMS in File Processing:

- Data redundancy and inconsistency.
- Difficult in accessing data.
- Data isolation.
- Data integrity.
- Concurrent access is not possible.
- Security Problems.

Advantages of DBMS

Because information is so important in most organizations, computer scientists have developed a large body of concepts and techniques for managing data. These concepts and technique form the focus of this book.

- Data Independence.
- Efficient Data Access.
- Data Integrity and security.
- Data administration.

- Concurrent access and Crash recovery.
- Reduced Application Development Time.

VIEW OF DATA

in DBMS narrate how the data is visualized at each level of data abstraction? **Data abstraction** allow developers to keep complex data structures away from the users. The developers achieve this by hiding the complex data structures through **levels of abstraction**.

There is one more feature that should be kept in mind i.e. the **data independence**. While changing the data schema at one level of the database must not modify the data schema at the next level. In this section, we will discuss the view of data in DBMS with data abstraction, data independence, data schema in detail.

View of Data in DBMS

Data Abstraction

Data abstraction is **hiding the complex data structure** in order to **simplify the user's interface** of the system. It is done because many of the users interacting with the database system are not that much computer trained to understand the complex data structures of the database system.

To achieve data abstraction, we will discuss a **Three-Schema architecture** which abstracts the database at three levels discussed below:

Three-Schema Architecture:

The main objective of this architecture is to have an effective separation between the **user interface** and the **physical database**. So, the user never has to be concerned regarding the internal storage of the database and it has a simplified interaction with the database system.

The three-schema architecture defines the view of data at three levels:

1. Physical level (internal level)
2. Logical level (conceptual level)
3. View level (external level)

1. Physical Level/ Internal Level

The physical or the internal level schema describes **how the data is stored in the hardware**. It also describes how the data can be accessed. The physical level shows the data abstraction at the lowest level and it has **complex data structures**. Only the database administrator operates at this level.

2. Logical Level/ Conceptual Level

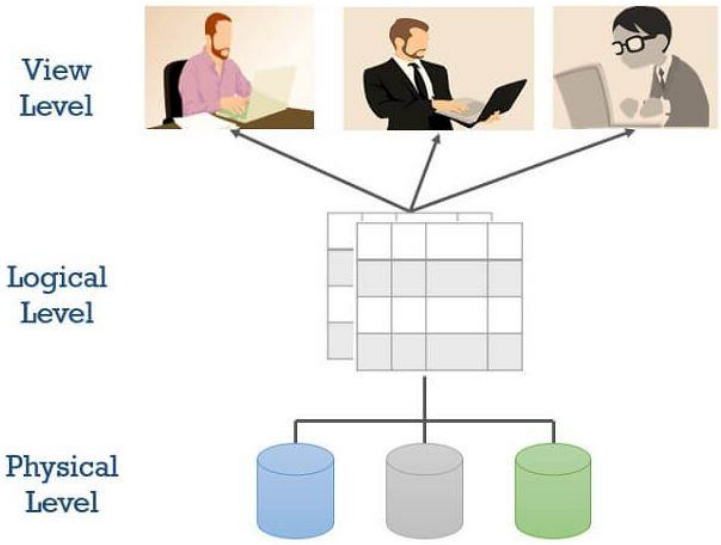
It is a level above the physical level. Here, the data is stored in the form of the **entity set**, **entities**, their **data types**, the **relationship** among the entity sets, **user operations** performed to retrieve or modify the data and certain **constraints on the**

data. Well adding constraints to the view of data adds the security. As users are restricted to access some particular parts of the database.

It is the developer and database administrator who operates at the logical or the conceptual level.

3. View Level/ User level/ External level

It is the highest level of data abstraction and exhibits only a part of the whole database. It exhibits the data in which the user is interested. The view level can describe many views of the same data. Here, the user retrieves the information using different application from the database.



Three-Schema Architecture

INSTANCES AND SCHEMAS

What is an instance?

We can define an instance as the information stored in the database at a particular point of time. Let us discuss it with the help of an example.

As we discussed above the database comprises of several entity sets and the relationship between them. Now, the data in the database keeps on changing with time. As we keep inserting or deleting the data to and from the database.

Now, at a particular time if we retrieve any information from the database then that corresponds to an instance.

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.

What is schema?

Whenever we talk about the database the developers have to deal with the definition of database and the data in the database.

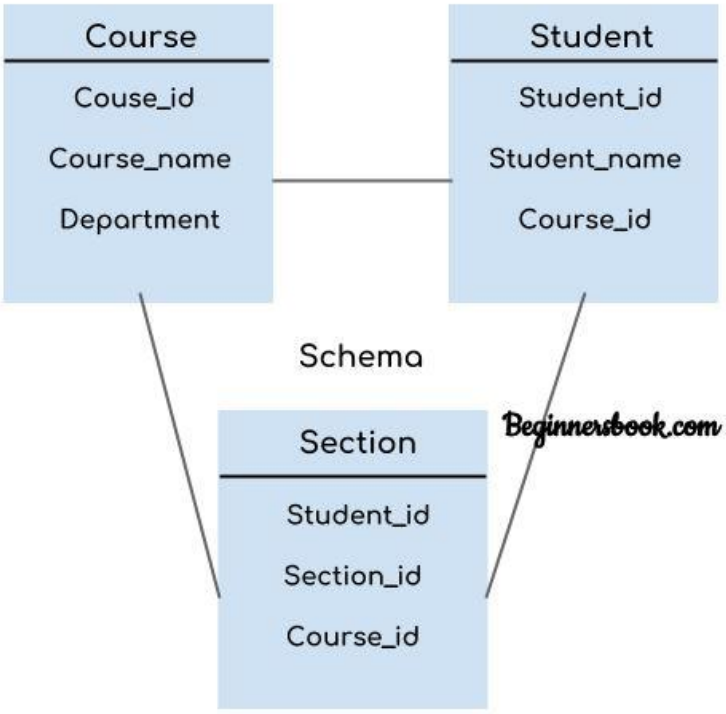
Definition of schema: Design of a database is called the schema. Schema is of three types: Physical schema, logical schema and view schema.

The definition of a database comprises of the description of what data it would contain what would be the relationship between the data. This definition is the database 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.



- View of data in DBMS describes the abstraction of data at three-level i.e. **physical** level, **logical** level, **view** level.

- The physical level of abstraction defines how data is **stored** in the storage and also reveals its access path.
    - Abstraction at the logical level describes **what data** would be stored in the database? what would be the **relation** between the data? and the **constraints** applied to the data.
    - The view level or external level of abstraction describes the **application** which the users use to retrieve the information from the database.
    - **Data independenc**e explains the extent to which data at a certain level can be modified without disturbing the data next higher levels.
    - An **instance** is the retrieval of information from the database at a certain point of time. An instance in a database keeps on **changing with time.**
    - **Schema** is the overall design of the entire database. Schema of the database is not changed frequently.

DATA INDEPENDENCE

Data independence defines the extent to which the data schema can be changed at one level without modifying the data schema at the next level. Data independence can be classified as shown below:

Logical Data Independence:

Logical data independence describes the degree up to which the logical or conceptual schema can be changed without modifying the external schema. Now, a question arises what is the need to change the data schema at a logical or conceptual level?

Well, the changes to data schema at the logical level are made either to **enlarge** or **reduce** the database by adding or deleting more entities, entity sets, or changing the constraints on data.

Physical Data Independence: Physical data independence defines the extent up to which the data schema can be changed at the physical or internal level without modifying the data schema at logical and view level.

the physical schema is changed if we add additional storage to the system or we reorganize some files to enhance the retrieval speed of the records.

DATA MODEL

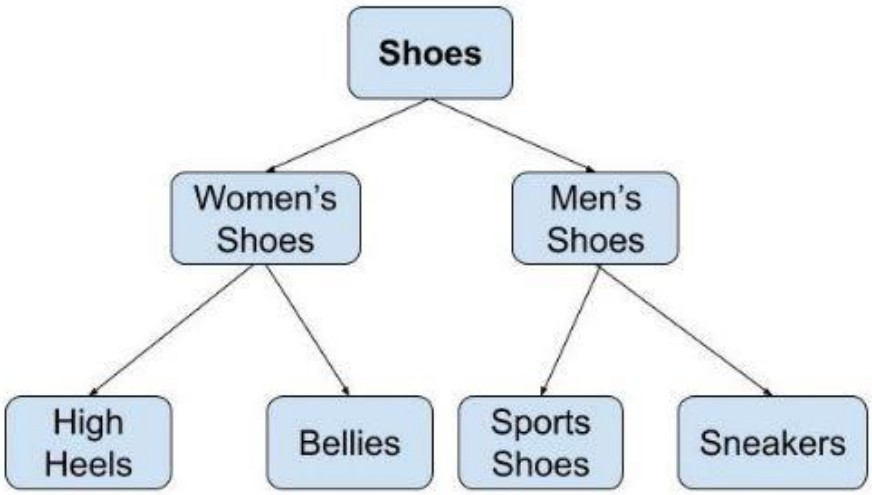
Data Model gives us an idea that how the final system will look like after its complete implementation. It defines the data elements and the relationships between the data elements. Data Models are used to show how data is stored, connected, accessed and updated in the database management system. Here, we use a set of symbols and text to represent the information so that members of the organisation can communicate and understand it. Though there are many data models being used nowadays but the Relational model is the most widely used model. Apart from the Relational model, there are many other types of data models about which we will study in details in this blog. Some of the Data Models in DBMS are:

1. Hierarchical Model
2. Network Model
3. Entity-Relationship Model

4. Relational Model

Hierarchical Model

Hierarchical Model was the first DBMS model. This model organises the data in the hierarchical tree structure. The hierarchy starts from the root which has root data and then it expands in the form of a tree adding child node to the parent node. This model easily represents some of the real-world relationships like food recipes, sitemap of a website etc. ***Example:*** We can represent the relationship between the shoes present on a shopping website in the following way:



Hierarchical Model

One-to-many relationship:

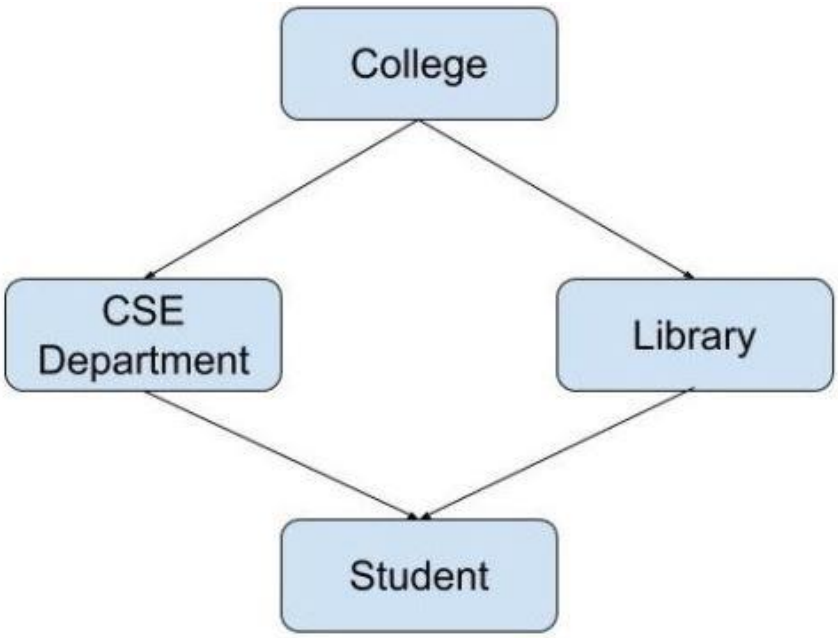
Parent-Child Relationship:

Deletion Problem

Pointers:

Network Model

This model is an extension of the hierarchical model. It was the most popular model before the relational model. This model is the same as the hierarchical model, the only difference is that a record can have more than one parent. It replaces the hierarchical tree with a graph. ***Example:*** In the example below we can see that node student has two parents i.e. CSE Department and Library. This was earlier not possible in the hierarchical model.



Network Model

Ability to Merge more Relationships:

Many paths:

Circular Linked List:

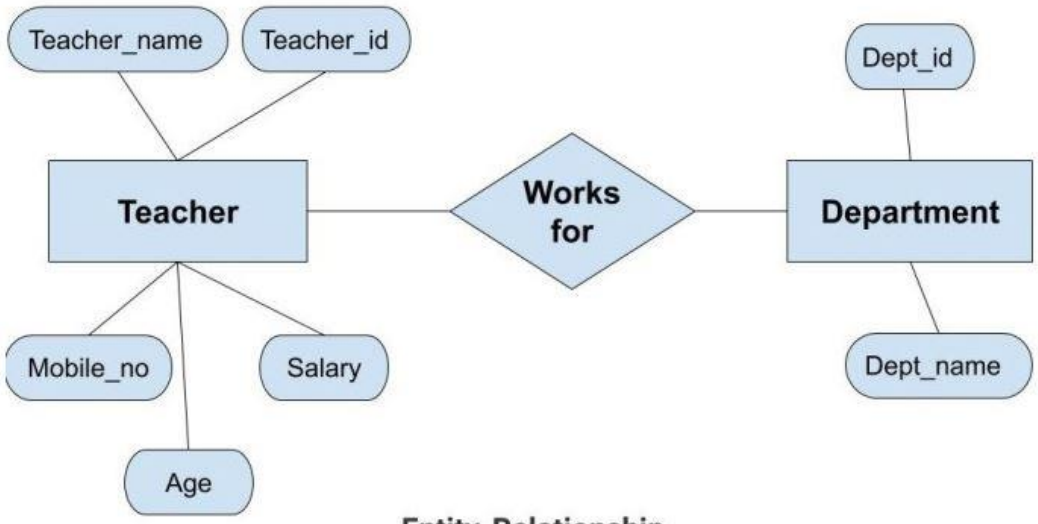
Entity-Relationship Model

Entity-Relationship Model or simply ER Model is a high-level data model diagram. In this model, we represent the real-world problem in the pictorial form to make it easy for the stakeholders to understand. It is also very easy for the developers to understand the system by just looking at the ER diagram. We use the ER diagram as a visual tool to represent an ER Model. ER diagram has the following three components:

Entities: Entity is a real-world thing. It can be a person, place, or even a concept

Attributes: An entity contains a real-world property called attribute. This is the characteristics of that attribute.

- ***Relationship:*** Relationship tells how two attributes are related. *Example:* Teacher works for a department.



Entity-Relationship Model

Graphical Representation for Better Understanding

ER Diagram

Database Design

Relational Model

Relational Model is the most widely used model. In this model, the data is maintained in the form of a two-dimensional table. All the information is stored in the form of row and columns. The basic structure of a relational model is tables. So, the tables are also called *relations* in the relational model. ***Example:*** In this example, we have an Employee table.

Emp_id	Emp_name	Job_name	Salary	Mobile_no	Dep_id	Project_id
AfterA001	John	Engineer	100000	9111037890	2	99
AfterA002	Adam	Analyst	50000	9587569214	3	100
AfterA003	Kande	Manager	890000	7895212355	2	65

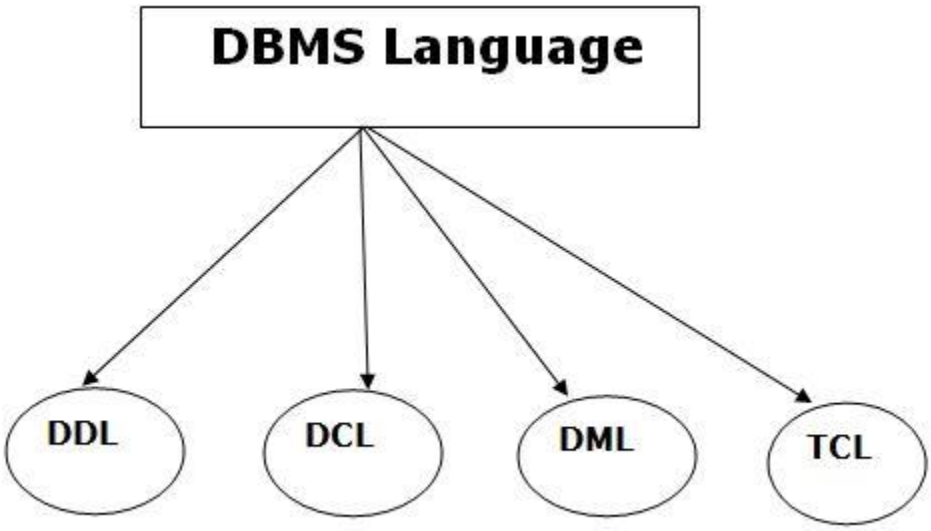
EMPLOYEE TABLE

Tuples: Each row in the table is called tuple. A row contains all the information about any instance of the object.

Attribute or field: Attributes are the property which defines the table or relation. The values of the attribute should be from the same domain.

Database Language

- A DBMS has appropriate languages and interfaces to express database queries and updates.
- Database languages can be used to read, store and update the data in the database.



DATA DEFINITION LANGUAGE

- **DDL** stands for **D**ata **D**efinition **L**anguage. It is used to define database structure or pattern.
- It is used to create schema, tables, indexes, constraints, etc. in the database.
- Using the DDL statements, you can create the skeleton of the database.
- Data definition language is used to store the information of metadata like the number of tables and schemas, their names, indexes, columns in each table, constraints, etc.

Here are some tasks that come under DDL:

- **Create:** It is used to create objects in the database.
- **Alter:** It is used to alter the structure of the database.
- **Drop:** It is used to delete objects from the database.
- **Truncate:** It is used to remove all records from a table.
- **Rename:** It is used to rename an object.
- **Comment:** It is used to comment on the data dictionary.

DATA MANIPULATION LANGUAGE

DML stands for **D**ata **M**anipulation **L**anguage. It is used for accessing and manipulating data in a database. It handles user requests.

Here are some tasks that come under DML:

- **Select:** It is used to retrieve data from a database.
- **Insert:** It is used to insert data into a table.
- **Update:** It is used to update existing data within a table.
- **Delete:** It is used to delete all records from a table.
- **Merge:** It performs UPSERT operation, i.e., insert or update operations.
- **Call:** It is used to call a structured query language or a Java subprogram.
- **Explain Plan:** It has the parameter of explaining data.
- **Lock Table:** It controls concurrency.

3. DATA CONTROL LANGUAGE

- **DCL** stands for **D**ata **C**ontrol **L**anguage. It is used to retrieve the stored or saved data.
- The DCL execution is transactional. It also has rollback parameters.

(But in Oracle database, the execution of data control language does not have the feature of rolling back.)

Here are some tasks that come under DCL:

- **Grant:** It is used to give user access privileges to a database.
- **Revoke:** It is used to take back permissions from the user.

There are the following operations which have the authorization of Revoke:

CONNECT, INSERT, USAGE, EXECUTE, DELETE, UPDATE and SELECT.

4. TRANSACTION CONTROL LANGUAGE

TCL is used to run the changes made by the DML statement. TCL can be grouped into a logical transaction.

Here are some tasks that come under TCL:

- **Commit:** It is used to save the transaction on the database.
- **Storage Management** is defined as it refers to the management of the data storage equipment's that are used to store the user/computer generated data. Hence it is a tool or set of processes used by an administrator to keep your data and storage equipment's safe.
- Storage management is a process for users to optimize the use of storage devices and to protect the integrity of data for any media on which it resides and the category of storage management generally contain the different type of subcategories covering aspects such as security, virtualization and more, as well as different types of provisioning or

automation, which is generally made up the entire storage management software market.

- **Rollback:** It is used to restore the database to original since the last Commit.

Databases are stored in file formats, which contain records. At physical level, the actual data is stored in electromagnetic format on some device. These storage devices can be broadly categorized into three types −

- **Primary Storage** − The memory storage that is directly accessible to the CPU comes under this category. CPU's internal memory (registers), fast memory (cache), and main memory (RAM) are directly accessible to the CPU, as they are all placed on the motherboard or CPU chipset. This storage is typically very small, ultra-fast, and volatile. Primary storage requires continuous power supply in order to maintain its state. In case of a power failure, all its data is lost.
- **Secondary Storage** − Secondary storage devices are used to store data for future use or as backup. Secondary storage includes memory devices that are not a part of the CPU chipset or motherboard, for example, magnetic disks, optical disks (DVD, CD, etc.), hard disks, flash drives, and magnetic tapes.
- **Tertiary Storage** − Tertiary storage is used to store huge volumes of data. Since such storage devices are external to the computer system, they are the slowest in speed. These storage devices are mostly used to take the back up of an entire system. Optical disks and magnetic tapes are widely used as tertiary storage.

DBA

A Database Administrator (DBA) is individual or person responsible for controlling, maintenance, coordinating, and operation of database management system. Managing, securing, and taking care of database system is prime responsibility.

They are responsible and in charge for authorizing access to database, coordinating, capacity, planning, installation, and monitoring uses and for acquiring and gathering software and hardware resources as and when needed. Their role also varies from configuration, database design, migration, security, troubleshooting, backup, and data recovery. Database administration is major and key function in any firm or organization that is relying on one or more databases. They are overall commander of Database system.

Types of Database Administrator (DBA) :

Administrative DBA –

Their job is to maintain server and keep it functional. They are concerned with data backups, security, trouble shooting, replication, migration etc.

Data Warehouse DBA –

Assigned earlier roles, but held accountable for merging data from various sources into data warehouse. They also design warehouse, with cleaning and scrubs data prior to loading.

Development DBA –

They build and develop queries, stores procedure, etc. that meets firm or organization needs. They are par at programmer.

Application DBA –

They particularly manages all requirements of application components that interact with database and accomplish activities such as application installation and coordinating, application upgrades, database cloning, data load process management, etc.

Architect –

They are held responsible for designing schemas like building tables. They work to build structure that meets organisation needs. The design is further used by developers and development DBAs to design and implement real application.

OLAP DBA –

They design and builds multi-dimensional cubes for determination support or OLAP systems.

DATABASE USERS

Different types of Database Users

Database users are categorized based up on their interaction with the data base

These are seven types of data base users in DBMS.

Database Administrator (DBA) :

Database Administrator (DBA) is a person/team who defines the schema and also controls the 3 levels of database.

The DBA will then create a new account id and password for the user if he/she need to access the data base.

DBA is also responsible for providing security to the data base and he allows only the authorized users to access/modify the data base.

DBA also monitors the recovery and back up and provide technical support.

The DBA has a DBA account in the DBMS which called a system or superuser account.

DBA repairs damage caused due to hardware and/or software failures.

Naive / Parametric End Users :

Parametric End Users are the unsophisticated who don't have any DBMS knowledge but they frequently use the data base applications in their daily life to get the desired results.

For examples, Railway's ticket booking users are naive users. Clerks in any bank is a naive user because they don't have any DBMS knowledge but they still use the database and perform their given task.

System Analyst :

System Analyst is a user who analyzes the requirements of parametric end users. They check whether all the requirements of end users are satisfied.

Sophisticated Users :

Sophisticated users can be engineers, scientists, business analyst, who are familiar with the database. They can develop their own data base applications according to their requirement. They don't write the program code but they interact the data base by writing SQL queries directly through the query processor.

Data Base Designers :

Data Base Designers are the users who design the structure of data base which includes tables, indexes, views, constraints, triggers, stored procedures. He/she controls what data must be stored and how the data items to be related.

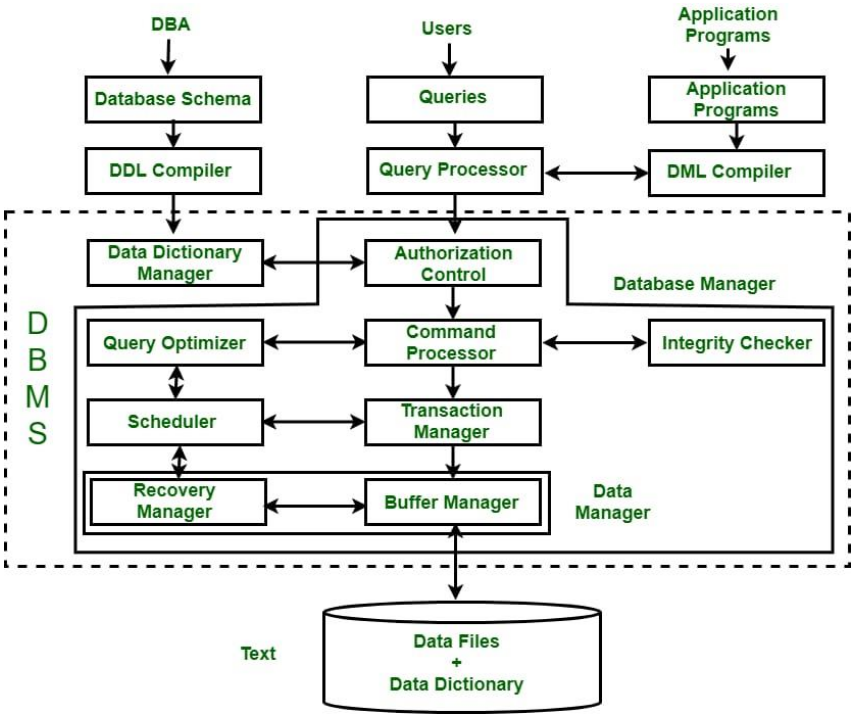
Application Program :

Application Program are the back end programmers who writes the code for the application program

RELATIONAL DATA MODEL

Relational data model is the primary data model, which is used widely around the world for data storage and processing. This model is simple and it has all the properties and capabilities required to process data with storage efficiency.

Structure of Database Management System



Database Management System (DBMS) is a software that allows access to data stored in a database and provides an easy and effective method of –

Defining the information.

Storing the information.

Manipulating the information.

Protecting the information from system crashes or data theft.

Differentiating access permissions for different users.

The database system is divided into three components: Query Processor, Storage Manager, and Disk Storage. These are explained as following below.

1. Query Processor :

It interprets the requests (queries) received from end user via an application program into instructions. It also executes the user request which is received from the DML compiler.

Query Processor contains the following components –

DML Compiler –

It processes the DML statements into low level instruction (machine language), so that they can be executed.

DDL Interpreter –

It processes the DDL statements into a set of table containing meta data (data about data).

Embedded DML Pre-compiler –

It processes DML statements embedded in an application program into procedural calls.

Query Optimizer –

It executes the instruction generated by DML Compiler.

2. Storage Manager :

Storage Manager is a program that provides an interface between the data stored in the database and the queries received. It is also known as Database Control System. It maintains the consistency and integrity of the database by applying the constraints and executes the DCL statements. It is responsible for updating, storing, deleting, and retrieving data in the database.

It contains the following components –

Authorization Manager –

It ensures role-based access control, i.e,. checks whether the particular person is privileged to perform the requested operation or not.

Integrity Manager –

It checks the integrity constraints when the database is modified.

Transaction Manager –

It controls concurrent access by performing the operations in a scheduled way that it receives the transaction. Thus, it ensures that the database remains in the consistent state before and after the execution of a transaction.

File Manager –

It manages the file space and the data structure used to represent information in the database.

Buffer Manager –

It is responsible for cache memory and the transfer of data between the secondary storage and main memory.

3. Disk Storage :

It contains the following components –

Data Files –

It stores the data.

Data Dictionary –

It contains the information about the structure of any database object. It is the repository of information that governs the metadata.

Indices –

It provides faster retrieval of data item

Concepts

Tables − In relational data model, relations are saved in the format of Tables. This format stores the relation among entities. A table has rows and columns, where rows represents records and columns represent the attributes.

Tuple − A single row of a table, which contains a single record for that relation is called a tuple.

Relation instance − A finite set of tuples in the relational database system represents relation instance. Relation instances do not have duplicate tuples.

Relation schema − A relation schema describes the relation name (table name), attributes, and their names.

Relation key − Each row has one or more attributes, known as relation key, which can identify the row in the relation (table) uniquely.

Attribute domain − Every attribute has some pre-defined value scope, known as attribute domain.

CONSTRAINTS

Every relation has some conditions that must hold for it to be a valid relation. These conditions are called Relational Integrity Constraints. There are three main integrity constraints −

Key constraints

Domain constraints

Referential integrity constraints

Key Constraints

There must be at least one minimal subset of attributes in the relation, which can identify a tuple uniquely. This minimal subset of attributes is called key for that relation. If there are more than one such minimal subsets, these are called candidate keys.

Key constraints force that −

in a relation with a key attribute, no two tuples can have identical values for key attributes.

a key attribute can not have NULL values.

Key constraints are also referred to as Entity Constraints.

Domain Constraints

Attributes have specific values in real-world scenario. For example, age can only be a positive integer. The same constraints have been tried to employ on the attributes of a relation. Every attribute is bound to have a specific range of values. For example, age cannot be less than zero and telephone numbers cannot contain a digit outside 0-9.

Entity Integrity Constraint

We can use any of the following two constraints as a part of this category

Primary Key

Unique

An entity integrity constraint is used to identify/recognize the row or record of a table uniquely.

Primary Key Constraint

When this constraint is associated with the column of a table it will not allow NULL values into the column and it will maintain unique values as part of the table.

We can add only one Primary Key constraint on a table.

We can say that a Primary Key constraint is a combination of Unique and NOT NULL constraints.

Unique Constraint

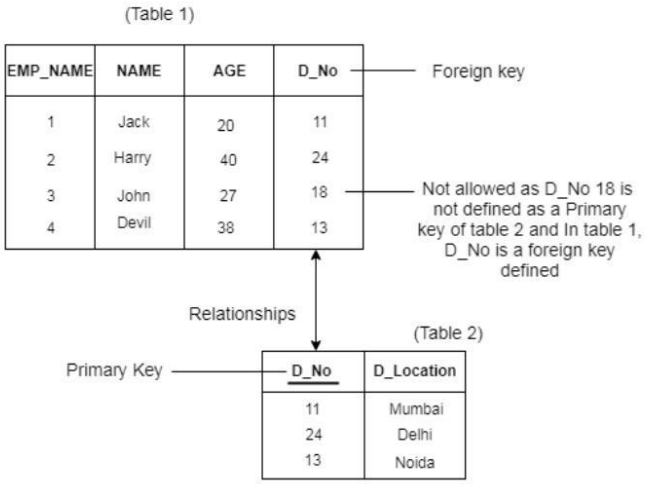
When this constraint is associated with the column(s) of a table it will not allow us to store a repetition of data/values in the column, but Unique constraint allows ONE NULL value. More than one NULL value is also considered as repetition, hence it does not store a repetition of NULL values also.

Referential integrity Constraints

Referential integrity constraints work on the concept of Foreign Keys. A foreign key is a key attribute of a relation that can be referred in other relation.

Referential integrity constraint states that if a relation refers to a key attribute of a different or same relation, then that key element must exist.

Example:



. Referential Integrity Constraints

A referential integrity constraint is specified between two tables.

In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

ER DATA model

o ER model stands for an Entity-Relationship model. It is a high-level data model. This model is used to define the data elements and relationship for a specified system.

o It develops a conceptual design for the database. It also develops a very simple and easy to design view of data.

o In ER modeling, the database structure is portrayed as a diagram called an entity-relationship diagram.

For example, Suppose we design a school database. In this database, the student will be an entity with attributes like address, name, id, age, etc. The address can be another entity with attributes like city, street name, pin code, etc and there will be a relationship between them.

Component of ER Diagram

1. Entity:

An entity may be any object, class, person or place. In the ER diagram, an entity can be represented as rectangles. Consider an organization as an example-manager, product, employee, department etc. can be taken as an entity.

a. Weak Entity

An entity that depends on another entity called a weak entity. The weak entity doesn't contain any key attribute of its own. The weak entity is represented by a double rectangle.

2. Attribute

The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute.

For example, id, age, contact number, name, etc. can be attributes of a student.

a. Key Attribute

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined.

b. Composite Attribute - An attribute that composed of many other attributes is known as a composite attribute. The composite attribute is represented by an ellipse, and those ellipses are connected with an ellipse.

c. Multivalued Attribute

An attribute can have more than one value. These attributes are known as a multivalued attribute. The double oval is used to represent multivalued attribute.

For example, a student can have more than one phone number.

d. Derived Attribute

An attribute that can be derived from other attribute is known as a derived attribute. It can be represented by a dashed ellipse.

For example, A person's age changes over time and can be derived from another attribute like Date of birth.

3. Relationship

A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent the relationship.

Types of relationship are as follows:

a. One-to-One Relationship

When only one instance of an entity is associated with the relationship, then it is known as one to one relationship.

For example, A female can marry to one male, and a male can marry to one female.

b. One-to-many relationship

When only one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then this is known as a one-to-many relationship.

For example, Scientist can invent many inventions, but the invention is done by the only specific scientist.

c. Many-to-one relationship

When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.

For example, Student enrolls for only one course, but a course can have many students.

d. Many-to-many relationship

When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.

For example, Employee can assign by many projects and project can have many employees

Constraints in an entity-relationship model

Skip to end of metadataGo to start of metadata

Constraints are only available in the NIAM view, and they are used for modeling limitations on the relations between entities. There are three types of constraints:

Uniqueness constraint for indicating the key of an entity for uniquely identifying the entity. Is linked to attributes of an entity.

Exclusion constraint for indicating that each of the entities excludes the other. Is linked to predicates of a fact.

Completeness constraint for indicating that two entities together complete the collection. Is linked to predicates of a fact.

Keys

Keys are the attributes of the entity, which uniquely identifies the record of the entity. For example STUDENT_ID identifies individual students, passport#, license # etc.

As we have seen already, there are different types of keys in the database.

Super Key is the one or more attributes of the entity, which uniquely identifies the record in the database.

Candidate Key is one or more set of keys of the entity. For a person entity, his SSN, passport#, license# etc can be a super key.

Primary Key is the candidate key, which will be used to uniquely identify a record by the query. Though a person can be identified using his SSN, passport# or license#, one can choose any one of them as primary key to uniquely identify a person. Rest of them will act as a candidate key.

Foreign Key of the entity attribute in the entity which is the primary key of the related entity. Foreign key helps to establish the mapping between two or more entities.

ER Design Issues

In the previous sections of the data modeling, we learned to design an ER diagram. We also discussed different ways of defining entity sets and relationships among them. We also understood the various designing shapes that represent a relationship, an entity, and its attributes. However, users often mislead the concept of the elements and the design process of the ER diagram. Thus, it leads to a complex structure of the ER diagram and certain issues that does not meet the characteristics of the real-world enterprise model.

Here, we will discuss the basic design issues of an ER database schema in the following points:

1) Use of Entity Set vs Attributes

The use of an entity set or attribute depends on the structure of the real-world enterprise that is being modelled and the semantics associated with its attributes. It leads to a mistake when the user use the primary key of an entity set as an attribute of another entity set. Instead, he should use the relationship to do so. Also, the primary key attributes are implicit in the relationship set, but we designate it in the relationship sets.

2) Use of Entity Set vs. Relationship Sets

It is difficult to examine if an object can be best expressed by an entity set or relationship set. To understand and determine the right use, the user need to designate a relationship set for describing an action that occurs in-between the entities. If there is a requirement of representing the object as a relationship set, then its better not to mix it with the entity set.

3) Use of Binary vs n-ary Relationship Sets

Generally, the relationships described in the databases are binary relationships. However, non-binary relationships can be represented by several binary relationships. For example, we can create and represent a ternary relationship 'parent' that may relate to a child, his father, as well as his mother. Such relationship can also be represented by two binary relationships i.e, mother and father, that may relate to their child. Thus, it is possible to represent a non-binary relationship by a set of distinct binary relationships.

4) Placing Relationship Attributes

The cardinality ratios can become an affective measure in the placement of the relationship attributes. So, it is better to associate the attributes of one-to-one or one-to-many relationship sets with any participating entity sets, instead of any relationship set. The decision of placing the specified attribute as a relationship or entity attribute should possess the charactestics of the real world enterprise that is being modelled.

For example, if there is an entity which can be determined by the combination of participating entity sets, instead of determing it as a separate entity. Such type of attribute must be associated with the many-to-many relationship sets.

Thus, it requires the overall knowledge of each part that is involved inb desgining and modelling an ER diagram. The basic requirement is to analyse the real-world enterprise and the connectivity of one entity or attribute with other.

ER DIAGRAM

An ER diagram shows the relationship among entity sets. An entity set is a group of similar entities and these entities can have attributes. In terms of DBMS, an entity is a table or attribute of a table in database, so by showing relationship among tables and their attributes, ER diagram shows the complete logical structure of a database. Lets have a look at a simple ER diagram to understand this concept.

A simple ER Diagram:

In the following diagram we have two entities Student and College and their relationship. The relationship between Student and College is many to one as a

college can have many students however a student cannot study in multiple colleges at the same time. Student entity has attributes such as Stu_Id, Stu_Name & Stu_Addr and College entity has attributes such as Col_ID & Col_Name.

E-R Diagram

Here are the geometric shapes and their meaning in an E-R Diagram. We will discuss these terms in detail in the next section(Components of a ER Diagram) of this guide so don’t worry too much about these terms now, just go through them once.

Rectangle: Represents Entity sets.

Ellipses: Attributes

Diamonds: Relationship Set

Lines: They link attributes to Entity Sets and Entity sets to Relationship Set

Double Ellipses: Multivalued Attributes

Dashed Ellipses: Derived Attributes

Double Rectangles: Weak Entity Sets

Double Lines: Total participation of an entity in a relationship set

Components of a ER Diagram

ER Diagram Components

Weak Entity Set in ER diagrams

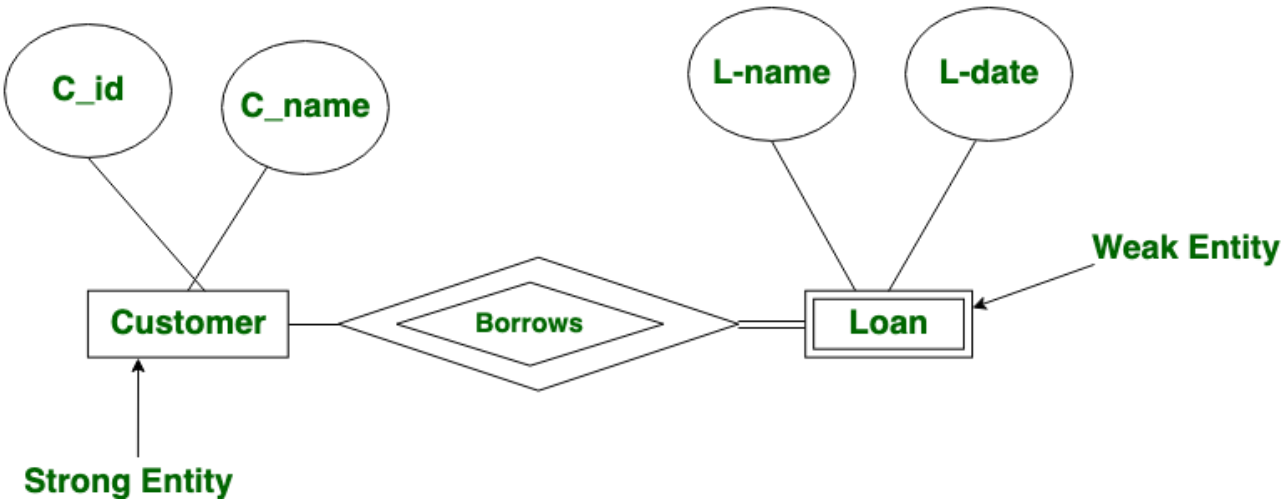
An entity type should have a key attribute which uniquely identifies each entity in the entity set, but there exists some entity type for which key attribute can't be defined. These are called Weak Entity type.

The entity sets which do not have sufficient attributes to form a primary key are known as **weak entity sets** and the entity sets which have a primary key are known as strong entity sets.

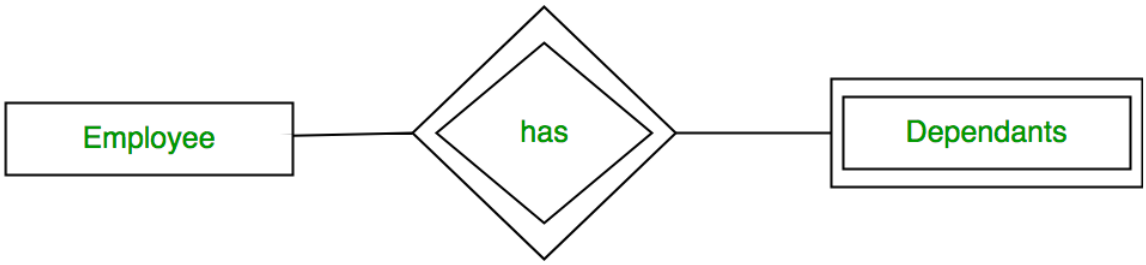
As the weak entities do not have any primary key, they cannot be identified on their own, so they depend on some other entity (known as owner entity). The weak entities have total participation constraint (existence dependency) in its identifying relationship with owner identity. Weak entity types have partial keys. Partial Keys are set of attributes with the help of which the tuples of the weak entities can be distinguished and identified.

Note – Weak entity always has total participation but Strong entity may not have total participation.

Weak entity is **depend on strong entity** to ensure the existence of weak entity. Like strong entity, weak entity does not have any primary key, It has partial discriminator key. Weak entity is represented by double rectangle. The relation etween one strong and one weak entity is represented by double diamond

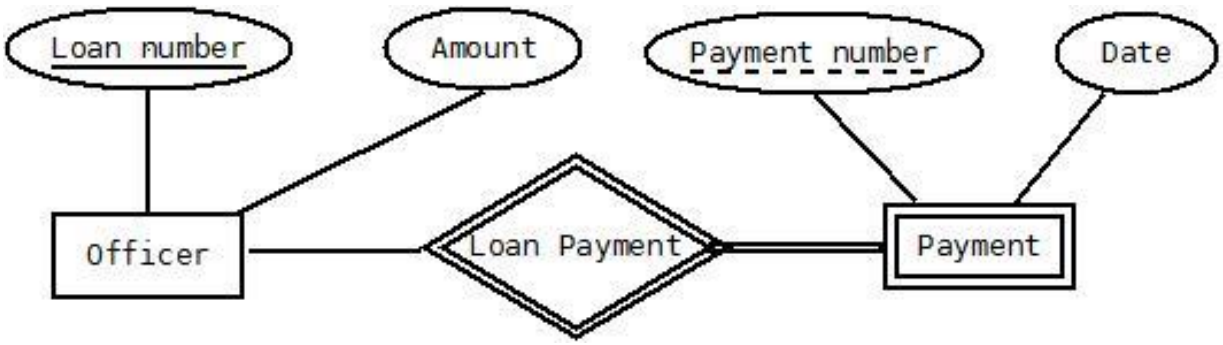


Weak entities are represented with **double rectangular** box in the ER Diagram and the identifying relationships are represented with double diamond. Partial Key attributes are represented with dotted lines.



Example-1:

In the below ER Diagram, 'Payment' is the weak entity. 'Loan Payment' is the identifying relationship and 'Payment Number' is the partial key. Primary Key of the Loan along with the partial key would be used to identify the records.



Extended Entity-Relationship (EE-R) Model

EER is a high-level data model that incorporates the extensions to the original ER model. Enhanced ERD are high level models that represent the requirements and complexities of complex database.

In addition to ER model concepts EE-R includes −

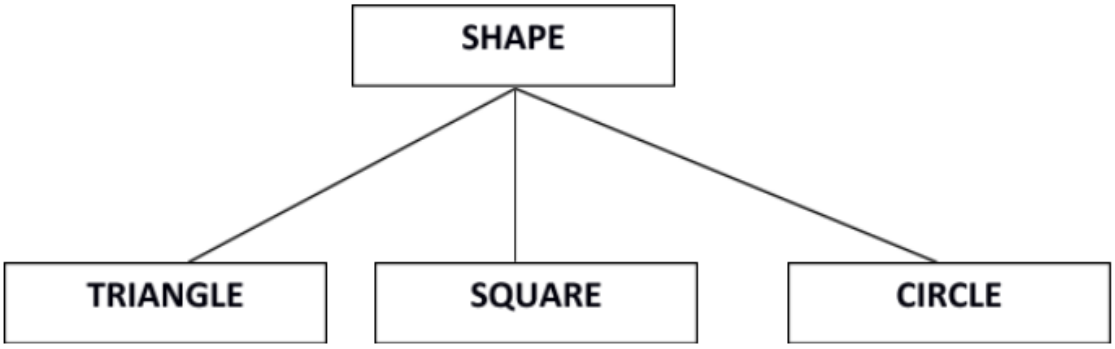
- Subclasses and Super classes.
- Specialization and Generalization.
- Category or union type.
- Aggregation.

These concepts are used to create EE-R diagrams.

Subclasses and Super class:

Super class is an entity that can be divided into further subtype.

For **example** − consider Shape super class.

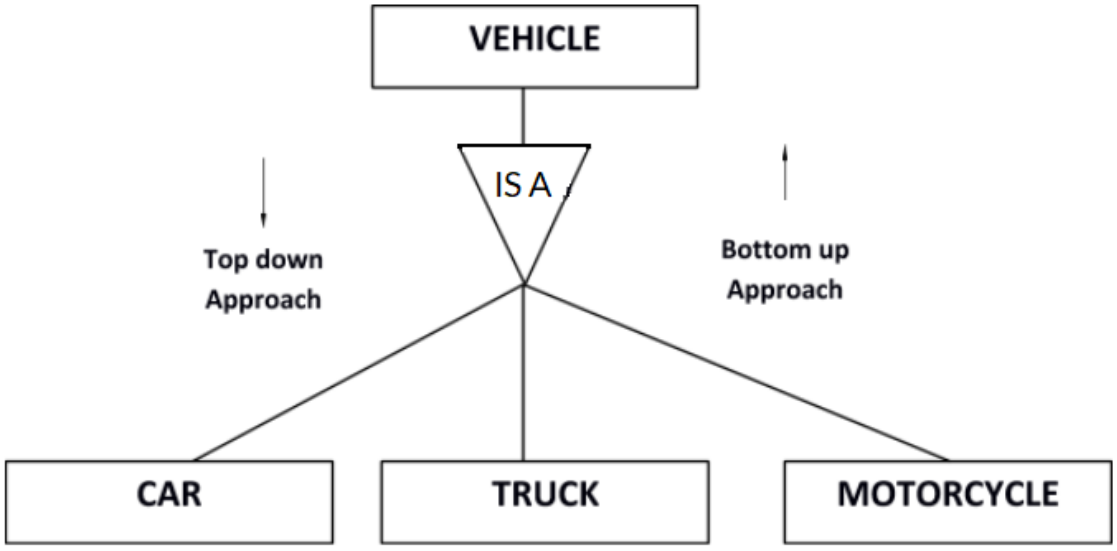


Super class shape has sub groups: Triangle, Square and Circle.

Sub classes are the group of entities with some unique attributes.Sub class inherits the properties and attributes from super class.

Specialization and Generalization:

Generalization is a process of generalizing an entity which contains generalized attributes or properties of generalized entities.



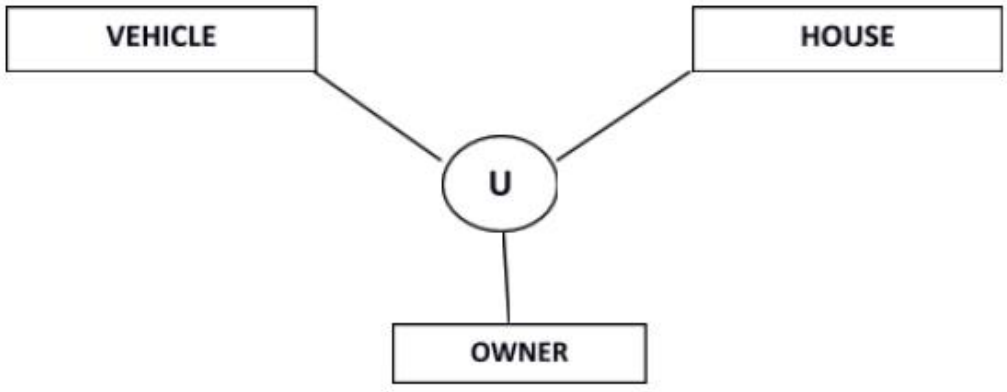
It is a Bottom up process i.e. consider we have 3 sub entities Car, Truck and Motorcycle. Now these three entities can be generalized into one super class named as Vehicle.

Specialization is a process of identifying subsets of an entity that share some different characteristic. It is a top down approach in which one entity is broken down into low level entity.

In above example Vehicle entity can be a Car, Truck or Motorcycle.

Category or Union:

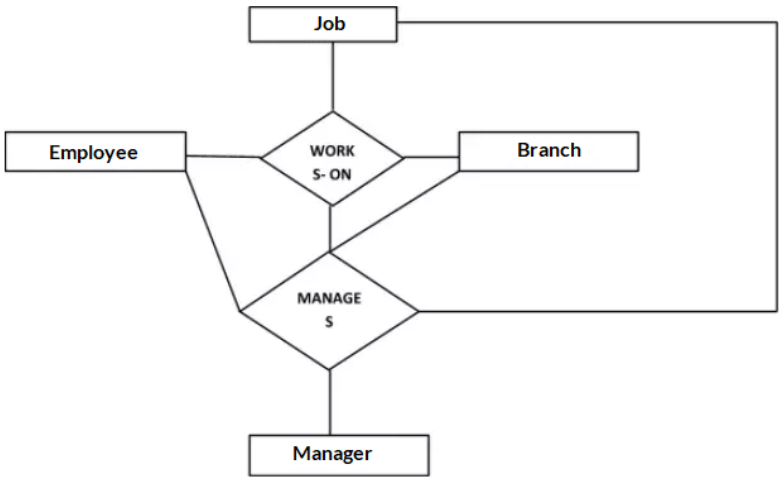
Relationship of one super or sub class with more than one super class.



Owner is the subset of two super class: Vehicle and House.

Aggregation:

Represents relationship between a whole object and its component.

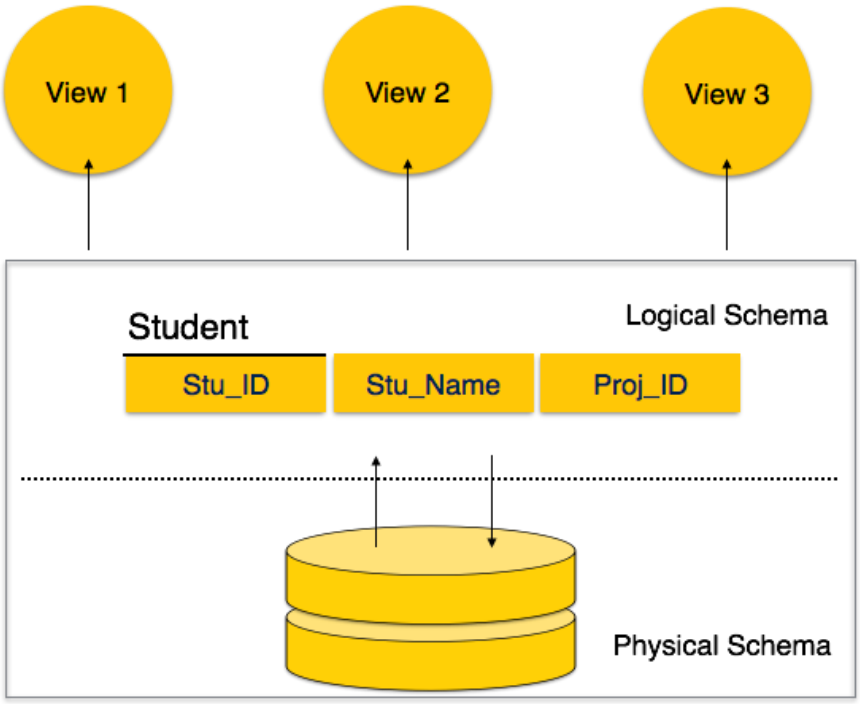


Consider a ternary relationship Works_On between Employee, Branch and Manager. Now the best way to model this situation is to use aggregation, So, the relationship-set, Works_On is a higher level entity-set. Such an entity-set is treated in the same manner as any other entity-set. We can create a binary relationship, Manager, between Works_On and Manager to represent who manages what tasks.

Database Schema

A database schema is the skeleton structure that represents the logical view of the entire database. It defines how the data is organized and how the relations among them are associated. It formulates all the constraints that are to be applied on the data.

A database schema defines its entities and the relationship among them. It contains a descriptive detail of the database, which can be depicted by means of schema diagrams. It’s the database designers who design the schema to help programmers understand the database and make it useful.



A database schema can be divided broadly into two categories −

- **Physical Database Schema** − This schema pertains to the actual storage of data and its form of storage like files, indices, etc. It defines how the data will be stored in a secondary storage.
- **Logical Database Schema** − This schema defines all the logical constraints that need to be applied on the data stored. It defines tables, views, and integrity constraints.

Database Instance

It is important that we distinguish these two terms individually. Database schema is the skeleton of database. It is designed when the database doesn't exist at all.

Once the database is operational, it is very difficult to make any changes to it. A database schema does not contain any data or information.

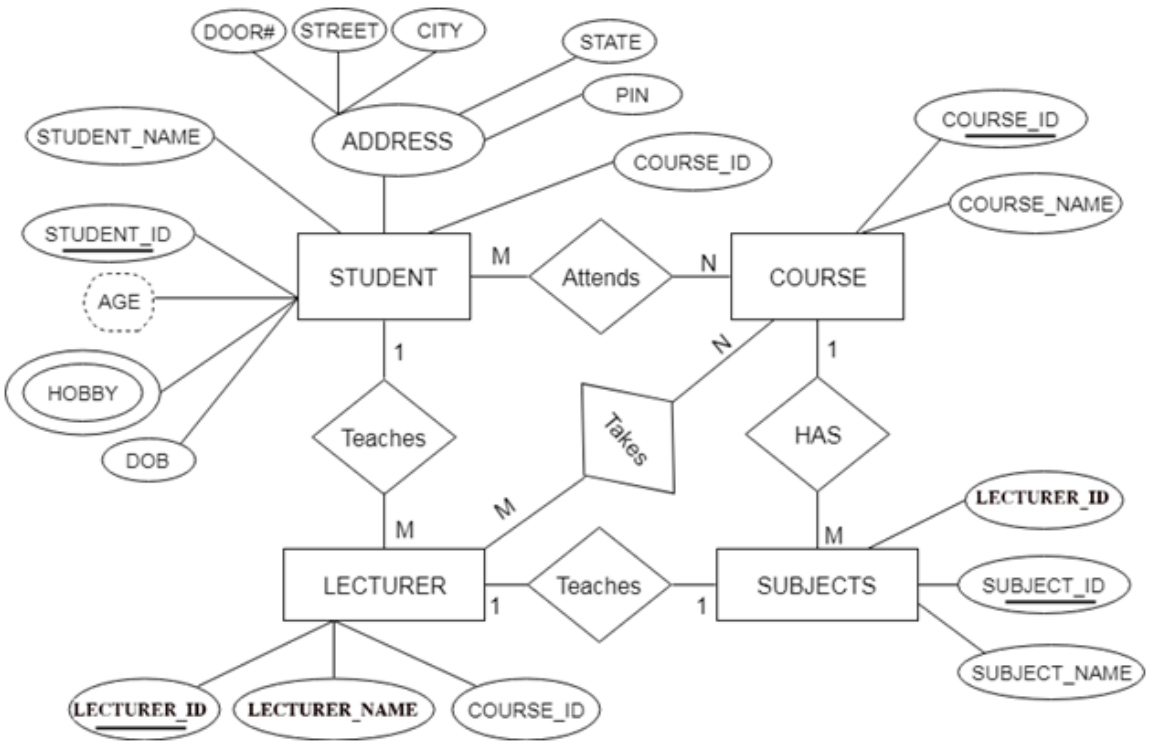
A database instance is a state of operational database with data at any given time. It contains a snapshot of the database. Database instances tend to change with time. A DBMS ensures that its every instance (state) is in a valid state, by diligently following all the validations, constraints, and conditions that the database designers have imposed.

Reduction of ER diagram to Table

The database can be represented using the notations, and these notations can be reduced to a collection of tables.

In the database, every entity set or relationship set can be represented in tabular form.

The ER diagram is given below:



There are some points for converting the ER diagram to the table:

- **Entity type becomes a table.**

In the given ER diagram, LECTURE, STUDENT, SUBJECT and COURSE forms individual tables.

- **All single-valued attribute becomes a column for the table.**

In the STUDENT entity, STUDENT_NAME and STUDENT_ID form the column of STUDENT table. Similarly, COURSE_NAME and COURSE_ID form the column of COURSE table and so on.

- **A key attribute of the entity type represented by the primary key.**

In the given ER diagram, COURSE_ID, STUDENT_ID, SUBJECT_ID, and LECTURE_ID are the key attribute of the entity.

- **The multivalued attribute is represented by a separate table.**

In the student table, a hobby is a multivalued attribute. So it is not possible to represent multiple values in a single column of STUDENT table. Hence we create a table STUD_HOBBY with column name STUDENT_ID and HOBBY. Using both the column, we create a composite key.

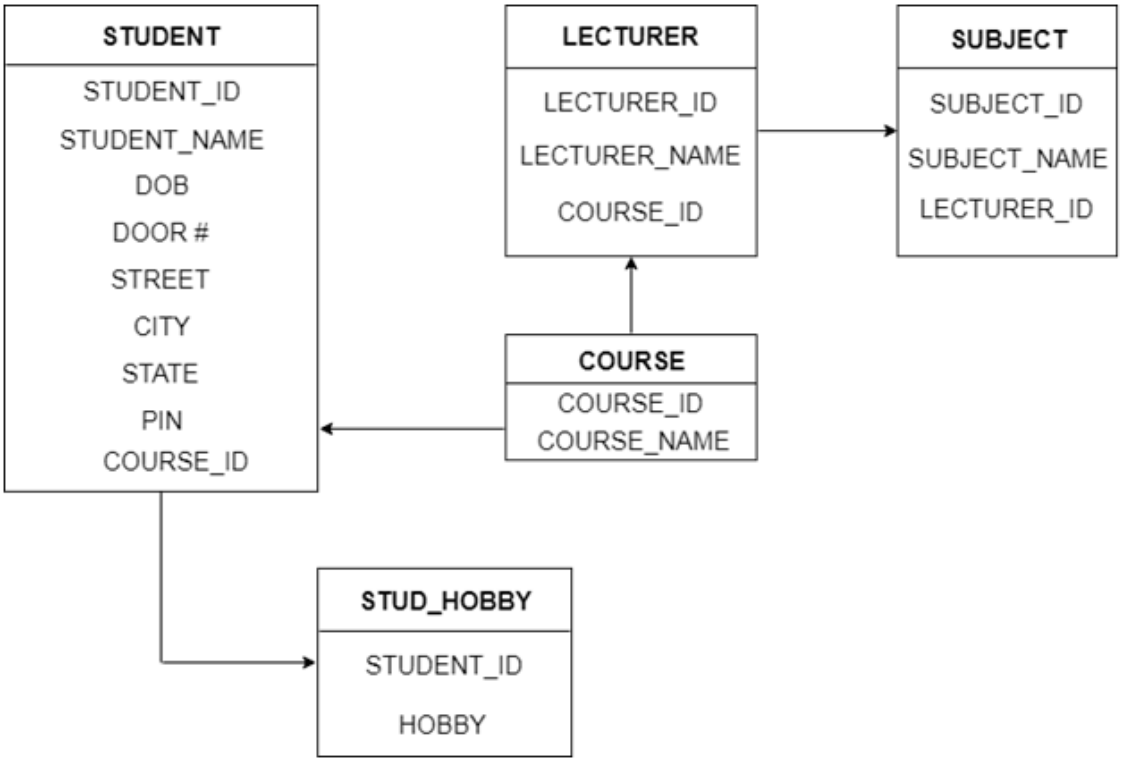
- **Composite attribute represented by components.**

In the given ER diagram, student address is a composite attribute. It contains CITY, PIN, DOOR#, STREET, and STATE. In the STUDENT table, these attributes can merge as an individual column.

- **Derived attributes are not considered in the table.**

In the STUDENT table, Age is the derived attribute. It can be calculated at any point of time by calculating the difference between current date and Date of Birth.

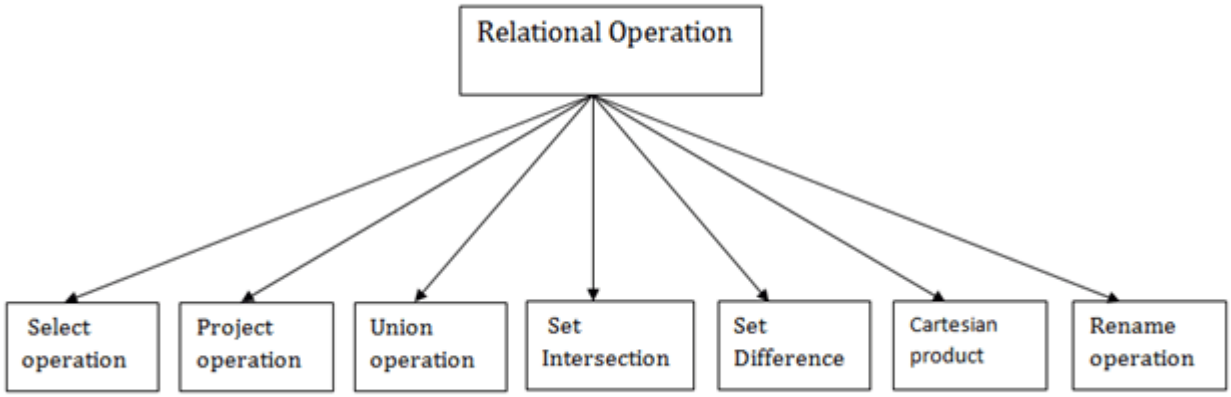
Using these rules, you can convert the ER diagram to tables and columns and assign the mapping between the tables. Table structure for the given ER diagram is as below:



Relational Algebra

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries.

Types of Relational operation



1. Select Operation:

- The select operation selects tuples that satisfy a given predicate.
- It is denoted by sigma (σ).

1. Notation: σ p(r)

Where:

σ is used for selection prediction

r is used for relation

p is used as a propositional logic formula which may use connectors like: AND OR and NOT. These relational can use as relational operators like =, ≠, ≥, <, >, ≤.

For example: LOAN Relation

BRANCH_NAME	LOAN_NO	AMOUNT
Downtown	L-17	1000
Redwood	L-23	2000
Perryride	L-15	1500
Downtown	L-14	1500
Mianus	L-13	500
Roundhill	L-11	900
Perryride	L-16	1300

Input:

1. σ BRANCH_NAME="perryride" (LOAN)

Output:

BRANCH_NAME	LOAN_NO	AMOUNT
Perryride	L-15	1500
Perryride	L-16	1300

2. Project Operation:

- This operation shows the list of those attributes that we wish to appear in the result. Rest of the attributes are eliminated from the table.
- It is denoted by ∏.

1. Notation: ∏ A1, A2, An (r)

Where

A1, **A2**, **A3** is used as an attribute name of relation **r**.

Example: CUSTOMER RELATION

NAME	STREET	CITY
Jones	Main	Harrison
Smith	North	Rye
Hays	Main	Harrison
Curry	North	Rye
Johnson	Alma	Brooklyn
Brooks	Senator	Brooklyn

Input:

1. ∏ NAME, CITY (CUSTOMER)

Output:

NAME	CITY
Jones	Harrison
Smith	Rye
Hays	Harrison
Curry	Rye
Johnson	Brooklyn
Brooks	Brooklyn

Difference between Selection and Projection in DBMS

S. No.	Category	Selection	Projection
1.	Other Names	The selection operation is also known as horizontal partitioning.	The Project operation is also known as vertical partitioning.

S. No.	Category	Selection	Projection
2.	Use	It is used to choose the subset of tuples from the relation that satisfies the given condition mentioned in the syntax of selection.	It is used to select certain required attributes, while discarding other attributes.
3.	Partitioning	It partitions the table horizontally.	It partitions the table vertically.
4.	Which used first	The selection operation is performed before projection (if they are to be used together).	The projection operation is performed after selection (if they are to be used together).
5.	Operator Used	Select operator is used in Selection Operation.	Project operator is used in Projection Operation.
6.	Operator Symbol	Select operator is denoted by Sigma symbol.	Project operator is denoted by Pi symbol.
7.	Commutative	Selection is commutative.	Projection is not commutative.
8.	Column	Select is used to select all	Project is used to select

S. No.	Category	Selection	Projection
	Selection	columns of a specific tuple.	specific columns.
9.	SQL Statements used	SELECT, FROM, WHERE	SELECT, FROM

SQL Set Operation

The SQL Set operation is used to combine the two or more SQL SELECT statements.

Types of Set Operation

1. Union
2. UnionAll
3. Intersect
4. Minus

1. Union

- The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
- In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.
- The union operation eliminates the duplicate rows from its resultset.

Syntax

SELECT column_name FROM table1

UNION
SELECT column_name FROM table2;

Example:

The First table

ID	NAME
1	Jack
2	Harry
3	Jackson

The Second table

ID	NAME
3	Jackson
4	Stephan
5	David

Union SQL query will be:

```
1. SELECT * FROM First
2. UNION
3. SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry

3	Jackson
4	Stephan
5	David

2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

Syntax:

```
1. SELECT column_name FROM table1
2. UNION ALL
3. SELECT column_name FROM table2;
```

Example: Using the above First and Second table.

Union All query will be like:

```
1. SELECT * FROM First
2. UNION ALL
3. SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson
3	Jackson
4	Stephan
5	David

3. Intersect

- It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
- In the Intersect operation, the number of datatype and columns must be the same.
- It has no duplicates and it arranges the data in ascending order by default.

Syntax

1. SELECT column_name FROM table1
2. INTERSECT

3. SELECT column_name FROM table2;

Example:

Using the above First and Second table.

Intersect query will be:

1. SELECT * FROM First
2. INTERSECT
3. SELECT * FROM Second;

The resultset table will look like:

ID	NAME
3	Jackson

4. Minus

- It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
- It has no duplicates and data arranged in ascending order by default.

Syntax:

1. SELECT column_name FROM table1

2. MINUS
3. SELECT column_name FROM table2;

Example

Using the above First and Second table.

Minus query will be:

1. SELECT * FROM First
2. MINUS
3. SELECT * FROM Second;

The resultset table will look like:

ID	NAME
1	Jack
2	Harry

RENAME (ρ) Operation in Relational Algebra

The RENAME operation is used to rename the output of a relation.

Sometimes it is simple and suitable to break a complicated sequence of operations and rename it as a relation with different names. Reasons to rename a relation can be many, like –

- We may want to save the result of a relational algebra expression as a relation so that we can use it later.
- We may want to join a relation with itself, in that case, it becomes too confusing to specify which one of the tables we are talking about, in that case, we rename one of the tables and perform join operations on them.

Notation:

ρ x **(R)**

where the symbol 'ρ' is used to denote the RENAME operator and R is the result of the sequence of operation or expression which is saved with the name X.

- **Example-1:** Query to rename the relation Student as Male Student and the attributes of Student – RollNo, SName as (Sno, Name).

Sno	Name
2600	Ronny
2655	Raja

ρ MaleStudent(Sno, Name) $\pi_{RollNo, SName}(\sigma_{Condition}$(Student))

SQL | DIVISION

Division is typically required when you want to find out entities that are interacting with **all entities** of a set of different type entities.

The division operator is used when we have to evaluate queries which contain the keyword ‘all’.

Some instances where division operator is used are:

- Which person has account in all the banks of a particular city?
- Which students have taken all the courses required to graduate?

In all these queries, the description after the keyword ‘all’ defines a set which contains some elements and the final result contains those units who satisfy these requirements.

Important: Division is not supported by SQL implementations. However, it can be represented using other operations.(like cross join, Except, In)

SQL Implementation of Division

Given two relations(tables): R(x,y) , S(y).

R and S : tables

x and y : column of R

y : column of S

R(x,y) div S(y) means gives all distinct values of x from R that are associated with all values of y in S.

Computation of Division : R(x,y) div S(y)

Steps:

- Find out all possible combinations of S(y) with R(x) by computing R(x) x(cross join) S(y), say r1
- Subtract actual R(x,y) from r1, say r2

- x in r2 are those that are not associated with every value in S(y); therefore R(x)-r2(x) gives us x
  that are associated with all values in S

Queries

1. Implementation 1:

```
SELECT * FROM R
WHERE x not in ( SELECT x FROM (
(SELECT x , y FROM (select y from S ) as p cross join
(select distinct x from R) as sp)
EXCEPT
(SELECT x , y FROM R) ) AS r );
```

9. Implementation 2 : Using correlated subquery

```
SELECT * FROM R as sx
WHERE NOT EXISTS (
(SELECT p.y FROM S as p )
EXCEPT
(SELECT sp.y FROM  R as sp WHERE sp.x = sx.x ) );
```

Relational algebra

Using steps which is mention above:

All possible combinations

r1 ← πx(R) x S

x values with “incomplete combinations”,

r2x ← πx(r1-R)

and

result ← πx(R)-r2x

div S = πx(R)- πx((πx(R) x S) – R)

ExamplesSupply Schema

sid (integer)	pid (integer)
101	1
102	1
101	3
103	2
102	2
102	3
102	4
102	5

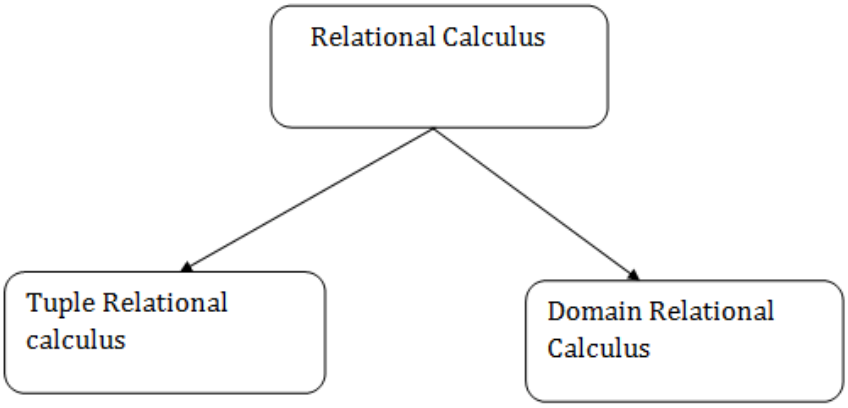
pid (integer)
1
2
3
4
5

Here **sid** means **supplierID** and **pid** means **partsID**.

Tables: suppliers(sid,pid) , parts(pid

Relational Calculus

- Relational calculus is a non-procedural query language. In the non-procedural query language, the user is concerned with the details of how to obtain the end results.
- The relational calculus tells what to do but never explains how to do.



1.Tuple Relational Calculus (TRC)

- The tuple relational calculus is specified to select the tuples in a relation. In TRC, filtering variable uses the tuples of a relation.
- The result of the relation can have one or more tuples.

Notation:

1. {T | P (T)} or {T | Condition (T)}

Where

T is the resulting tuples

P(T) is the condition used to fetch T.

For example:

1. { T.name | Author(T) AND T.article = 'database' }

OUTPUT: This query selects the tuples from the AUTHOR relation. It returns a tuple with 'name' from Author who has written an article on 'database'.

TRC (tuple relation calculus) can be quantified. In TRC, we can use Existential (∃) and Universal Quantifiers (∀).

For example:

1. { R| ∃T ∈ Authors(T.article='database' AND R.name=T.name)}

Output: This query will yield the same result as the previous one.

2. Domain Relational Calculus (DRC)

- The second form of relation is known as Domain relational calculus. In domain relational calculus, filtering variable uses the domain of attributes.
- Domain relational calculus uses the same operators as tuple calculus. It uses logical connectives ∧ (and), ∨ (or) and ¬ (not).
- It uses Existential (∃) and Universal Quantifiers (∀) to bind the variable.

Notation:

1. { a1, a2, a3, ..., an | P (a1, a2, a3, ... ,an)}

Where a1,a2 are attribute

P stands for formula build by inner attribute

MODULE -2

Relational Database Design

Data Anomalies

Normalization is necessary if you do not do it then the overall integrity of the data stored in the database will eventually degrade. Specifically, this is due to data anomalies. These anomalies naturally occur and result in data that does not match the real-world the database purports to represent.

Anomalies are caused when there is too much redundancy in the database's information. Anomalies can often be caused when the tables that make up the database suffer from poor construction. So, what does "poor construction" mean? Poor table design will become evident if, when the designer creates the database, he doesn't identify the entities that depend on each other for existence, like the rooms of a hotel and the hotel, and then minimize the chance that one would ever exist independent of the other.

The normalization process was created largely in order to reduce the negative effects of creating tables that will introduce anomalies into the database.

There are three types of **Data Anomalies**: Update Anomalies, Insertion Anomalies, and Deletion Anomalies.

Update Anomalies happen when the person charged with the task of keeping all the records current and accurate, is asked, for example, to change an employee’s

title due to a promotion. If the data is stored redundantly in the same table, and the person misses any of them, then there will be multiple titles associated with the employee. The end user has no way of knowing which is the correct title.

Insertion Anomalies happen when inserting vital data into the database is not possible because other data is not already there. For example, if a system is designed to require that a customer be on file before a sale can be made to that customer, but you cannot add a customer until they have bought something, then you have an insert anomaly. It is the classic "catch-22" situation.

Deletion Anomalies happen when the deletion of unwanted information causes desired information to be deleted as well. For example, if a single database record contains information about a particular product along with information about a salesperson for the company and the salesperson quits, then information about the product is deleted along with salesperson information

Functional Dependency

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

$X \rightarrow Y$

The left side of FD is known as a determinant, the right side of the production is known as a dependent.

For example:

Assume we have an employee table with attributes: Emp_Id, Emp_Name, Emp_Address.

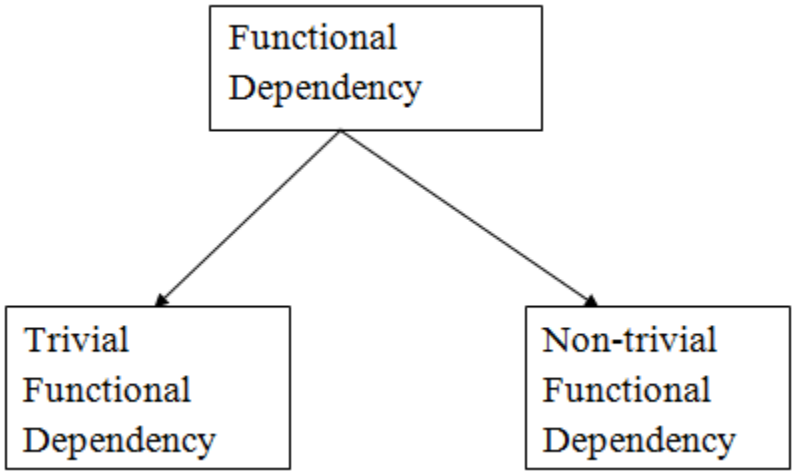
Here Emp_Id attribute can uniquely identify the Emp_Name attribute of employee table because if we know the Emp_Id, we can tell that employee name associated with it.

Functional dependency can be written as:

Emp_Id → Emp_Name

We can say that Emp_Name is functionally dependent on Emp_Id.

Types of Functional dependency



1. Trivial functional dependency

5. A → B has trivial functional dependency if B is a subset of A.
6. The following dependencies are also trivial like: A → A, B → B

Example:

Consider a table with two columns Employee_Id and Employee_Name.
{Employee_id, Employee_Name} → Employee_Id is a trivial functional dependency as
Employee_Id is a subset of {Employee_Id, Employee_Name}.
Also, Employee_Id → Employee_Id and Employee_Name → Employee_Name are trivial dependencies too.

2. Non-trivial functional dependency

2. A → B has a non-trivial functional dependency if B is not a subset of A.
3. When A intersection B is NULL, then A → B is called as complete non-trivial.

Example:

ID → Name
Name → DOB

Lossless Join and Dependency Preserving Decomposition

Decomposition of a relation is done when a relation in relational model is not in appropriate normal form. Relation R is decomposed into two or more relations if decomposition is lossless join as well as dependency preserving.

Lossless Join Decomposition

If we decompose a relation R into relations R1 and R2,

- Decomposition is lossy if $R1 \bowtie R2 \supset R$
- Decomposition is lossless if $R1 \bowtie R2 = R$

To check for lossless join decomposition using FD set, following conditions must hold:

- Union of Attributes of R1 and R2 must be equal to attribute of R. Each attribute of R must be either in R1 or in R2.
  $Att(R1) \cup Att(R2) = Att(R)$
- Intersection of Attributes of R1 and R2 must not be NULL.
  $Att(R1) \cap Att(R2) \neq \Phi$
- Common attribute must be a key for at least one relation (R1 or R2)

Att(R1) ∩ Att(R2) -> Att(R1) or Att(R1) ∩ Att(R2) -> Att(R2)

For Example, A relation R (A, B, C, D) with FD set{A->BC} is decomposed into R1(ABC) and R2(AD) which is a lossless join decomposition as:

1. First condition holds true as Att(R1) U Att(R2) = (ABC) U (AD) = (ABCD) = Att(R).
2. Second condition holds true as Att(R1) ∩ Att(R2) = (ABC) ∩ (AD) ≠ Φ
3. Third condition holds true as Att(R1) ∩ Att(R2) = A is a key of R1(ABC) because A->BC is given.

Dependency Preserving Decomposition

If we decompose a relation R into relations R1 and R2, All dependencies of R either must be a part of R1 or R2 or must be derivable from combination of FD's of R1 and R2.
For Example, A relation R (A, B, C, D) with FD set{A->BC} is decomposed into R1(ABC) and R2(AD) which is dependency preserving because FD A->BC is a part of R1(ABC).

GATE Question: Consider a schema R(A,B,C,D) and functional dependencies A->B and C->D. Then the decomposition of R into R1(AB) and R2(CD) is [GATE-CS-2001]

A. dependency preserving and lossless join
B. lossless join but not dependency preserving
C. dependency preserving but not lossless join
D. not dependency preserving and not lossless join
Answer: For lossless join decomposition, these three conditions must hold true:

- Att(R1) U Att(R2) = ABCD = Att(R)
- Att(R1) ∩ Att(R2) = Φ, which violates the condition of lossless join decomposition. Hence the decomposition is not lossless.

For dependency preserving decomposition,
A->B can be ensured in R1(AB) and C->D can be ensured in R2(CD). Hence it is dependency preserving decomposition.

Normalization

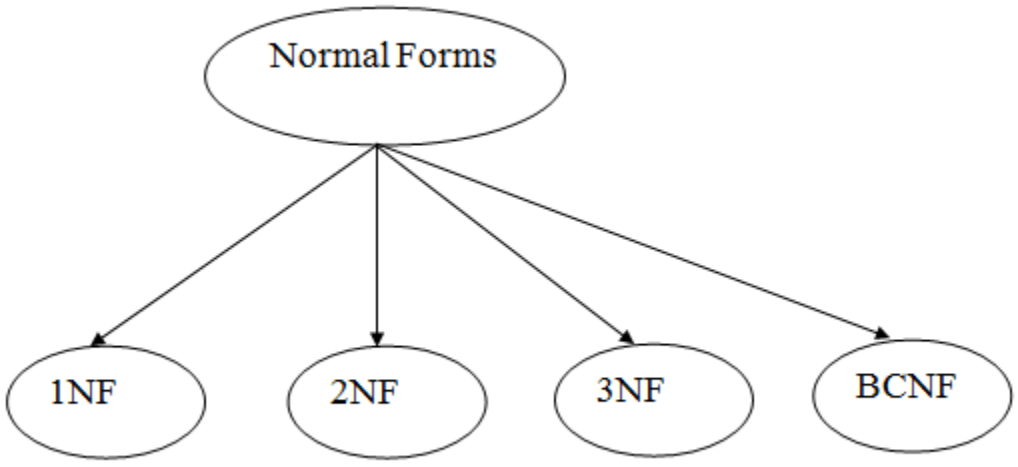
2. Normalization is the process of organizing the data in the database.

Normal Form	Description

3. Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
4. Normalization divides the larger table into the smaller table and links them using relationship.
5. The normal form is used to reduce redundancy from the database table.

Types of Normal Forms

There are the four types of normal forms:



1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
3NF	A relation will be in 3NF if it is in 2NF and no transition dependency exists.
4NF	A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
5NF	A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.

First Normal Form (1NF)

- A relation will be 1NF if it contains an atomic value.
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
- First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

Example: Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP_PHONE.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar

12	Sam	7390372389	Punjab

The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab

Second Normal Form (2NF)

2. In the 2NF, relational must be in 1NF.
3. In the second normal form, all non-key attributes are fully functional dependent on the primary key

Example: Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38

83	Computer	38

In the given table, non-prime attribute TEACHER_AGE is dependent on TEACHER_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

TEACHER_DETAIL table:

TEACHER_ID	TEACHER_AGE
25	30
47	35
83	38

TEACHER_SUBJECT table:

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology
47	English
83	Math
83	Computer

Third Normal Form (3NF)

1. A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
2. 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
3. If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

4. X is a super key.
5. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

Example:

EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CIT
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

Super key in the table above:

{EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP}....so on

Candidate key: {EMP_ID}

Non-prime attributes: In the given table, all attributes except EMP_ID are non-prime.

EMP_ZIP	EMP_STATE	EMP_CITY

Here, EMP_STATE & EMP_CITY dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID. The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent on super key(EMP_ID). It violates the rule of third normal form.

That's why we need to move the EMP_CITY and EMP_STATE to the new <EMPLOYEE_ZIP> table, with EMP_ZIP as a Primary key.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_ZIP
222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP table:

201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

Boyce Codd normal form (BCNF)

- BCNF is the advance version of 3NF. It is stricter than 3NF.
- A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

Example: Let's assume there is a company where employees work in more than one department.

EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

In the above table Functional dependencies are as follows:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate key: {EMP-ID, EMP-DEPT}

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300
D283	232
D283	549

Functional dependencies:

15. EMP_ID → EMP_COUNTRY
16. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Fourth normal form (4NF)

4. A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
5. For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

Example

STUDENT

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.In the STUDENT relation, a student with STU_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi-valued dependency on STU_ID, which leads to unnecessary repetition of data.

So to make the above table into 4NF, we can decompose it into two tables:

STUDENT_COURSE

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

STUDENT_HOBBY

STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing
74	Cricket
59	Hockey

Fifth normal form (5NF)

4. A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
5. 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
6. 5NF is also known as Project-join normal form (PJ/NF).

Example

SUBJECT	LECTURER	SEMESTER
Computer	Anshika	Semester 1
Computer	John	Semester 1
Math	John	Semester 1
Math	Akash	Semester 2
Chemistry	Praveen	Semester 1

In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields required to identify a valid data.

Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL. But all three columns together acts as a primary key, so we can't leave other two columns blank.

So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:

P1

SEMESTER	SUBJECT
Semester 1	Computer
Semester 1	Math
Semester 1	Chemistry
Semester 2	Math

P2

SUBJECT	LECTURER
Computer	Anshika
Computer	John
Math	John

Math	Akash
Chemistry	Praveen

P3

SEMSTER	LECTURER
Semester 1	Anshika
Semester 1	John
Semester 1	John
Semester 2	Akash
Semester 1	Praveen

Dependencies in DBMS is a relation between two or more attributes. It has the following types in DBMS −

2. Fully-Functional Dependency
3. Transitive Dependency
4. Multivalued Dependency
5. Partial Dependency
6. Join Dependency

Fully-functionally Dependency

An attribute is fully functional dependent on another attribute, if it is Functionally Dependent on that attribute and not on any of its proper subset.

For example, an attribute Q is fully functional dependent on another attribute P, if it is Functionally Dependent on P and not on any of the proper subset of P.

Let us see an example −

<ProjectCost>

ProjectID	**ProjectCost**
001	1000
002	5000

<EmployeeProject>

EmpID	**ProjectID**	**Days** (spent on the project)
E099	001	320
E056	002	190

The above relations states:

EmpID, ProjectID, ProjectCost -> Days

However, it is not fully functional dependent.

Whereas the subset **{EmpID, ProjectID}** can easily determine the **{Days}** spent on the project by the employee.

This summarizes and gives our fully functional dependency −

{EmpID, ProjectID} -> (Days)

Transitive Dependency

When an indirect relationship causes functional dependency it is called Transitive Dependency.

If P -> Q and Q -> R is true, then P-> R is a transitive dependency.

Multivalued Dependency

When existence of one or more rows in a table implies one or more other rows in the same table, then the Multi-valued dependencies occur.

If a table has attributes P, Q and R, then Q and R are multi-valued facts of P.

It is represented by double arrow −

->->

For our example:

P->->Q **Q->->R**

In the above case, Multivalued Dependency exists only if Q and R are independent attributes.

Partial Dependency

Partial Dependency occurs when a nonprime attribute is functionally dependent on part of a candidate key.

The 2nd Normal Form (2NF) eliminates the Partial Dependency. Let us see an example −

<StudentProject>

StudentID	ProjectNo	StudentName	ProjectName
S01	199	Katie	Geo Location
S02	120	Ollie	Cluster Exploration

In the above table, we have partial dependency; let us see how −

The prime key attributes are **StudentID** and **ProjectNo.**

As stated, the non-prime attributes i.e. **StudentName** and **ProjectName** should be functionally dependent on part of a candidate key, to be Partial Dependent.

The **StudentName** can be determined by **StudentID** that makes the relation Partial Dependent.

The **ProjectName** can be determined by **ProjectID**, which that the relation Partial Dependent.

Join Dependency

If a table can be recreated by joining multiple tables and each of this table have a subset of the attributes of the table, then the table is in Join Dependency. It is a generalization of Multivalued Dependency

Join Dependency can be related to 5NF, wherein a relation is in 5NF, only if it is already in 4NF and it cannot be decomposed further.

Example

<Employee>

EmpName	**EmpSkills**	**EmpJob** (Assigned Work)
Tom	Networking	EJ001
Harry	Web Development	EJ002
Katie	Programming	EJ002

The above table can be decomposed into the following three tables; therefore it is not in 5NF:

<EmployeeSkills>

EmpName	EmpSkills
Tom	Networking
Harry	Web Development
Katie	Programming

<EmployeeJob>

EmpName	EmpJob
Tom	EJ001
Harry	EJ002
Katie	EJ002

<JobSkills>

EmpSkills	EmpJob
Networking	EJ001
Web Development	EJ002
Programming	EJ002

Domain key Normal form

A relation is in DKNF when insertion or delete anomalies are not present in the database. Domain-Key Normal Form is the highest form of Normalization. The reason is that the insertion and updation anomalies are removed. The constraints are verified by the domain and key constraints.

A table is in Domain-Key normal form only if it is in 4NF, 3NF and other normal forms. It is based on constraints −

Domain Constraint

Values of an attribute had some set of values, for example, EmployeeID should be four digits long −

EmpID	**EmpName**	**EmpAge**
0921	Tom	33
0922	Jack	31

Key Constraint

An attribute or its combination is a candidate key

General Constraint

Predicate on the set of all relations.

Every constraint should be a logical sequence of the domain constraints and key constraints applied to the relation. The practical utility of DKNF is less.

MODULE 3

Structured Query Language (SQL)

Structured Query Language is a standard Database language which is used to create, maintain and retrieve the relational database. Following are some interesting facts about SQL.

- SQL is case insensitive. But it is a recommended practice to use keywords (like SELECT, UPDATE, CREATE, etc) in capital letters and use user defined things (liked table name, column name, etc) in small letters.
- We can write comments in SQL using "–" (double hyphen) at the beginning of any line.
- SQL is the programming language for relational databases (explained below) like MySQL, Oracle, Sybase, SQL Server, Postgre, etc. Other non-relational databases (also called NoSQL) databases like MongoDB, DynamoDB, etc do not use SQL
- Although there is an ISO standard for SQL, most of the implementations slightly vary in syntax. So we may encounter queries that work in SQL Server but do not work in MySQL..

Relational Database

Relational database means the data is stored as well as retrieved in the form of relations (tables). Table 1 shows the relational database with only one relation called **STUDENT** which

stores **ROLL_NO**, **NAME**, **ADDRESS**, **PHONE** and **AGE** of students.

STUDENT

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

TABLE 1

These are some important terminologies that are used in terms of relation.

Attribute: Attributes are the properties that define a relation. e.g.; **ROLL_NO**, **NAME** etc.

Tuple: Each row in the relation is known as tuple. The above relation contains 4 tuples, one of which is shown as:

1	RAM	DELHI	9455123451	18

Degree: The number of attributes in the relation is known as degree of the relation. The **STUDENT** relation defined above has degree 5.

Cardinality: The number of tuples in a relation is known as cardinality. The **STUDENT** relation defined above has cardinality 4.

Column: Column represents the set of values for a particular attribute. The column **ROLL_NO** is extracted from relation STUDENT.

The queries to deal with relational database can be categories as:

Data Definition Language: It is used to define the structure of the database. e.g; CREATE TABLE, ADD COLUMN, DROP COLUMN and so on.

Data Manipulation Language: It is used to manipulate data in the relations. e.g.; INSERT, DELETE, UPDATE and so on.

Data Query Language: It is used to extract the data from the relations. e.g.; SELECT

So first we will consider the Data Query Language. A generic query to retrieve from a relational database is:

1. **SELECT** [**DISTINCT**] Attribute_List **FROM** R1,R2….RM
2. [**WHERE** condition]
3. [**GROUP BY** (Attributes)[**HAVING** condition]]
4. [**ORDER BY**(Attributes)[**DESC**]];

Part of the query represented by statement 1 is compulsory if you want to retrieve from a relational database. The statements written inside [] are optional. We will look at the possible query combination on relation shown in Table 1.

Query By Example (QBE)

- Difficulty Level : Easy
- Last Updated : 01 Apr, 2020

If we talk about normal queries we fire on the database they should be correct and in a well-defined structure which means they should follow a proper syntax if the syntax or query is wrong definitely we will get an error and due to that our

application or calculation definitely going to stop. So to overcome this problem QBE was introduced. QBE stands for **Query By Example** and it was developed in 1970 by Moshe Zloof at IBM.

It is a graphical query language where we get a user interface and then we fill some required fields to get our proper result.

In SQL we will get an error if the query is not correct but in the case of QBE if the query is wrong either we get a wrong answer or the query will not be going to execute but we will never get any error.

Note-:

In QBE we don't write complete queries like SQL or other database languages it comes with some blank so we need to just fill that blanks and we will get our required result.

Example

Consider the example where a table 'SAC' present in the database with Name, Phone_Number and Branch fields. And we want to get the name of SAC-Representative name who belongs to the MCA Branch. If we write this query in SQL we have to write it like

```
SELECT NAME

FROM SAC

WHERE BRANCH = 'MCA'"
```

And definitely we will get our correct result. But in the case of QBE, it may be done as like there is a field present and we just need to fill it with "MCA" and then click on SEARCH button we will get our required result.

Points about QBE:

- Supported by most of the database programs.

- It is a Graphical Query Language.
- Created in parallel to SQL development.

Attention reader! Don’t stop learning now. Get hold of all the important CS Theory concepts for SDE interviews with the **CS Theory Course** at a student-friendly price and become industry ready.

SQL Data Types

Each column in a database table is required to have a name and a data type.

An SQL developer must decide what type of data that will be stored inside each column when creating a table. The data type is a guideline for SQL to understand what type of data is expected inside of each column, and it also identifies how SQL will interact with the stored data.

Note: Data types might have different names in different database. And even if the name is the same, the size and other details may be different! **Always check the documentation!**

MySQL Data Types (Version 8.0)

In MySQL there are three main data types: string, numeric, and date and time.

String data types:

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The *size* parameter specifies the column length in characters - can be from 0 to 255. Default is 1
VARCHAR(size)	A VARIABLE length string (can contain letters,

	numbers, and special characters). The *size* parameter specifies the maximum column length in characters - can be from 0 to 65535
BINARY(size)	Equal to CHAR(), but stores binary byte strings. The *size* parameter specifies the column length in bytes. Default is 1
VARBINARY(size)	Equal to VARCHAR(), but stores binary byte strings. The *size* parameter specifies the maximum column length in bytes.
TINYBLOB	For BLOBs (Binary Large OBjects). Max length: 255 bytes
TINYTEXT	Holds a string with a maximum length of 255 characters
TEXT(size)	Holds a string with a maximum length of 65,535 bytes
BLOB(size)	For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters
MEDIUMBLOB	For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data

LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters
LONGBLOB	For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data
ENUM(val1, val2, val3, ...)	A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them
SET(val1, val2, val3, ...)	A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list

Numeric data types:

Data type	**Description**
BIT(*size*)	A bit-value type. The number of bits per value is specified in *size*. The *size* parameter can hold a value from 1 to 64. The default value for *size* is 1.

TINYINT(*size*)	A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The *size* parameter specifies the maximum display width (which is 255)
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
SMALLINT(*size*)	A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The *size* parameter specifies the maximum display width (which is 255)
MEDIUMINT(*size*)	A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The *size* parameter specifies the maximum display width (which is 255)
INT(*size*)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The *size* parameter specifies the maximum display width (which is 255)
INTEGER(*size*)	Equal to INT(size)
BIGINT(*size*)	A large integer. Signed range is from -

	9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The *size* parameter specifies the maximum display width (which is 255)
FLOAT(*size*, *d*)	A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions
FLOAT(*p*)	A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE()
DOUBLE(*size*, *d*)	A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter
DOUBLE PRECISION(*size*, *d*)	
DECIMAL(*size*, *d*)	An exact fixed-point number. The total number

	of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0.
DEC(*size*, *d*)	Equal to DECIMAL(size,d)

Note: All the numeric data types may have an extra option: UNSIGNED or ZEROFILL. If you add the UNSIGNED option, MySQL disallows negative values for the column. If you add the ZEROFILL option, MySQL automatically also adds the UNSIGNED attribute to the column.

Date and Time data types:

Data type	**Description**
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
DATETIME(*fsp*)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time
TIMESTAMP(*fsp*)	A timestamp. TIMESTAMP values are stored as the

	number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition
TIME(*fsp*)	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format.

The SQL CREATE TABLE Statement

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

Syntax

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

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

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

Tip: For an overview of the available data types, go to our complete Data Types Reference.

SQL CREATE TABLE Example

The following example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

Example

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

Create Table Using Another Table

A copy of an existing table can also be created using CREATE TABLE.

The new table gets the same column definitions. All columns or specific columns can be selected.

If you create a new table using an existing table, the new table will be filled with the existing values from the old table.

Syntax

CREATE TABLE *new_table_name* AS
  SELECT *column1, column2,...*
  FROM *existing_table_name*
  WHERE ....;

The following SQL creates a new table called "TestTables" (which is a copy of the "Customers" table):

Example

CREATE TABLE TestTable AS
SELECT customername, contactname
FROM customers;

The SQL INSERT INTO Statement

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

INSERT INTO Syntax

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

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

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

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

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

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA
90	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Finland
91	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Poland

INSERT INTO Example

The following SQL statement inserts a new record in the "Customers" table:

Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen
```

21', 'Stavanger', '4006', 'Norway');the selection from the "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA
90	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Finland
91	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Poland
92	Cardinal	Tom B. Erichsen	Skagen 21	Stavanger	4006	Norway

The SQL UPDATE Statement

The UPDATE statement is used to modify the existing records in a table.

UPDATE Syntax

UPDATE *table_name*
SET *column1 = value1, column2 = value2, ...*
WHERE *condition*;

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK

5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

UPDATE Table

The following SQL statement updates the first customer (CustomerID = 1) with a new contact person *and* a new city.

Example

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CustomerID = 1;
```

The selection from the "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico

3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

The SQL DELETE Statement

The DELETE statement is used to delete existing records in a table.

DELETE Syntax

DELETE FROM *table_name* WHERE *condition*;

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany

2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-9	

SQL DELETE Example

The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

Example

```
DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';
```

The "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country

2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

DELETE FROM *table_name*;

The following SQL statement deletes all rows in the "Customers" table, without deleting the table:

Example

DELETE FROM Customers;

SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

CREATE VIEW Syntax

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

Note: A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

SQL CREATE VIEW Examples

The following SQL creates a view that shows all customers from Brazil:

Example

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
```

```
FROM Customers
WHERE Country = 'Brazil';
```

We can query the view above as follows:

Example

```
SELECT * FROM [Brazil Customers];
```

The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

Example

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName, Price
FROM Products
WHERE Price > (SELECT AVG(Price) FROM Products);
```

SQL ALTER TABLE Statement

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

The ALTER TABLE statement is also used to add and drop various constraints on an existing table

ALTER TABLE - ADD Column

To add a column in a table, use the following syntax:

ALTER TABLE *table_name*
ADD *column_name datatype*;

The following SQL adds an "Email" column to the "Customers" table:

Example

```
ALTER TABLE Customers
ADD Email varchar(255);
```

ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

ALTER TABLE *table_name*
DROP COLUMN *column_name*;

The following SQL deletes the "Email" column from the "Customers" table:

Example

```
ALTER TABLE Customers
DROP COLUMN Email;
```

ALTER TABLE - ALTER/MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

SQL Server / MS Access:

ALTER TABLE *table_name*
ALTER COLUMN *column_name datatype*;

My SQL / Oracle (prior version 10G):

ALTER TABLE *table_name*
MODIFY COLUMN *column_name datatype*;

Oracle 10G and later:

ALTER TABLE *table_name*
MODIFY *column_name datatype*;

SQL ALTER TABLE Example

Look at the "Persons" table:

ID	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Now we want to add a column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

```
ALTER TABLE Persons
ADD DateOfBirth date;
```

Notice that the new column, "DateOfBirth", is of type date and is going to hold a date. The data type specifies what type of data the column can hold. For a complete reference of all the data types available in MS Access, MySQL, and SQL Server, go to our complete Data Types reference.

The "Persons" table will now look like this:

ID	LastName	FirstName	Address	City	DateOfBirth
1	Hansen	Ola	Timoteivn 10	Sandnes	
2	Svendson	Tove	Borgvn 23	Sandnes	
3	Pettersen	Kari	Storgt 20	Stavanger	

The SQL DROP TABLE Statement

The DROP TABLE statement is used to drop an existing table in a database.

Syntax

DROP TABLE *table_name*;

SQL DROP TABLE Example

The following SQL statement drops the existing table "Shippers":

Example

```
DROP TABLE Shippers;
```

SQL TRUNCATE TABLE

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

Syntax

TRUNCATE TABLE *table_name*;

SQL Constraints

SQL constraints are used to specify rules for data in a table.

SQL Create Constraints

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

Syntax

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

SQL Constraints

SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Uniquely identifies a row/record in another table
- CHECK - Ensures that all values in a column satisfies a specific condition
- DEFAULT - Sets a default value for a column when no value is specified
- INDEX - Used to create and retrieve data from the database very quickly

SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a table.

Primary keys must contain UNIQUE values, and cannot contain NULL values.

A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

SQL PRIMARY KEY on CREATE TABLE

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

MySQL:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID)
);
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
    ID int NOT NULL PRIMARY KEY,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
```

```
    Age int
);
```

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

SQL PRIMARY KEY on ALTER TABLE

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
```

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CONSTRAINT PK_Person PRIMARY KEY (ID,LastName);
```

Note: If you use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).

DROP a PRIMARY KEY Constraint

To drop a PRIMARY KEY constraint, use the following SQL:

MySQL:

```
ALTER TABLE Persons
DROP PRIMARY KEY;
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT PK_Person;
```

SQL FOREIGN KEY Constraint

A FOREIGN KEY is a key used to link two tables together.

A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.

The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

Look at the following two tables:

"Persons" table:

PersonID	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

"Orders" table:

OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3

3	22456	2
4	24562	1

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

SQL FOREIGN KEY on CREATE TABLE

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

MySQL:

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Orders (
    OrderID int NOT NULL PRIMARY KEY,
    OrderNumber int NOT NULL,
    PersonID int FOREIGN KEY REFERENCES Persons(PersonID)
);
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
    REFERENCES Persons(PersonID)
);
```

SQL FOREIGN KEY on ALTER TABLE

To create a FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
ADD CONSTRAINT FK_PersonOrder
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint, use the following SQL:

MySQL:

```
ALTER TABLE Orders
DROP FOREIGN KEY FK_PersonOrder;
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
DROP CONSTRAINT FK_PersonOrder;
```

SQL UNIQUE Constraint

The UNIQUE constraint ensures that all values in a column are different.

Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint.

However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table

SQL UNIQUE Constraint on CREATE TABLE

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
    ID int NOT NULL UNIQUE,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

MySQL:

```
CREATE TABLE Persons (
    ID int NOT NULL,
```

```
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
   UNIQUE (ID)
);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT UC_Person UNIQUE (ID,LastName)
);
```

SQL UNIQUE Constraint on ALTER TABLE

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD UNIQUE (ID);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);
```

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

MySQL:

```
ALTER TABLE Persons
DROP INDEX UC_Person;
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT UC_Person;
```

SQL: ALTER command

alter command is used for altering the table structure, such as,

- to add a column to existing table
- to rename any existing column
- to change datatype of any column or to modify its size.

- to drop a column from the table.

ALTER Command: Add a new Column

Using ALTER command we can add a column to any existing table. Following is the syntax,

Here is an Example for this,

ALTER TABLE table_name ADD(

column_name datatype);

address VARCHAR(200

The above command will add a new column address to the table **student**, which will hold data of type varchar which is nothing but string, of length 200.

ALTER Command: Add multiple new Columns

Using ALTER command we can even add multiple new columns to any existing table. Following is the syntax,

ALTER TABLE table_name ADD(

column_name1 datatype1,

column-name2 datatype2,

column-name3 datatype3);

Here is an Example for this,

The above command will add three new columns to the **student** table

```
ALTER TABLE student ADD(
   father_name VARCHAR(60),
   mother_name VARCHAR(60),
   dob DATE);
```

ALTER Command: Add Column with default value

ALTER command can add a new column to an existing table with a default value too. The default value is used when no value is inserted in the column. Following is the syntax,

```
ALTER TABLE table_name ADD(
   column-name1 datatype1 DEFAULT some_value)
```

Here is an Example for this,

The above command will add a new column with a preset default value to the table **student**.

```
ALTER TABLE student ADD(
   dob DATE DEFAULT '01-Jan-99');
```

ALTER Command: Modify an existing Column

ALTER command can also be used to modify data type of any existing column. Following is the syntax,

Here is an Example for this,

```
ALTER TABLE table_name modify(
   column_name datatype
);
ALTER TABLE student MODIFY(
   address varchar(300));
```

Remember we added a new column address in the beginning? The above command will modify the address column of the **student** table, to now hold upto 300 characters.

ALTER Command: Rename a Column

Using ALTER command you can rename an existing column. Following is the syntax,

Here is an example for this,

```
ALTER TABLE table_name RENAME
   old_column_name TO new_column_name;
ALTER TABLE student RENAME
```

address TO location;

The above command will rename address column to location.

ALTER Command: Drop a Column

ALTER command can also be used to drop or remove columns.
Following is the syntax,

Here is an example for this,

ALTER TABLE table_name DROP(

column_name);

ALTER TABLE student DROP(

address);

The above command will drop the address column from the table **student**.

SELECT command

The SQL **SELECT** statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

Syntax

The basic syntax of the SELECT statement is as follows −

SELECT column1, column2, columnN FROM table_name;

Here, column1, column2... are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax.

```
SELECT * FROM table_name;
```

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 |
+----+----------+-----+-----------+----------+
```

The following code is an example, which would fetch the ID, Name and Salary fields of the customers available in CUSTOMERS table.

```
SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;
```

This would produce the following result −

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
```

1	Ramesh	2000.00
2	Khilan	1500.00
3	kaushik	2000.00
4	Chaitali	6500.00
5	Hardik	8500.00
6	Komal	4500.00
7	Muffy	10000.00
+----+----------+----------+

If you want to fetch all the fields of the CUSTOMERS table, then you should use the following query.

```
SQL> SELECT * FROM CUSTOMERS;
```

This would produce the result as shown below.

+----+----------+-----+-----------+----------+
| 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
+----+----------+-----+-----------+----------+

Logical operators

The Logical operators are those that are true or false. They return a true or false values to combine one or more true or false values.

The Logical operators are:

Operator	Description
AND	Logical AND compares between two Booleans as expression and returns true when both expressions are true...
OR	Logical OR compares between two Booleans as expression and returns true when one of the expression is true...
NOT	Not takes a single Boolean as an argument and changes its value from false to true or from true to false....

Special operators

Operator	Description	Operates on
IN	The IN operator checks a value within a set of values separated by commas and retrieve the rows from the table which are matching....	Any set of values of the same datatype
BETWEEN	The SQL BETWEEN operator tests an expression against a range. The range consists of a beginning, followed by an AND keyword and an end expression....	Numeric, characters, or datetime values
ANY	ANY compares a value to each value in a list or results from a query and	A value to a list or a single - columns set

	evaluates to true if the result of an inner query contains at least one row....	of values
ALL	ALL is used to select all records of a SELECT STATEMENT. It compares a value to every value in a list or results from a query. The ALL must be preceded by the comparison operators and evaluates to TRUE if the query returns no rows....	A value to a list or a single - columns set of values
SOME	SOME compare a value to each value in a list or results from a query and evaluate to true if the result of an inner query contains at least one row...	A value to a list or a single - columns set of values
EXISTS	The EXISTS checks the existence of a result of a subquery. The EXISTS subquery tests whether a subquery fetches at least one row. When no data is returned then this operator returns 'FALSE'...	Table

Syntax:

```
SELECT [column_name | * | expression] [logical operator]
[column_name | * | expression .....]
FROM <table_name>
WHERE <expressions> [ logical operator |
arithmetic operator | ...] <expressions>;
```

Parameters:

Name	Description
column_name	Name of the column of a table.

*	All the columns of a table.
expression	Expression made up of a single constant, variable, scalar function, or column name and can also be the pieces of a SQL query that compare values against other values or perform arithmetic calculations.
table_name	Name of the sable.
logical operator	AND, OR , NOT etc.
arithmetic operator	Plus(+), minus(-), multiply(*) and divide(/).

SQL Logical AND operator

Logical AND compares two Booleans as expression and returns TRUE when both of the conditions are TRUE and returns FALSE when either is FALSE; otherwise, returns UNKNOWN (an operator that has one or two NULL expressions returns UNKNOWN).

Example:

Sample table: customer

To get data of 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1. 'cust_country' must be 'UK',

2. and 'grade' of the 'customer' must be 2,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code, cust_name,

cust_city,cust_country,grade

FROM customer

WHERE cust_country = 'UK' AND grade = 2;
```

Copy

Output:

```
CUST_CODE CUST_NAME          CUST_CITY
CUST_COUNTRY         GRADE
--------- ------------------- ---------------------------------- ------------------- ----------
C00013    Holmes             London                        UK                2
C00024    Cook               London                       UK                2
```

SQL Logical OR operator

Logical OR compares two Booleans as expression and returns TRUE when either of the conditions is TRUE and returns FALSE when both are FALSE. otherwise, returns UNKNOWN (an operator that has one or two NULL expressions returns UNKNOWN).

Example:

To get data of 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' with following conditions -

1. either 'cust_country' is ’USA’,

2. or 'grade' of the 'customer' is 3,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code,cust_name,

cust_city,cust_country,grade

FROM customer

WHERE cust_country = 'USA' OR grade = 3;
```

Copy

Output:

```
CUST_CODE      CUST_NAME        CUST_CITY
CUST_COUNTRY           GRADE
-------------- ---------------- ---------------------------------- ------------------- ----------
C00001         Micheal           New York                          USA                  2
C00020         Albert           New York                          USA                  3
C00002         Bolt             New York                          USA                  3
C00010         Charles           Hampshair                         UK                   3
C00012         Steven           San Jose                          USA                  1
```

```
C00009      Ramesh      Mumbai                India          3
C00011      Sundariya   Chennai               India          3
```

SQL Logical NOT operator

Logical NOT takes a single Boolean as an argument and changes its value from false to true or from true to false.

Example:

To get all columns from the 'customer' table with following condition -

1. grade for the customer not greater than 1,

the following SQL statement can be used :

SQL Code:

```
SELECT * FROM customer
WHERE NOT grade>1;
```

Copy

Output:

```
CUST_CODE      CUST_NAME      CUST_CITY      WORKING_AREA
CUST_COUNTRY       GRADE
--------------- --------------- --------------- -------------- -------------------- ---------- ------
---
C00015         Stuart         London         London         UK                   1       .....
C00021         Jacks          Brisban        Brisban        Australia            1       ....
C00019         Yearannaidu    Chennai        Chennai        India                1       .....
```

C00005	Sasikant	Mumbai	Mumbai	India	1	.....
C00007	Ramanathan	Chennai	Chennai	India	1	.....
C00004	Winston	Brisban	Brisban	Australia	1	.....
C00023	Karl	London	London	UK	0	.....
C00006	Shilton	Torento	Torento	Canada	1	.....
C00012	Steven	San Jose	San Jose	USA	1	.....
C00008	Karolina	Torento	Torento	Canada	1	.....

SQL Logical multiple AND operator

In the following topics, we are discussing the usage of multiple AND operator.

In the following example, more than one 'AND' operators along with the SQL SELECT STATEMENT is used.

Example:

To get data of 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1.'cust_country' is ’UK',

2.and 'cust_city' is 'London',

3.and 'grade' of the 'customer' must be greater than 1,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code,cust_name,cust_city,cust_country,grade
```

```
FROM customer

WHERE cust_country='UK'

AND cust_city='London' AND  grade>1;
```

Copy

Output:

```
CUST_CODE      CUST_NAME      CUST_CITY      CUST_COUNTRY
GRADE
-------------- -------------- -------------- -------------------- ----------
C00013         Holmes         London         UK                            2
C00024         Cook           London         UK                           2
```

SQL Logical AND OR comparison operator

In the following topic, we are discussing the usage of 'AND' and 'OR' operator.

Using AND OR comparison operator with the select statement an example have shown.

Example:

To get data of 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1. 'cust_country' is ’UK’ or cust_city is 'London' ,

2. and 'grade' of the 'customer' must be other than 3 ,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code, cust_name,

cust_city, cust_country, grade

FROM customer

WHERE (cust_country = 'UK'

OR cust_city = 'London')

AND grade <> 3;
```

Copy

Output:

```
CUST_CODE      CUST_NAME      CUST_CITY      CUST_COUNTRY
GRADE
-------------- -------------- -------------- -------------------- ----------
C00013         Holmes         London         UK                            2
C00024         Cook           London         UK                            2
C00015         Stuart         London         UK                            1
C00023         Karl           London         UK                            0
```

SQL Logical NOT AND operator

In the following example, **NOT, AND** operator along with the SQL SELECT STATEMENT have used.

Example:

To get data of 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1. 'cust_country' must be 'India',

2. and 'grade' of the 'customer' must be other than 3,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code, cust_name,

cust_city, cust_country, grade

FROM customer

WHERE cust_country = 'India'

AND NOT grade = 3;
```

Copy

Output:

```
CUST_CODE      CUST_NAME      CUST_CITY      CUST_COUNTRY
GRADE
-------------- -------------- -------------- -------------------- ----------
C00025         Ravindran      Bangalore      India                     2
C00017         Srinivas       Bangalore      India                     2
C00014         Rangarappa     Bangalore      India                     2
C00016         Venkatpati     Bangalore      India                     2
C00019         Yearannaidu    Chennai        India                     1
```

C00007	Ramanathan	Chennai	India	1
C00005	Sasikant	Mumbai	India	1
C00022	Avinash	Mumbai	India	2

SQL Logical NOT AND comparison operator

In the following example, we are going to discuss the usage of NOT and AND comparison operator along with the SQL SELECT STATEMENT.

Example:

To get data of 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1.'cust_country' is other than 'India',

2.and 'grade' of the 'customer' must be 3,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code, cust_name,

cust_city, cust_country, grade

FROM customer

WHERE NOT cust_country = 'India' AND grade = 3;
```

Copy

Output:

```
CUST_CODE      CUST_NAME      CUST_CITY      CUST_COUNTRY
GRADE
-------------- -------------- -------------- -------------------- ----------
C00020         Albert         New York       USA                  3
C00002         Bolt           New York       USA                  3
C00010         Charles        Hampshair      UK                   3
```

SQL Logical multiple NOT operator

In the following topics, we are discussing the usage of multiple NOT operator.

In the following example, more than one NOT operators with the SQL SELECT STATEMENT have used.

Example:

To get data of 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1. 'cust_country' is other than 'India',

2. and 'cust_city' must be other than 'London',

3. and 'grade' of the 'customer' must be other than 1,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code, cust_name,

cust_city, cust_country, grade
```

```
FROM customer

WHERE NOT cust_country = 'India'

AND NOT cust_city = 'London' AND NOT grade = 1;
```

Copy

Output:

```
CUST_CODE      CUST_NAME      CUST_CITY      CUST_COUNTRY
GRADE
-------------- -------------- -------------- -------------------- ----------
C00001         Micheal        New York       USA                           2
C00020         Albert         New York       USA                           3
C00002         Bolt           New York       USA                           3
C00018         Fleming        Brisban        Australia                     2
C00010         Charles        Hampshair      UK                            3
C00003         Martin         Torento        Canada                        2
```

SQL Logical multiple NOT with equal to (=) operator

In the following topic, we are discussing the usage of multiple NOT operator with EQUAL TO operator.

In the following example, more than one Not operators and comparison operator equal to **(=)** with the SQL SELECT STATEMENT have used.

Example:

To get data of 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1. 'cust_country' is other than 'India',

2. and 'cust_city' must be other than 'London',

3. and 'grade' of the 'customer' must be 1 ,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code, cust_name,

cust_city, cust_country, grade

FROM customer

WHERE NOT cust_country = 'India'

AND NOT cust_city = 'London' AND grade = 1;
```

Copy

Output:

```
CUST_CODE      CUST_NAME      CUST_CITY      CUST_COUNTRY
GRADE
-------------- -------------- -------------- -------------------- ----------
C00021         Jacks          Brisban        Australia                 1
C00004         Winston        Brisban        Australia                 1
C00006         Shilton        Torento        Canada                    1
C00012         Steven         San Jose       USA                       1
C00008         Karolina       Torento        Canada                    1
```

SQL Logical multiple NOT with not equal to operator

In the following topic, we are discussing the usage of multiple NOT operator with NOT EQUAL TO operator.

In the following example, 'NOT' operator and comparison operator 'not equal to' (< >) along with the SQL SELECT STATEMENT have used.

Example:

To get data of 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1. 'cust_country' is other than 'India',S

2. and 'cust_city' must be other than 'London' ,

3. and 'grade' of the 'customer' must be not equal to other than 1,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code,cust_name,
cust_city,cust_country,grade
FROM customer
WHERE NOT cust_country='India'
AND NOT cust_city='London'
AND NOT grade<>1;
```

Copy

Output:

```
CUST_CODE      CUST_NAME      CUST_CITY      CUST_COUNTRY
GRADE
-------------- -------------- -------------- -------------------- ----------
C00021         Jacks          Brisban        Australia                 1
C00004         Winston         Brisban        Australia                 1
C00006         Shilton        Torento        Canada                    1
C00012         Steven         San Jose       USA                       1
C00008         Karolina        Torento        Canada                    1
```

SQL Logical NOT AND OR operator

In the example 'NOT' 'AND' 'OR' operator along with the SQL SELECT STATEMENT have used.

Example - 1:

To get data of 'cust_code', 'cust_name', ' cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1. 'cust_country' is not other than 'UK',

2. or 'cust_city' must be other than 'Bangalore' ,

3. and 'grade' of the 'customer' must be greater than 1,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code,

cust_name, cust_city, cust_country, grade

FROM customer

WHERE NOT (cust_country = 'UK' OR cust_city = 'Bangalore')

AND grade > 1;
```

Copy

Output:

```
CUST_CODE      CUST_NAME      CUST_CITY      CUST_COUNTRY
GRADE
-------------- -------------- -------------- -------------------- ----------
C00001         Micheal        New York       USA                           2
C00020         Albert         New York       USA                           3
C00002         Bolt           New York       USA                           3
C00018         Fleming        Brisban        Australia                     2
C00022         Avinash        Mumbai         India                         2
C00003         Martin         Torento        Canada                        2
C00009         Ramesh         Mumbai         India                         3
C00011         Sundariya      Chennai        India                         3
```

Example - 2:

To get data data of all 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1. grade is 1 or 'agent_code' is A003 will not come,

2. and 'cust_country' is 'India' will not come,

here is the SQL statement can be used :

SQL Code:

```
SELECT cust_code,cust_name,cust_city,cust_country,grade

FROM customer

WHERE NOT((grade=1 OR agent_code='A003')

AND cust_country='India');
```

Copy

Output:

```
CUST_CODE      CUST_NAME      CUST_CITY      CUST_COUNTRY
GRADE
-------------- -------------- -------------- -------------------- ----------
C00013         Holmes         London         UK                            2
C00001         Micheal        New York       USA                           2
C00020         Albert         New York       USA                           3
C00025         Ravindran      Bangalore      India                         2
C00024         Cook           London         UK                            2
C00015         Stuart         London         UK                            1
C00002         Bolt           New York       USA                           3
C00018         Fleming        Brisban        Australia                     2
C00021         Jacks          Brisban        Australia                     1
C00022         Avinash        Mumbai         India                         2
```

C00004	Winston	Brisban	Australia	1
C00023	Karl	London	UK	0
C00006	Shilton	Torento	Canada	1
C00010	Charles	Hampshair	UK	3
C00017	Srinivas	Bangalore	India	2
C00012	Steven	San Jose	USA	1
C00008	Karolina	Torento	Canada	1
C00003	Martin	Torento	Canada	2
C00009	Ramesh	Mumbai	India	3
C00014	Rangarappa	Bangalore	India	2
C00016	Venkatpati	Bangalore	India	2
C00011	Sundariya	Chennai	India	3

SQL Logical AND NOT OR with EQUAL TO (=) operator

In the following topic, we are discussing the usage of logical AND, NOT, OR and comparison operator EQUAL TO (=) in a select statement.

Example:

To get data of 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1. 'cust_country' is not other than 'UK' ,

2. or 'cust_city' must be not other than 'Bangalore' ,

3. and 'grade' of the 'customer' must be greater than 1 and other than 3,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code, cust_name,

cust_city, cust_country, grade

FROM customer

WHERE NOT (cust_country = 'UK' OR cust_city = 'Bangalore')

AND grade > 1 AND NOT grade = 3;
```

Copy

Output:

```
CUST_CODE      CUST_NAME      CUST_CITY      CUST_COUNTRY
GRADE
-------------- -------------- -------------- -------------------- ----------
C00001         Micheal        New York       USA                         2
C00018         Fleming        Brisban        Australia                   2
C00022         Avinash        Mumbai         India                       2
C00003         Martin         Torento        Canada                      2
```

SQL Logical AND NOT OR with LESS THAN, GREATER THAN operator

In the following topic, we are discussing the usage of logical AND NOT OR with LESS THAN (<) GREATER THAN (>) operator.

Example:

To get data of 'cust_code', 'cust_name', 'cust_city', 'cust_country' and 'grade' from the 'customer' table with following conditions -

1. 'cust_country' is not other than 'UK',

2. or 'cust_city' must be not other than 'Bangalore',

3. and 'grade' of the customer must be within the range 1 to 3,

the following SQL statement can be used :

SQL Code:

```
SELECT cust_code, cust_name, cust_city,

cust_country, grade

FROM customer

WHERE NOT (cust_country = 'UK' OR cust_city = 'Bangalore')

AND grade > 1 and grade < 3;
```

Copy

Output:

```
CUST_CODE  CUST_NAME       CUST_CITY        CUST_COUNTRY
GRADE
---------- ---------------- ----------------- ---------------- ------------
C00001     Micheal          New York          USA                2
C00018     Fleming          Brisban           Australia          2
C00022     Avinash          Mumbai            India              2
C00003     Martin           Torento           Canada             2
```

SQL Logical AND OR NOT with date value

In the following topic, we are discussing the usage of three operators 'AND', 'OR' and 'NOT' with date value.

Using AND , OR, NOT and comparison operator with the select statement an example have shown

Example - 1:

Sample table:orders

To get data of 'ord_num', 'ord_amount', 'advance_amount', 'ord_date', 'cust_code', 'agent_code' from the 'orders' table with following conditions -

1. combination of 'ord_date' is '20-Jul-08' and 'ord_num' is greater than 200120 will not appear,

2. or 'ord_amount' must be greater than or equal to 4000,

the following SQL statement can be used :

SQL Code:

```
SELECT ord_num, ord_amount, advance_amount,

ord_date, cust_code, agent_code

FROM orders

WHERE NOT ((ord_date = '20-Jul-08' AND ord_num > 200120)
```

```
OR ord_amount < 4000);
```

Copy

Output:

```
  ORD_NUM ORD_AMOUNT ADVANCE_AMOUNT ORD_DATE
CUST_CODE  AGENT_CODE
---------- ---------- -------------- --------- ---------- ----------
   200119       4000            700 16-SEP-08 C00007     A010
   200134       4200           1800 25-SEP-08 C00004     A005
   200108       4000            600 15-FEB-08 C00008     A004
   200107       4500            900 30-AUG-08 C00007     A010
   200113       4000            600 10-JUN-08 C00022     A002
```

Example - 2:

To get data of 'ord_num', 'ord_amount', 'advance_amount', 'ord_date', 'cust_code', 'agent_code' from the 'orders' table with following conditions -

1. 'ord_amount' will be below 1000,

2. combination of and 'ord_date' is '20-Jul-08' and ord_num is greater than 200108 will not appear

the following SQL statement can be used :

SQL Code:

```
SELECT ord_num,ord_amount,advance_amount,ord_date,

cust_code,agent_code
```

```
FROM orders

WHERE(ord_amount<1000 AND NOT(ord_date='20-Jul-08'

AND ord_num>200108));
```

Copy

Output:

```
ORD_NUM    ORD_AMOUNT ADVANCE_AMOUNT ORD_DATE
CUST_CODE  AGENT_CODE
---------- ---------- -------------- --------- ---------- ----------
   200117        800            200 20-OCT-08 C00014     A001
   200123        500            100 16-SEP-08 C00022     A002
   200116        500            100 13-JUL-08 C00010     A009
   200124        500            100 20-JUN-08 C00017     A007
   200126        500            100 24-JUN-08 C00022     A002
   200131        900            150 26-AUG-08 C00012     A012
```

Range Searching

In order to select data that is within a range of values, the BETWEEN operator is used. The BETWEEN operator allows the selection of rows that contain values within a specified lower and upper limit. The range coded after the word BETWEEN is inclusive.

The lower value must be coded first. The two values in between the range must be linked with the keyword AND. A BETWEEN operator can be used with both character and numeric data types. However, one cannot mix the data types that is the lower value of a range of values from a character column and the other from a numeric column.

Example:

1. Retrieve product_no,description,profit_percent,sell_price from the table product_master where the values contained within the field profit_percent is Between 10 and 20 both inclusive.

SELECT product_no,description,profit_percent,sell_price FROM Product_Master WHERE profit_percent BETWEEN 10 AND 20;

The above select will retrieve all the records from the product_master table where the profit_percent is in between 10 and 20 (both values inclusive).

Pattern Matching

MySQL provides standard SQL pattern matching as well as a form of pattern matching based on extended regular expressions similar to those used by Unix utilities such as **vi**, **grep**, and **sed**.

SQL pattern matching enables you to use _ to match any single character and `%` to match an arbitrary number of characters (including zero characters). In MySQL, SQL patterns are case-insensitive by default. Some examples are shown here. Do not use = or <> when you use SQL patterns. Use the `LIKE` or `NOT LIKE` comparison operators instead.

To find names beginning with `b`:

```
mysql> SELECT * FROM pet WHERE name LIKE 'b%';
+--------+--------+---------+------+------------+------------+
| name   | owner  | species | sex  | birth      | death      |
+--------+--------+---------+------+------------+------------+
| Buffy  | Harold | dog     | f    | 1989-05-13 | NULL       |
| Bowser | Diane  | dog     | m    | 1989-08-31 | 1995-07-29 |
+--------+--------+---------+------+------------+------------+
```

To find names ending with `fy`:

```
mysql> SELECT * FROM pet WHERE name LIKE '%fy';
+--------+--------+---------+------+------------+-------+
| name   | owner  | species | sex  | birth      | death |
+--------+--------+---------+------+------------+-------+
| Fluffy | Harold | cat     | f    | 1993-02-04 | NULL  |
| Buffy  | Harold | dog     | f    | 1989-05-13 | NULL  |
+--------+--------+---------+------+------------+-------+
```

To find names containing a `w`:

```
mysql> SELECT * FROM pet WHERE name LIKE '%w%';
+----------+-------+---------+------+------------+------------+
| name     | owner | species | sex  | birth      | death      |
+----------+-------+---------+------+------------+------------+
| Claws    | Gwen  | cat     | m    | 1994-03-17 | NULL       |
| Bowser   | Diane | dog     | m    | 1989-08-31 | 1995-07-29 |
| Whistler | Gwen  | bird    | NULL | 1997-12-09 | NULL       |
+----------+-------+---------+------+------------+------------+
```

To find names containing exactly five characters, use five instances of the _ pattern character:

```
mysql> SELECT * FROM pet WHERE name LIKE '_____';
+-------+--------+---------+------+------------+-------+
| name  | owner  | species | sex  | birth      | death |
+-------+--------+---------+------+------------+-------+
| Claws | Gwen   | cat     | m    | 1994-03-17 | NULL  |
| Buffy | Harold | dog     | f    | 1989-05-13 | NULL  |
+-------+--------+---------+------+------------+-------+
```

The other type of pattern matching provided by MySQL uses extended regular expressions. When you test for a match for this type of pattern, use the `REGEXP` and `NOT REGEXP` operators (or `RLIKE` and `NOT RLIKE`, which are synonyms).

The following list describes some characteristics of extended regular expressions:

- `.` matches any single character.
- A character class `[...]` matches any character within the brackets. For example, `[abc]` matches `a`, `b`, or `c`. To name a range of characters, use a dash. `[a-z]` matches any letter, whereas `[0-9]` matches any digit.
- `*` matches zero or more instances of the thing preceding it. For example, `x*` matches any number of `x` characters, `[0-9]*` matches any number of digits, and `.*` matches any number of anything.
- A regular expression pattern match succeeds if the pattern matches anywhere in the value being tested. (This differs from a `LIKE` pattern match, which succeeds only if the pattern matches the entire value.)
- To anchor a pattern so that it must match the beginning or end of the value being tested, use `^` at the beginning or `$` at the end of the pattern.

To demonstrate how extended regular expressions work, the `LIKE` queries shown previously are rewritten here to use `REGEXP`.

To find names beginning with `b`, use `^` to match the beginning of the name:

```
mysql> SELECT * FROM pet WHERE name REGEXP '^b';
+--------+--------+---------+------+------------+------------+
| name   | owner  | species | sex  | birth      | death      |
+--------+--------+---------+------+------------+------------+
| Buffy  | Harold | dog     | f    | 1989-05-13 | NULL       |
| Bowser | Diane  | dog     | m    | 1989-08-31 | 1995-07-29 |
```

```
+--------+--------+---------+------+------------+------------+
```

To force a `REGEXP` comparison to be case-sensitive, use the `BINARY` keyword to make one of the strings a binary string. This query matches only lowercase `b` at the beginning of a name:

```
SELECT * FROM pet WHERE name REGEXP BINARY '^b';
```

To find names ending with `fy`, use `$` to match the end of the name:

```
mysql> SELECT * FROM pet WHERE name REGEXP 'fy$';
+--------+--------+---------+------+------------+-------+
| name   | owner  | species | sex  | birth      | death |
+--------+--------+---------+------+------------+-------+
| Fluffy | Harold | cat     | f    | 1993-02-04 | NULL  |
| Buffy  | Harold | dog     | f    | 1989-05-13 | NULL  |
+--------+--------+---------+------+------------+-------+
```

To find names containing a `w`, use this query:

```
mysql> SELECT * FROM pet WHERE name REGEXP 'w';
+----------+-------+---------+------+------------+------------+
| name     | owner | species | sex  | birth      | death      |
+----------+-------+---------+------+------------+------------+
| Claws    | Gwen  | cat     | m    | 1994-03-17 | NULL       |
| Bowser   | Diane | dog     | m    | 1989-08-31 | 1995-07-29 |
| Whistler | Gwen  | bird    | NULL | 1997-12-09 | NULL       |
+----------+-------+---------+------+------------+------------+
```

Because a regular expression pattern matches if it occurs anywhere in the value, it is not necessary in the previous query to put a wildcard on either side of the pattern to get it to match the entire value as would be true with an SQL pattern.

To find names containing exactly five characters, use `^` and `$` to match the beginning and end of the name, and five instances of `.` in between:

```
mysql> SELECT * FROM pet WHERE name REGEXP '^.....$';
+-------+--------+---------+------+------------+-------+
| name  | owner  | species | sex  | birth      | death |
+-------+--------+---------+------+------------+-------+
| Claws | Gwen   | cat     | m    | 1994-03-17 | NULL  |
| Buffy | Harold | dog     | f    | 1989-05-13 | NULL  |
+-------+--------+---------+------+------------+-------+
```

You could also write the previous query using the `{n}` (“repeat-*n*-times”) operator:

```
mysql> SELECT * FROM pet WHERE name REGEXP '^.{5}$';
+-------+--------+---------+------+------------+-------+
| name  | owner  | species | sex  | birth      | death |
+-------+--------+---------+------+------------+-------+
| Claws | Gwen   | cat     | m    | 1994-03-17 | NULL  |
| Buffy | Harold | dog     | f    | 1989-05-13 | NULL  |
+-------+--------+---------+------+------------+-------+
```

GROUP BY

The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e if a particular column has same values in different rows then it will arrange these rows in a group.

Important Points:

- GROUP BY clause is used with the SELECT statement.
- In the query, GROUP BY clause is placed after the WHERE clause.

- In the query, GROUP BY clause is placed before ORDER BY clause if used any.

Syntax:

SELECT column1, function_name(column2)

FROM table_name

WHERE condition

GROUP BY column1, column2

ORDER BY column1, column2;

function_name: Name of the function used for example, SUM() , AVG().

table_name: Name of the table.

condition: Condition used.

Sample Table:

Employee

SI NO	NAME	SALARY	AGE
1	Harsh	2000	19
2	Dhanraj	3000	20
3	Ashish	1500	19
4	Harsh	3500	19
5	Ashish	1500	19

Student

SUBJECT	YEAR	NAME
English	1	Harsh
English	1	Pratik
English	1	Ramesh
English	2	Ashish
English	2	Suresh
Mathematics	1	Deepak
Mathematics	1	Sayan

Example:

- **Group By single column**: Group By single column means, to place all the rows with same value of only that particular column in one group. Consider the query as shown below:
- SELECT NAME, SUM(SALARY) FROM Employee
- GROUP BY NAME;

The above query will produce the below output:

NAME	SALARY
Ashish	3000
Dhanraj	3000
Harsh	5500

As you can see in the above output, the rows with duplicate NAMEs are grouped under same NAME and their corresponding SALARY is the sum of the SALARY of duplicate rows. The SUM() function of SQL is used here to calculate the sum.

- **Group By multiple columns**: Group by multiple column is say for example, **GROUP BY column1, column2**. This means to place all the rows with same values of both the columns **column1** and **column2** in one group. Consider the below query:
- SELECT SUBJECT, YEAR, Count(*)
- FROM Student

- GROUP BY SUBJECT, YEAR;

Output:

SUBJECT	YEAR	Count
English	1	3
English	2	2
Mathematics	1	2

As you can see in the above output the students with both same SUBJECT and YEAR are placed in same group. And those whose only SUBJECT is same but not YEAR belongs to different groups. So here we have grouped the table according to two columns or more than one column.

HAVING Clause

We know that WHERE clause is used to place conditions on columns but what if we want to place conditions on groups?

This is where HAVING clause comes into use. We can use HAVING clause to place conditions to decide which group will be the part of final result-set. Also we can not use the aggregate functions like SUM(), COUNT() etc. with WHERE clause. So we have to use HAVING clause if we want to use any of these functions in the conditions.

Syntax:

SELECT column1, function_name(column2)

FROM table_name

WHERE condition

GROUP BY column1, column2

HAVING condition

ORDER BY column1, column2;

function_name: Name of the function used for example, SUM() , AVG().

table_name: Name of the table.

condition: Condition used.

Example:

SELECT NAME, SUM(SALARY) FROM Employee

GROUP BY NAME

HAVING SUM(SALARY)>3000;

Output:

NAME	SUM(SALARY)
HARSH	5500

As you can see in the above output only one group out of the three groups appears in the result-set as it is the only group where sum of SALARY is greater than 3000. So we have used HAVING clause here to place this condition as the condition is required to be placed on groups not columns.

SQL JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Let's look at a selection from the "Orders" table:

OrderID	CustomerID	OrderDate

10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, look at a selection from the "Customers" table:

CustomerID	CustomerName	ContactName
1	Alfreds Futterkiste	Maria Anders
2	Ana Trujillo Emparedados y helados	Ana Trujillo
3	Antonio Moreno Taquería	Antonio Moreno

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

Example

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

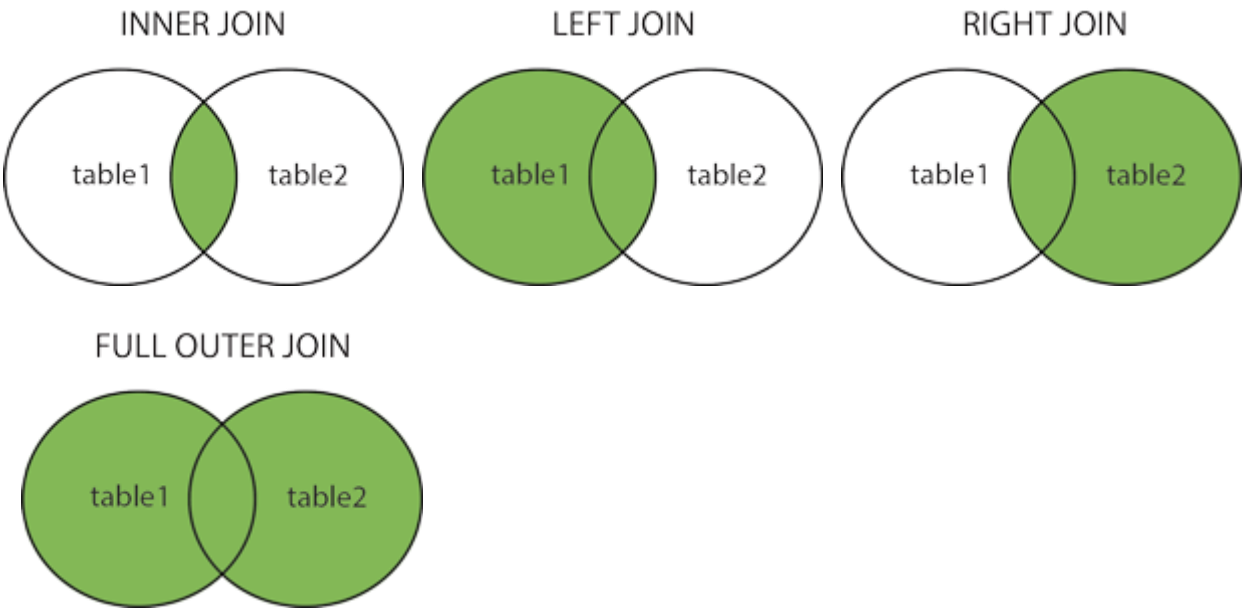
and it will produce something like this:

OrderID	CustomerName
10308	Ana Trujillo Emparedados y helados
10365	Antonio Moreno Taquería
10383	Around the Horn
10355	Around the Horn
10278	Berglunds snabbköp

Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table



DELETE

The SQL DELETE Query is used to delete the existing records from a table.

You can use the WHERE clause with a DELETE query to delete the selected rows, otherwise all the records would be deleted.

Syntax

The basic syntax of the DELETE query with the WHERE clause is as follows −

```
DELETE FROM table_name
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

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 |
+----+----------+-----+-----------+----------+
```

The following code has a query, which will DELETE a customer, whose ID is 6.

```
SQL> DELETE FROM CUSTOMERS
WHERE ID = 6;
```

Now, the CUSTOMERS table would have 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 |
```

```
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

If you want to DELETE all the records from the CUSTOMERS table, you do not need to use the WHERE clause and the DELETE query would be as follows −

```
SQL> DELETE FROM CUSTOMERS;
```

Now, the CUSTOMERS table would not have any record.

UPDATE

The SQL **UPDATE** Query is used to modify the existing records in a table. You can use the WHERE clause with the UPDATE query to update the selected rows, otherwise all the rows would be affected.

Syntax

The basic syntax of the UPDATE query with a WHERE clause is as follows −

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using the AND or the OR operators.

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 |
+----+----------+-----+-----------+----------+
```

The following query will update the ADDRESS for a customer whose ID number is 6 in the table.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6;
```

Now, the CUSTOMERS table would have 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 | Pune      |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

If you want to modify all the ADDRESS and the SALARY column values in the CUSTOMERS table, you do not need to use the WHERE clause as the UPDATE query would be enough as shown in the following code block.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune', SALARY = 1000.00;
```

Now, CUSTOMERS table would have the following records −

```
+----+----------+-----+---------+---------+
| ID | NAME     | AGE | ADDRESS | SALARY  |
+----+----------+-----+---------+---------+
|  1 | Ramesh   |  32 | Pune    | 1000.00 |
|  2 | Khilan   |  25 | Pune    | 1000.00 |
|  3 | kaushik  |  23 | Pune    | 1000.00 |
|  4 | Chaitali |  25 | Pune    | 1000.00 |
|  5 | Hardik   |  27 | Pune    | 1000.00 |
|  6 | Komal    |  22 | Pune    | 1000.00 |
|  7 | Muffy    |  24 | Pune    | 1000.00 |
```

SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

CREATE VIEW Syntax

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

Note: A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

SQL CREATE VIEW Examples

The following SQL creates a view that shows all customers from Brazil:

Example

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

Rename View

views and tables share the same namespace, you can use the **RENAME TABLE** statement to change the name of a view.

Here is the basic syntax of the **RENAME TABLE** for renaming a view:

```
RENAME TABLE original_view_name
TO new_view_name;
```

In this syntax:

1. First, specify the name of the view that you want to rename after the **RENAME TABLE** keywords.
2. Then, specify the new name of the view after the **TO** keyword.

> Note that you cannot use the **RENAME TABLE** statement to move a view from one database to another. Attempting to do so will result in an error.

3. **SQL DROP VIEW** DROP VIEW view_name

Why use the SQL DROP VIEW statement?

When a view no longer useful you may drop the view permanently. Also if a view needs change within it, it would be dropped and then created again with changes in appropriate places.

The following SQL drops the "Brazil Customers" view:

Example

```
DROP VIEW [Brazil Customers];
```

PROGRAM WITH SQL-

USING SET

Sometimes we want to keep declaration and initialization separate. SET can be used to assign values to the variable, post declaring a variable.Below are the different ways to assign values using SET:

Example: Assigning a value to a variable using SET

Syntax:

```
DECLARE @Local_Variable <Data_Type>
```

```
SET @Local_Variable =  <Value>
```

Query:

```
DECLARE @COURSE_ID AS INT
SET @COURSE_ID = 5
PRINT @COURSE_ID
```

.

USING SELECT

Just like SET, we can also use SELECT to assign values to the variables, post declaring a variable using DECLARE. Below are different ways to assign a value using SELECT:

Example: Assigning a value to a variable using SELECT

Syntax:

```
DECLARE @LOCAL_VARIABLE <Data_Type>
SELECT @LOCAL_VARIABLE = <Value>
```

Query:

```
DECLARE @COURSE_ID INT
SELECT @COURSE_ID = 5
PRINT @COURSE_ID
```

Example: Assigning a value to multiple variable using SELECT

Syntax:

```
DECLARE @Local_Variable _1 <Data_Type>, @Local_Variable _2 <Data_Type>,SELECT @Local_Variable _1 = <Value_1>,  @Local_Variable _2 = <Value_2>
```

Rules: Unlike SET, SELECT can be used to assign a value **to multiple variables** separated by the **comma**.

PROCEDURAL FLOW

At first glance, it appears that T-SQL is weak in procedural-flow options. Although it's less rich than some other languages, it suffices. The data-handling boolean extensions — such as EXISTS, IN, and CASE — offset the limitations of IF and WHILE.

Using If for Conditional T-SQL

This is your grandfather's IF. The T-SQL IF command determines the execution of *only* the next single statement — one IF, one command. In addition, there's no THEN and no END IF command to terminate the IF block:

```
IF Condition
Statement;
```

In the following script, the IF condition should return a false, preventing the next command from executing:

```
IF 1 = 0

PRINT 'Line One';
```

```
PRINT ‘Line Two';
```

Result:

```
Line Two
```

Note

The IF statement is not followed by a semicolon; in fact, a semicolon causes an error. That's because the IF statement is actually a prefix for the following statement; the two are compiled as a single statement.

Using Begin/End to Conditionally Execute Multiple Statements

An IF command that can control only a single command is less than useful. However, a BEGIN/END block can make multiple commands appear to the IF command as the next single command:

```
IF Condition
 Begin;
  Multiple lines;
 End;
```

Using If exists() as an Existence-Based Condition

While the IF command may seem limited, the condition clause can include several powerful SQL features similar to a WHERE clause, such as IF EXISTS() and IF. . .IN().

The IF EXISTS() structure uses the presence of any rows returned from a ...

USING WHILE LOOP

A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

```
WHILE condition LOOP
   sequence_of_statements
END LOOP;
```

Example

```
DECLARE
  a number(2) := 10;
BEGIN
  WHILE a < 20 LOOP
    dbms_output.put_line('value of a: ' || a);
    a := a + 1;
  END LOOP;
END;
/
```

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

PL/SQL procedure successfully completed.

USING GOTO Statement

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

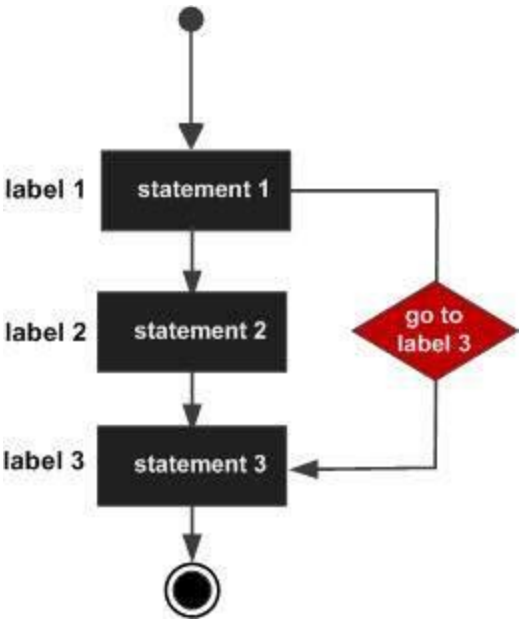
NOTE − The use of GOTO statement is not recommended in any programming language because it makes it difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

Syntax

The syntax for a GOTO statement in PL/SQL is as follows −

```
GOTO label;
..
..
<< label >>
statement;
```

Flow Diagram



Example

```
DECLARE
  a number(2) := 10;
BEGIN
  <<loopstart>>
  -- while loop execution
  WHILE a < 20 LOOP
  dbms_output.put_line ('value of a: ' || a);
    a := a + 1;
    IF a = 15 THEN
      a := a + 1;
      GOTO loopstart;
    END IF;
  END LOOP;
END;
/
```

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 16

value of a: 17

value of a: 18

value of a: 19

PL/SQL procedure successfully completed.

Restrictions with GOTO Statement

GOTO Statement in PL/SQL imposes the following restrictions −

- A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.
- A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.
- A GOTO statement cannot branch from an outer block into a sub-block (i.e., an inner BEGIN-END block).
- A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.

- A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.

Global Variables in MS SQL Server

Global variables are pre-defined system variables. It starts with **@@**. It provides information about the present user environment for SQL Server. SQL Server provides multiple global variables, which are very effective to use in Transact-SQL. The following are some frequently used global variables –

- @@SERVERNAME
- @@CONNECTIONS
- @@MAX_CONNECTIONS
- @@CPU_BUSY
- @@ERROR
- @@IDLE
- @@LANGUAGE
- @@TRANCOUNT
- @@VERSION

These are explained as following below.

1. **@@SERVERNAME :**
   This is used to find the name of the machine/computer on which SQL Server is running.
   Example –

```
Select @@servername
```

   Output –

```
SERVERXX\CTRXREST
```

2. **@@CONNECTIONS :**

   This is used to find number of logins or attempted logins since SQL Server was last started.

   Example –

```
Select @@connections
```

   Output –

```
59846824
```

3. **@@MAX_CONNECTIONS :**

   This is used to find the maximum number of simultaneous connections that can be made with SQL Server or instance in this computer environment.

   Example –

```
select @@max_connections
```

   Output –

```
32767
```

4. **@@CPU_BUSY :**

   This is used to find the amount of time, in microseconds, that the CPU has spent doing SQL Server work since the last time SQL Server was running.

   Example –

```
Select @@cpu_busy
```

   Output –

```
887468
```

5. **@@ERROR :**

   This is used to check the error status (succeeded or failed) of the most recently executed statement. It contains Zero (0) if the previous transaction succeeded, else, it contains the last error number generated by the system.

   Example –

```
Select @@error
```

   Output –

```
0
```

6. **@@IDLE :**

   The amount of time, in microseconds, that SQL Server has been idle since it was last started.

   Example –

```
Select @@idle
```

   Output –

```
123691249
```

7. **@@LANGUAGE :**

   This is used to find the name of the language that is currently used by the SQL Server.

   Example –

```
Select @@language
```

   Output –

```
us_english
```

8. **@@TRANCOUNT :**

   This is used to count the number of open transactions in the current session.

   Example –

```
Select @@trancount
```

   Output –

```
0
```

9. **@@VERSION :**

   This is used to find the current version of the SQL Server Software.

   Example –

```
Select @@version
```

   Output –

```
Microsoft SQL Server 2014 (SP3-CU-GDR) (KB4535288) - 12.0.6372.1 (X64)
Dec 12 2019 15:14:11
Copyright (c) Microsoft Corporation
Standard Edition (64-bit) on Windows NT 6.3 <X64> (Build 9600: ) (Hypervisor)
```

Security

Database security is the technique that protects and secures the database against intentional or accidental threats. Security concerns will be relevant not only to the data resides in an organization's database: the breaking of security may harm other parts of the system, which may ultimately affect the database structure. Consequently, database security includes hardware parts, software parts, human resources, and data. To efficiently do the uses of security needs appropriate controls, which are distinct in a specific mission and purpose for the system. The requirement for getting proper security while often having been neglected or overlooked in the past days; is now more and more thoroughly checked by the different organizations.

We consider database security about the following situations:

- Theft and fraudulent.
- Loss of confidentiality or secrecy.
- Loss of data privacy.
- Loss of data integrity.
- Loss of availability of data.

These listed circumstances mostly signify the areas in which the organization should focus on reducing the risk that is the chance of incurring loss or damage to data within a database. In some conditions, these areas are directly related such that an activity that leads to a loss in one area may also lead to a loss in another since all of the data within an organization are interconnected.

Locks:

Locks are mechanism used to ensure data integrity. The oracle engine automatically locks j table
data while executing SQL statements like Select/insert/UPDATE/DELETE. This K type of locking is
called implicit locking

There are two types of Locks

1. Shared lock
2. Exclusive lock

Shared lock:

Shared locks are placed on resources whenever a read operation (select) is performed.
Multiple shared locks can be simultaneously set on a resource.

Exclusive lock:

Exclusive locks are placed on resources whenever a write operation (INSERT, UPDATE And DELETE) are performed.
Only one exclusive lock can be placed on a resource at a time.
i.e. the first user who acquires an exclusive lock will continue to have the sole ownership of the resource, and no other user can acquire an exclusive lock on that resource

Levels of Locks:

Oracle does not provide a field level lock.
Oracle provides the following three levels of Locking.

- Row level
- Page level
- Table level

Row Level locking

If the WHERE clause evaluates to only one row in the table, a row level lock is used.

Page Level locking

If the WHERE clause evaluates to a set of data, a page level lock is used.

Table Level locking

If there is no WHERE clause, the query accesses the entire table, a table level lock is used.

Can't update entire table data when update is done by other user.

Syntax:

LOCK TABLE <tablename> [<tablename>]….. IN { ROW SHARE / ROW EXCLUSIVE / SHARE UPDATE / SHARE / SHARE ROW EXCLUSIVE / EXCLUSIVE}[NOWAIT]

Exclusive lock:

They allow query on the locked resource but prohibit any other activity

Deadlock:

In a deadlock, two database operations wait for each other to release a lock.

A deadlock occurs when two users have a lock, each on a separate object, and, they want to acquire a lock on each other's object.

When this happens, the first user has to wait for the second user to release the lock, but the second user will not release it until the lock on the first user's object is freed. At this point, both the users are at an impasse and cannot proceed with their business.

In such a case, Oracle detects the deadlock automatically and solves the problem by aborting one of the two transactions.

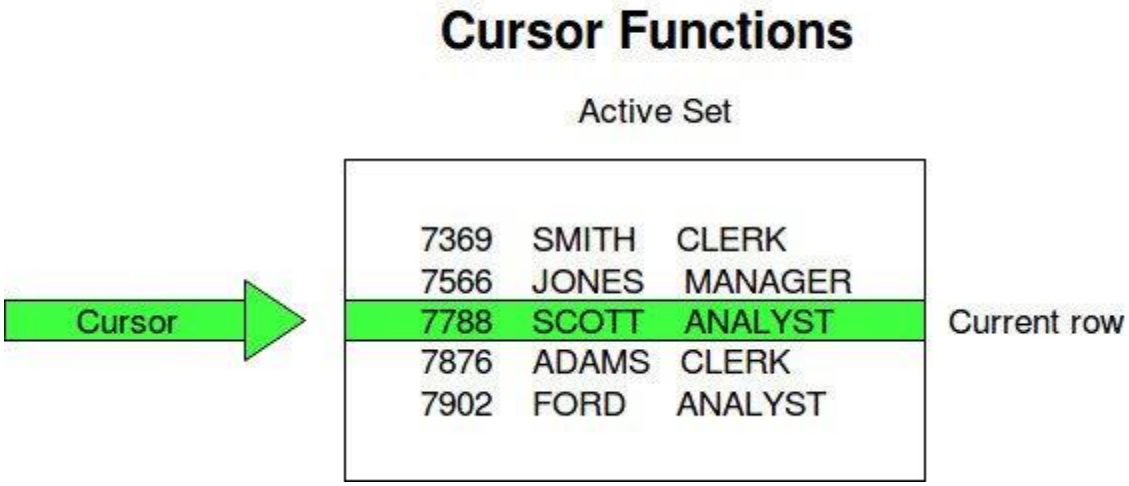
Cursor in SQL

To execute SQL statements, a work area is used by the Oracle engine for its internal processing and storing the information. This work area is private to SQL’s

operations. The ‘Cursor’ is the PL/SQL construct that allows the user to name the work area and access the stored information in it.

Use of Cursor

The major function of a cursor is to retrieve data, one row at a time, from a result set, unlike the SQL commands which operate on all the rows in the result set at one time Hence the size of the cursor is limited by the size of this pre-defined area.



Cursor Actions

- **Declare Cursor:** A cursor is declared by defining the SQL statement that returns a result set.
- **Open:** A Cursor is opened and populated by executing the SQL statement defined by the cursor.
- **Fetch:** When the cursor is opened, rows can be fetched from the cursor one by one or in a block to perform data manipulation.
- **Close:** After data manipulation, close the cursor explicitly.
- **Deallocate:** Finally, delete the cursor definition and release all the system resources associated with the cursor.

Types of Cursors

Cursors are classified depending on the circumstances in which they are opened.

- **Implicit Cursor:** If the Oracle engine opened a cursor for its internal processing it is known as an Implicit Cursor. It is created "automatically" for the user by Oracle when a query is executed and is simpler to code.
- **Explicit Cursor:** A Cursor can also be opened for processing data through a PL/SQL block, on demand. Such a user-defined cursor is known as an Explicit Cursor.

Work with cursors:

In computer science, a database **cursor** is a mechanism that enables traversal over the records in a database. Cursors facilitate subsequent processing in conjunction with the traversal, such as retrieval, addition and removal of database records. The database cursor characteristic of traversal makes cursors akin to the programming language concept of iterator.

Cursors are used by database programmers to process individual rows returned by database system queries. Cursors enable manipulation of whole result sets at once. In this scenario, a cursor enables the sequential processing of rows in a result set.

In SQL procedures, a cursor makes it possible to define a result set (a set of data rows) and perform complex logic on a row by row basis. By using the same mechanics, a SQL procedure can also define a result set and return it directly to the caller of the SQL procedure or to a client application.

A cursor can be viewed as a pointer to one row in a set of rows. The cursor can only reference one row at a time, but can move to other rows of the result set as needed.

Usage:

use cursors in SQL procedures, you need to do the following:

1. Declare a cursor that defines a result set.
2. Open the cursor to establish the result set.
3. Fetch the data into local variables as needed from the cursor, one row at a time.
4. Close the cursor when done.

To work with cursors you must use the following SQL statements

This section introduces the ways the SQL:2003 standard defines how to use cursors in applications in embedded SQL. Not all application bindings for relational database systems adhere to that standard, and some (such as CLI or JDBC) use a different interface.

A programmer makes a cursor known to the DBMS by using a `DECLARE ... CURSOR` statement and assigning the cursor a (compulsory) name:

```
DECLARE cursor_name CURSOR IS SELECT ... FROM ...
```

Before code can access the data, it must open the cursor with the `OPEN` statement. Directly following a successful opening, the cursor is positioned *before* the first row in the result set.

```
OPEN cursor_name
```

Applications position cursors on a specific row in the result set with the `FETCH` statement. A fetch operation transfers the data of the row into the application.

```
FETCH cursor_name INTO ...
```

Once an application has processed all available rows or the fetch operation is to be positioned on a non-existing row (compare scrollable cursors below), the DBMS returns a SQLSTATE '02000' (usually accompanied by an SQLCODE +100) to indicate the end of the result set.

The final step involves closing the cursor using the `CLOSE` statement:

```
CLOSE cursor_name
```

After closing a cursor, a program can open it again, which implies that the DBMS re-evaluates the same query or a different query and builds a new result set.

Scollar cursors:

Programmers may declare cursors as scrollable or not scrollable. The scrollability indicates the direction in which a cursor can move.

With a **non-scrollable** (or **forward-only**) cursor, you can `FETCH` each row at most once, and the cursor automatically moves to the next row. After you fetch the last row, if you fetch again, you will put the cursor after the last row and get the following code: SQLSTATE 02000 (SQLCODE +100).

A program may position a **scrollable** cursor anywhere in the result set using the `FETCH` SQL statement. The keyword SCROLL must be specified when declaring the cursor. The default is `NO SCROLL`, although different language bindings like JDBC may apply a different default.

```
DECLARE cursor_name sensitivity SCROLL CURSOR FOR SELECT ... FROM ...
```

The target position for a scrollable cursor can be specified relatively (from the current cursor position) or absolutely (from the beginning of the result set).

```
FETCH [ NEXT | PRIOR | FIRST | LAST ] FROM cursor_name
FETCH ABSOLUTE n FROM cursor_name
FETCH RELATIVE n FROM cursor_name;
```

Scrollable cursors can potentially access the same row in the result set multiple times. Thus, data modifications (insert, update, delete operations) from other transactions could affect the result set. A cursor can be SENSITIVE or INSENSITIVE to such data modifications. A sensitive cursor picks up data modifications affecting the result set of the cursor, and an insensitive cursor does not. Additionally, a cursor may be INSENSITIVE, in which case the DBMS tries to apply sensitivity as much as possible.

With hold :

Cursors are usually closed automatically at the end of a transaction, i.e. when a COMMIT or ROLLBACK (or an implicit termination of the transaction) occurs. That behavior can be changed if the cursor is declared using the WITH HOLD clause (the default is WITHOUT HOLD). A holdable cursor is kept open over

COMMIT and closed upon ROLLBACK. (Some DBMS deviate from this standard behavior and also keep holdable cursors open over ROLLBACK.)

```
DECLARE cursor_name CURSOR WITH HOLD FOR SELECT .... FROM ....
```

When a COMMIT occurs, a holdable cursor is positioned *before* the next row. Thus, a positioned UPDATE or positioned DELETE statement will only succeed after a FETCH operation occurred first in the transaction.

Error handling :

In this chapter, we will discuss Exceptions in PL/SQL. An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions −

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN*** −

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
```

```
  <exception handling goes here >
  WHEN exception1 THEN
    exception1-handling-statements
  WHEN exception2  THEN
    exception2-handling-statements
  WHEN exception3 THEN
    exception3-handling-statements
  ........
  WHEN others THEN
    exception3-handling-statements
END;
```

Example

Let us write a code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters −

```
DECLARE
  c_id customers.id%type := 8;
  c_name customerS.Name%type;
  c_addr customers.address%type;
BEGIN
  SELECT  name, address INTO  c_name, c_addr
  FROM customers
  WHERE id = c_id;
  DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
  DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
```

```
EXCEPTION
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

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

```
No such customer!
PL/SQL procedure successfully completed.
```

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND**, which is captured in the **EXCEPTION block**.

Raising Exceptions:

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception −

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  IF condition THEN
    RAISE exception_name;
```

```
  END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

You can use the above syntax in raising the Oracle standard exception or any user-defined exception. In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

User-defined Exceptions:

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure **DBMS_STANDARD.RAISE_APPLICATION_ERROR**.

The syntax for declaring an exception is −

```
DECLARE
  my-exception EXCEPTION;
```

Example

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception **invalid_id** is raised.

```
DECLARE
  c_id customers.id%type := &cc_id;
  c_name customerS.Name%type;
  c_addr customers.address%type;
```

```
  -- user defined exception
  ex_invalid_id  EXCEPTION;
BEGIN
  IF c_id <= 0 THEN
    RAISE ex_invalid_id;
  ELSE
    SELECT  name, address INTO  c_name, c_addr
    FROM customers
    WHERE id = c_id;
    DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
  END IF;

EXCEPTION
  WHEN ex_invalid_id THEN
    dbms_output.put_line('ID must be greater than zero!');
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

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

```
Enter value for cc_id: -6 (let's enter a value -6)
old  2: c_id customers.id%type := &cc_id;
```

new 2: c_id customers.id%type := -6;
ID must be greater than zero!

PL/SQL procedure successfully completed.

SQL Stored Procedures for SQL Server

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Stored Procedure Syntax

CREATE PROCEDURE *procedure_name*
AS
sql_statement
GO;

Execute a Stored Procedure

EXEC *procedure_name*;

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

---

Stored Procedure Example

The following SQL statement creates a stored procedure named "SelectAllCustomers" that selects all records from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers;
```

Stored procedures:

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Stored Procedure Syntax:

CREATE PROCEDURE *procedure_name*
AS
sql_statement
GO;

Execute a Stored Procedure:

EXEC *procedure_name*;

Stored Procedure Example:

The following SQL statement creates a stored procedure named "SelectAllCustomers" that selects all records from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers;
```

CREAT, ALTER, DROP:

Creat table:

The `CREATE TABLE` command creates a new table in the database.

The following SQL creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

Example

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

CREATE TABLE Using Another Table

A copy of an existing table can also be created using `CREATE TABLE`.

The following SQL creates a new table called "TestTables" (which is a copy of the "Customers" table):

Example

```
CREATE TABLE TestTable AS
SELECT customername, contactname
FROM customers;
```

Alter table:

The `ALTER TABLE` command adds, deletes, or modifies columns in a table.

The `ALTER TABLE` command also adds and deletes various constraints in a table.

The following SQL adds an "Email" column to the "Customers" table:

Example

```
ALTER TABLE Customers
ADD Email varchar(255);
```

The following SQL deletes the "Email" column from the "Customers" table:

Example

```
ALTER TABLE Customers
DROP COLUMN Email;
```

Drop table:

The `DROP TABLE` command deletes a table in the database.

The following SQL deletes the table "Shippers":

Example

```
DROP TABLE Shippers;
```

Passing and returning data to stored procedures:

There are several ways to capture the output from a stored procedure in SQL Server, and if you are working with an existing code set chances are you will not be able to only rely on one of the methods. From least to most flexibility, the 3 methods for passing data back from a stored procedure:

- the return code is a single integer value that can be set to infer meaning
- OUTPUT parameters can be used to pass one to many variables into a stored procedure to be updated with new values
- SELECT statements in a stored procedure to create results sets with one to many columns and one to many rows

Return Codes:

The return code is a light-code way to pass any whole-number (integer) back from a stored procedure. By adding RETURN before the value at the end of your procedure, the value can be captured with the procedure execution.

```
1   CREATE PROCEDURE RTNS_RETURNCODE
2       @APARAMATER NVARCHAR(10)
3   AS
4   BEGIN
5       UPDATE dbo.Badges
6       SET NAME = @APARAMATER
7       WHERE ID = 100
8        RETURN @@ROWCOUNT
9   END
10  GO
11  -- use of the return code
```

```
12  DECLARE @UPDATEDBADGES INT
13  EXEC @UPDATEDBADGES = RTNS_RETURNCODE
    N'BESTBADGEEVER'
    SELECT @UPDATEDBADGES
```

Output Parameters:

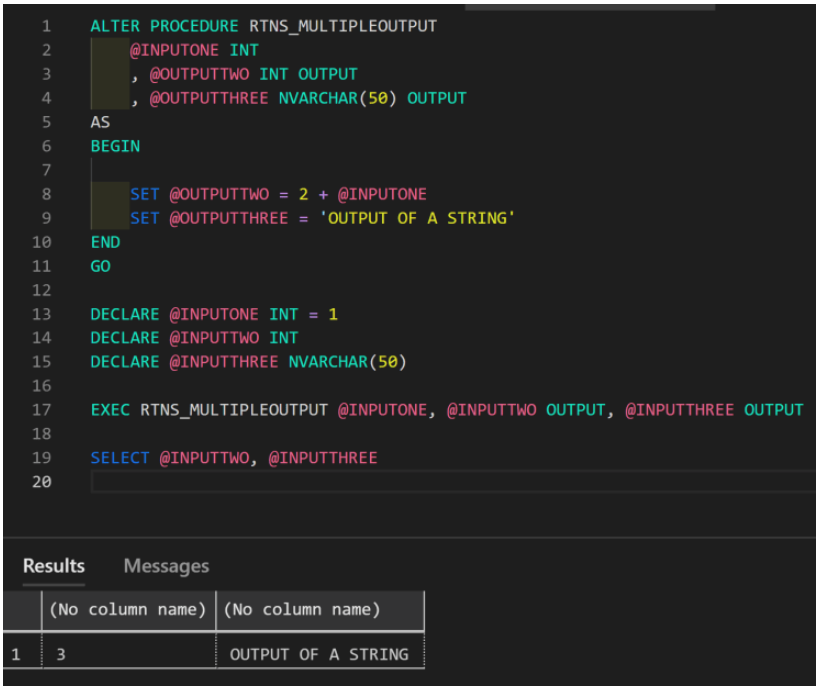
One step beyond the return code is the ability to pass in parameter(s) to a stored procedure that after execution contain the new value as set by the stored procedure. In creating the stored procedure, one or multiple parameters are modified with the OUTPUT keyword and updated within the stored procedure.

```
1  CREATE PROCEDURE RTNS_MULTIPLEOUTPUT
2     @INPUTONE INT
3     , @OUTPUTTWO INT OUTPUT
4     , @OUTPUTTHREE NVARCHAR(50) OUTPUT
5  AS
6  BEGIN
7   SET @OUTPUTTWO = 2 + @INPUTONE
8  SET @OUTPUTTHREE = 'OUTPUT OF A STRING'
9
```

END

When calling the stored procedure, the OUTPUT keyword needs to be specified with the variables where we are looking to receive an updated value base.

The values of the variables after the stored procedure is executed reflect the operations of the stored procedure.



1 DECLARE @INPUTONE INT = 1

2 DECLARE @INPUTTWO INT

3 DECLARE @INPUTTHREE NVARCHAR(50)

4 EXEC RTNS_MULTIPLEOUTPUT @INPUTONE, @INPUTTWO OUTPUT,

5 @INPUTTHREE OUTPUT

7 SELECT @INPUTTWO, @INPUTTHREE

Results Sets:

In concluding a stored procedure with a SELECT statement, you open yourself up to a whole subset of ways to capture data output from a stored procedure.

```
1  CREATE PROCEDURE RTNS_RESULTSSET
2      @INPUTONE INT
3  AS
4  BEGIN
5      SELECT ID, CreationDate, TEXT
6      FROM dbo.Comments
7      WHERE PostId = @INPUTONE
8  END
```

When executing the stored procedure as a part of an INSERT statement, you can capture the SELECT results from the stored procedure. This can be an INSERT into a table variable, temporary table, or non-temporal table. The example below creates a table variable, then uses it to store the output from RTNS_RESULTSSET.

```
1  DECLARE @INPUTONE INT = 47497
2  DECLARE @CAPTURETABLE TABLE(ID INT, CREATIONDATE DATETIME,
3  COMMENTTEXT NVARCHAR(700))
```

```
4  INSERT INTO @CAPTURETABLE (ID, CREATIONDATE, COMMENTTEXT)
5     EXEC RTNS_RESULTSSET @INPUTONE
6  SELECT * FROM @CAPTURETABLE
```

It is prudent to note that for use in applications, multiple results sets will be returned to the application by nested stored procedures with uncaptured SELECT statements.

Using stored procedures within queries:

Stored procedures are typically executed with an EXEC statement. However, you can execute a stored procedure implicitly from within a SELECT statement, provided that the stored procedure returns a result set. The OPENROWSET function is key to this technique, which involves three steps.

Step 1. Enable the Ad Hoc Distributed Queries Option

By default, SQL Server doesn't allow ad hoc distributed queries using OPENROWSET. Thus, the first step is to enable the Ad Hoc Distributed Queries configuration option on the SQL Server machine from which you'll be executing the query. If the option is disabled on your machine, you can use the EnableOption.sql script to enable it. You can download this script and the other code discussed here by clicking the *Download the Code Here* button near the top of the page. After running EnableOption.sql, you can confirm that the option is enabled by looking at the script's output. It should include the statement *Configuration option 'Ad Hoc Distributed Queries' changed from 0 to 1.*

Step 2. Create the View

The next step is to create a view that provides the same result as the stored procedure you want to execute in queries. You can then use that view in a SELECT statement. I created the sp_ConvProc2View stored procedure in Listing 1 to transform stored procedures into views.

Listing 1: sp_ConvProc2View.sql

```
-- BEGIN CALLOUT A CREATE PROCEDURE sp_ConvProc2View
(@procName varchar(80), @viewName varchar(80)) -- END CALLOUT A AS
BEGIN   DECLARE @TSQLStmt nvarchar(500)   SET NOCOUNT OFF --
BEGIN CALLOUT B   SET @TSQLStmt = N'CREATE VIEW ' + @viewName +
N' AS SELECT * FROM ' +     'OPENROWSET ( ' + "" + 'SQLOLEDB' + "" + ',' +
"" + 'SERVER=.;Trusted_Connection=yes' + "" + ',' +     "" + 'SET FMTONLY
OFF EXEC ' + @procName + "" + ')' -- END CALLOUT B   EXEC sp_executesql
@TSQLStmt   SET NOCOUNT ON END GO
```

As callout A in Listing 1 shows, sp_ConvProc2View requires two parameters: the name of the stored procedure you want to transform into a view (@procName) and the name you want to give that view (@viewName). It passes those parameters to the OPENROWSET function, which it executes with dynamic SQL. Within a string, dynamic SQL is slower than ordinary SQL and more prone to SQL injection attacks. Because of the latter, you need to make sure that any parameters you pass to sp_ConvProc2View (and hence OPENROWSET) are properly edited to minimize SQL injection attacks.

As callout B in Listing 1 shows, sp_ConvProc2View uses the SQLOLEDB provider along with a trusted connection (Windows authentication) to connect the default SQL Server instance (SERVER=.) and run OPENROWSET against it. If you want to use a named instance, you need to replace the period with the name of the instance or modify the code so that the instance's name is provided by a parameter to the procedure. The SET FMTONLY OFF statement at the end of callout B ensures that the results (and not just the metadata) will be output.

Let's look at a couple of examples of how to use sp_ConvProc2View. Note that the view specified with the @viewName parameter must not already exist. If it does, an error will occur.

Suppose you want to transform the sp_lock system stored procedure (which provides information about locks) into a view named v$Lock. In this case, you'd run

```
EXEC sp_ConvProc2View
  @procName = 'sp_Lock',
  @viewName ='v$Lock'
```

If you want to create a view named v$Session from the sp_who system stored procedure (which provides information about current users, sessions, and

```
EXEC sp_ConvProc2View
  @procName = 'sp_who',
  @viewName ='v$Session'
```

Step 3. Use the View in a SELECT Statement

After you create the view, you can use it in a SELECT statement. You can make the SELECT statement as simple or complex as needed. I'll show you three examples that use the v$Lock and v$Session views created in step 2. Note that the code in these examples is case sensitive.

Example 1: A simple SELECT statement. If you just want to see the locks in all the currently active sessions, you can run the statement

```
SELECT * FROM V$Lock
```

You can add a WHERE clause if you want to filter or group data. For example, if you want to see only those sessions running a background task, you can run the code

```
SELECT * FROM v$Session
  WHERE Status LIKE '%background%'
```

Example 2: A complex SELECT statement. To obtain more in-depth information, you can join views in a SELECT statement, thereby implicitly running multiple system stored procedures at the same time. For example, you can join the v$Lock, v$Session, and sys.objects views to obtain detailed information about the currently active sessions that contain locks with code such as

```
SELECT S.status,S.dbname,S.cmd,O.name,
  L.Type,L.Mode,L.Status
FROM v$Session S JOIN v$Lock L ON
```

```
  S.spid = L.spid JOIN sys.objects O
ON L.ObjId = O.object_id
```

For each locked process in a session, this query returns:

- The status of the involved process
- The name of the database used by that process
- The type of command executing the process
- The name of object that's locked
- The type of lock
- The lock mode requested
- The status of that lock request

Example 3: A SELECT statement whose results are loaded into a table. You can use code that creates and executes a SELECT statement, then loads the SELECT statement's results into an output table. I created the sp_OutputAndFilterResults stored procedure for this purpose. After creating and executing a SELECT statement, it loads the results into a temporary output table. This stored procedure takes three parameters, which are case sensitive:

- The name of the view (@ViewName).
- The name of the output table (@OutputTable). This name needs to be unique. If a table by that name already exists, it'll be dropped by the stored procedure.
- A filter condition (@WhereClause). Specifying a filter condition is optional. When one isn't specified, @WhereClause is assigned a NULL value.

Using this information, sp_OutputAndFilterResults creates a SELECT statement, assigning it to the @TSQL variable. Then, if there's a filter condition, the stored procedure adds it to the SELECT statement. After using PRINT to display the command in @TSQL for debug purposes, the stored procedure dynamically executes that command.

To execute sp_OutputAndFilterResults, you use code such as

```
EXEC sp_OutputAndFilterResults
  @ViewName ='v$Session',
  @OutputTable = 'output_sess' ,
  @WhereClause = 'DBname = "master"'
SELECT * FROM output_sess
```

This code returns all the currently active sessions that are using the master database and stores them in a temporary table named output_sess.

Building user defined functions:

Like programming languages SQL Server also provides User Defined Functions (UDFs). From SQL Server 2000 the UDF feature was added. UDF is a programming construct that accepts parameters, does actions and returns the result of that action. The result either is a scalar value or result set. UDFs can be used in scripts, Stored Procedures, triggers and other UDFs within a database.

Benefits

1. UDFs support modular programming. Once you create a UDF and store it in a database then you can call it any number of times. You can modify the UDF independent of the source code.
2. UDFs reduce the compilation cost of T-SQL code by caching plans and reusing them for repeated execution.
3. They can reduce network traffic. If you want to filter data based on some complex constraints then that can be expressed as a UDF. Then you can use this UDF in a WHERE clause to filter data.

Types

1. Scalar Functions
2. Table Valued Functions

Consider the following Student and Subject tables for examples.

Rno	Marks	Name
1	60	Ram
2	50	Ganesh
3	55	Sham
4	70	Raj

Rno	Subject1	Subject2	Subject3
1	10	30	20
2	10	10	30
3	20	25	10
4	15	20	35

1. Scalar Functions

A Scalar UDF accepts zero or more parameters and return a single value. The return type of a scalar function is any data type except text, ntext, image, cursor and timestamp. Scalar functions can be use in a WHERE clause of the SQL Query.

Crating Scalar Function

To create a scalar function the following syntax is used.

```
1. CREATE FUNCTION  function-name (Parameters)
2. RETURNS  return-type
3. AS
4. BEGIN
5.     Statement 1
6.     Statement 2
7.           .
8.           .
9.     Statement n
10.    RETURN return-value
11.END
```

Example

Create a function as follows.

```
1. CREATE FUNCTION GetStudent(@Rno INT)
2.  RETURNS VARCHAR(50)
3.  AS
4. BEGIN
5.     RETURN (SELECT Name FROM Student WHERE Rno=@Rno)
6. END
```

To execute this function use the following command.

```
1. PRINT dbo.GetStudent(1)
```

Output: Ram

1. Table Valued Functions

A Table Valued UDF accepts zero or more parameters and return a table variable. This type of function is special because it returns a table that you can query the results of a join with other tables. A Table Valued function is further categorized into an “Inline Table Valued Function” and a “Multi-Statement Table Valued Function”.

A. Inline Table Valued Function

An Inline Table Valued Function contains a single statement that must be a SELECT statement. The result of the query becomes the return value of the function. There is no need for a BEGIN-END block in an Inline function.

Crating Inline Table Valued Function

To create a scalar function the following syntax is used.

1. **CREATE FUNCTION function-name** (Parameters)
2. **RETURNS return**-type
3. **AS**
4. **RETURN**

Query

Example

```
1. CREATE FUNCTION GetAllStudents(@Mark INT)
2. RETURNS TABLE
3. AS
4. RETURN
5.     SELECT *FROM Student WHERE Marks>=@Mark
```

To execute this function use the following command.

```
1. SELECT *FROM GetAllStudents(60)
```

Output

Rno	Marks	Name
1	60	Ram
4	70	Raj

B. Multi-Statement Table Valued Function

A Multi-Statement contains multiple SQL statements enclosed in BEGIN-END blocks. In the function body you can read data from databases and do some operations. In a Multi-Statement Table valued function the return value is declared as a table variable and includes the full structure of the table to be returned. The RETURN statement is without a value and the declared table variable is returned.

Crating Multi-Statement Table Valued Function

To create a scalar function the following syntax is used.

```
1. CREATE FUNCTION  function-name (Parameters)
2. RETURNS @TableName TABLE
3. (Column_1 datatype,
4.      .
5.      .
6.   Column_n datatype
7. )
8. AS
9. BEGIN
10.Statement 1
11.         Statement 2
12.              .
13.              .
14.         Statement n
15.         RETURN
16.         END
```

Example

Crate a function as follows.

```
1. CREATE FUNCTION GetAvg(@Name varchar(50))
2. RETURNS @Marks TABLE
3. (Name VARCHAR(50),
```

```
4.  Subject1 INT,
5.      Subject2 INT,
6.  Subject3 INT,
7.  Average DECIMAL(4,2)
8. )
9. AS
10.BEGIN
11.     DECLARE @Avg DECIMAL(4,2)
12.     DECLARE @Rno INT

13.     INSERT INTO @Marks (Name)VALUES(@Name)

14.     SELECT @Rno=Rno FROM Student WHERE Name=@Name
15.SELECT @Avg=(Subject1+Subject2+Subject3)/3 FROM Subjects WHER
   E Rno=@Rno
16.
17.     UPDATE @Marks SET
18.Subject1=(SELECT Subject1 FROM Subjects WHERE Rno=@Rno),
19.     Subject2=(SELECT Subject2 FROM Subjects WHERE Rno=@Rn
   o),
20.     Subject3=(SELECT Subject3 FROM Subjects WHERE Rno=@Rn
   o),
21.     Average=@Avg
22.     WHERE Name=@Name
23.RETURN
24.END
```

To execute this function use the following command.

```
1. SELECT * FROM GetAvg('Ram')
```

Output

Name	Subject1	Subject2	Subject3	Average
Ram	10	30	20	20.00

Creating and calling a scalar function:

SQL Server scalar function takes one or more parameters and returns a single value.

The scalar functions help you simplify your code. For example, you may have a complex calculation that appears in many queries. Instead of including the formula in every query, you can create a scalar function that encapsulates the formula and uses it in each query.

Creating a scalar function:

To create a scalar function, you use the CREATE FUNCTION statement as follows:

```
CREATE FUNCTION [schema_name.]function_name (parameter_list)
RETURNS data_type AS
BEGIN
   statements
   RETURN value
```

```
END
```

Code language: SQL (Structured Query Language) (sql)

In this syntax:

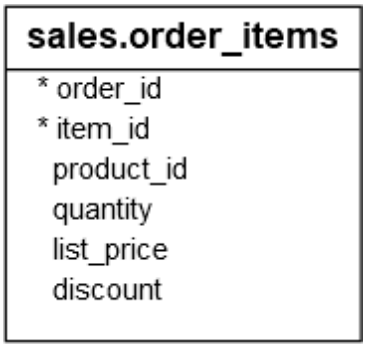
- First, specify the name of the function after the `CREATE FUNCTION` keywords. The schema name is optional. If you don't explicitly specify it, SQL Server uses `dbo` by default.
- Second, specify a list of parameters surrounded by parentheses after the function name.
- Third, specify the data type of the return value in the `RETURNS` statement.
- Finally, include a `RETURN` statement to return a value inside the body of the function.

The following example creates a function that calculates the net sales based on the quantity, list price, and discount:

```
CREATE FUNCTION sales.udfNetSale(
    @quantity INT,
    @list_price DEC(10,2),
    @discount DEC(4,2)
)
RETURNS DEC(10,2)
AS
BEGIN
    RETURN @quantity * @list_price * (1 - @discount);
END;
```

Code language: SQL (Structured Query Language) (sql)

Later on, we can use this to calculate net sales of any sales order in the order_items from the sample database.



Implementing triggers:

Trigger: A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

Syntax:

create trigger [trigger_name]

[before | after]

{insert | update | delete}

on [table_name]

[for each row]

[trigger_body]

Explanation of syntax:

1. create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
2. [before | after]: This specifies when the trigger will be executed.

{insert | update | delete}: This specifies the DML operation.

3. on [table_name]: This specifies the name of the table associated with the trigger.
4. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
5. [trigger_body]: This provides the operation to be performed as trigger is fired

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run.

AFTER triggers run the trigger action after the triggering statement is run.

Example:

Given Student Report Database, in which student marks assessment is recorded.

In such schema, create a trigger so that the total and average of specified marks is automatically inserted whenever a record is insert.

Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

Suppose the database Schema –

mysql> desc Student;

```
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| tid   | int(4)      | NO   | PRI | NULL    | auto_increment |
| name  | varchar(30) | YES  |     | NULL    |                |
| subj1 | int(2)      | YES  |     | NULL    |                |
| subj2 | int(2)      | YES  |     | NULL    |                |
```

| subj3 | int(2) | YES | | NULL | |

| total | int(3) | YES | | NULL | |

| per | int(3) | YES | | NULL | |

+-------+-------------+------+-----+---------+---------------+

7 rows in set (0.00 sec)

SQL Trigger to problem statement.

create trigger stud_marks

before INSERT

on

Student

for each row

set Student.total = Student.subj1 + Student.subj2 + Student.subj3, Student.per = Student.total * 60 / 100;

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,

mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);

Query OK, 1 row affected (0.09 sec)

mysql> select * from Student;

+-----+-------+-------+-------+-------+-------+------+

| tid | name | subj1 | subj2 | subj3 | total | per |

+-----+-------+-------+-------+-------+-------+------+

| 100 | ABCDE | 20 | 20 | 20 | 60 | 36 |

+-----+-------+-------+-------+-------+-------+------+

1 row in set (0.00 sec)

Multiple trigger interaction(using MySQL as the RBBMS):

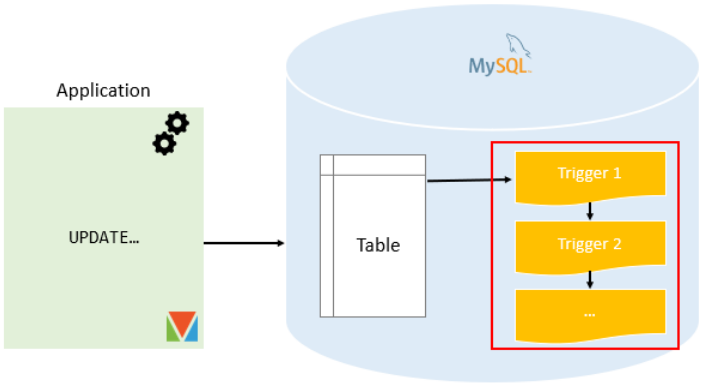
Before MySQL version 5.7.2, you can only create one trigger for an event in a table e.g., you can only create one trigger for the BEFORE UPDATE or AFTER UPDATE event. MySQL 5.7.2+ lifted this limitation and allowed you to create multiple triggers for a given table that have the same event and action time. These triggers will activate sequentially when an event occurs.
Here is the syntax for defining a trigger that will activate before or after an existing trigger in response to the same event and action time:

```
DELIMITER $$

CREATE TRIGGER trigger_name
{BEFORE|AFTER}{INSERT|UPDATE|DELETE}
ON table_name FOR EACH ROW
{FOLLOWS|PRECEDES} existing_trigger_name
BEGIN
   -- statements
END$$

DELIMITER ;
```

Code language: SQL (Structured Query Language) (sql)

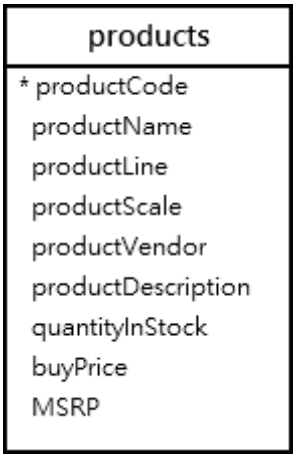
In this syntax, the FOLLOWS or PRECEDES specifies whether the new trigger should be invoked before or after an existing trigger.

- The FOLLOWS allows the new trigger to activate after an existing trigger.
- The PRECEDES allows the new trigger to activate before an existing trigger.



MySQL multiple triggers example

We will use the products table in the sample database for the demonstration.



Suppose that you want to change the price of a product (column MSRP) and log the old price in a separate table named PriceLogs .

First, create a new price_logs table using the following CREATE TABLE statement:

```
CREATE TABLE PriceLogs (
   id INT AUTO_INCREMENT,
   productCode VARCHAR(15) NOT NULL,
   price DECIMAL(10,2) NOT NULL,
   updated_at TIMESTAMP NOT NULL
                        DEFAULT CURRENT_TIMESTAMP
         ON UPDATE CURRENT_TIMESTAMP,
   PRIMARY KEY (id),
   FOREIGN KEY (productCode)
      REFERENCES products (productCode)
      ON DELETE CASCADE
      ON UPDATE CASCADE
);
```

Code language: SQL (Structured Query Language) (sql)

Second, create a new trigger that activates when the BEFORE UPDATE event of the products table occurs:

```
DELIMITER $$

CREATE TRIGGER before_products_update
  BEFORE UPDATE ON products
  FOR EACH ROW
BEGIN
    IF OLD.msrp <> NEW.msrp THEN
       INSERT INTO PriceLOgs(product_code,price)
       VALUES(old.productCode,old.msrp);
    END IF;
END$$
```

```
DELIMITER ;
```

Code language: SQL (Structured Query Language) (sql)

Third, check the price of the product S12_1099:

```
SELECT
   productCode,
   msrp
FROM
   products
WHERE
   productCode = 'S12_1099';
```

Code language: SQL (Structured Query Language) (sql)

	productCode	msrp
▶	S12_1099	194.57

Third, change the price of a product using the following UPDATE statement:

```
UPDATE products
SET msrp = 200
WHERE productCode = 'S12_1099';
```

Code language: SQL (Structured Query Language) (sql)

Fourth, query data from the PriceLogs table:

```
SELECT * FROM PriceLogs;
```

Code language: SQL (Structured Query Language) (sql)

	id	productCode	price	updated_at
▶	1	S12_1099	194.57	2019-09-08 09:07:02

It works as expected.

Suppose that you want to log the user who changed the price. To achieve this, you can add an additional column to the PriceLogs table.

However, for the purpose of multiple triggers demonstration, we will create a new separate table to store the data of users who made the changes.

Fifth, create the UserChangeLogs table:

```
CREATE TABLE UserChangeLogs (
   id INT AUTO_INCREMENT,
   productCode VARCHAR(15) DEFAULT NULL,
   updatedAt TIMESTAMP NOT NULL
         DEFAULT CURRENT_TIMESTAMP
      ON UPDATE CURRENT_TIMESTAMP,
   updatedBy VARCHAR(30) NOT NULL,
   PRIMARY KEY (id),
   FOREIGN KEY (productCode)
      REFERENCES products (productCode)
      ON DELETE CASCADE
      ON UPDATE CASCADE
);
```

Code language: SQL (Structured Query Language) (sql)

Sixth, create a BEFORE UPDATE trigger for the products table. This trigger activates after the before_products_update trigger.

```
DELIMITER $$

CREATE TRIGGER before_products_update_log_user
   BEFORE UPDATE ON products
   FOR EACH ROW
   FOLLOWS before_products_update
BEGIN
```

```
    IF OLD.msrp <> NEW.msrp THEN
        INSERT INTO
        UserChangeLogs(productCode,updatedBy)
    VALUES
        (OLD.productCode,USER());
    END IF;
END$$

DELIMITER ;
```

Code language: HTML, XML (xml)

Let's do a quick test.

Seventh, update the price of a product using the following UPDATE statement:

```
UPDATE
    products
SET
    msrp = 220
WHERE
    productCode = 'S12_1099';
```

Code language: SQL (Structured Query Language) (sql)

Eighth, query data from both PriceLogs and UserChangeLogs tables:

```
SELECT * FROM PriceLogs;
```

Code language: SQL (Structured Query Language) (sql)

id	productCode	price	updated_at
1	S12_1099	194.57	2019-09-08 09:07:02
2	S12_1099	200.00	2019-09-08 09:10:32

```
SELECT * FROM UserChangeLogs;
```

Code language: SQL (Structured Query Language) (sql)

	id	productCode	updatedAt	updatedBy
▶	1	S12_1099	2019-09-08 09:10:32	root@localhost

As you can see, both triggers were activated in the sequence as expected.

Information on trigger order

If you use the SHOW TRIGGERS statement to show the triggers, you will not see the order that triggers activate for the same event and action time.

```
SHOW TRIGGERS
FROM classicmodels
WHERE `table` = 'products';
```

Code language: SQL (Structured Query Language) (sql)

	Trigger	Event	Table	Statement	Timing
▶	before_products_update	UPDATE	products	BEGIN IF old.msrp <> new.msrp THEN INSERT ...	BEFORE
	before_products_update_log_user	UPDATE	products	BEGIN IF OLD.msrp <> NEW.msrp THEN INSER...	BEFORE

To find this information, you need to query the action_order column in the triggers table of the information_schema database as follows:

```
SELECT
    trigger_name,
    action_order
FROM
    information_schema.triggers
WHERE
    trigger_schema = 'classicmodels'
ORDER BY
    event_object_table ,
    action_timing ,
    event_manipulatio
```

MODULE -4

TRANSACTION

Transaction is an action or sequence of actions passed out by a single user and/or application program that reads or updates the contents of the database. A transaction is a logical piece of work of any database, which may be a complete program, a fraction of a program, or a single command (like the: SQL command INSERT or UPDATE) that may involve any number of processes on the database. From the database point of view, the implementation of an application program can be considered as one or more transactions with non-database processing working in between.Let's pick up an example of a simple transaction where a user transfers 620 from A's account into B's account. This transaction may seem small and straightforward but includes more than a few low-level tasks within it.

The first one is A's Account:

Open_Acct(A)

Old_Bal = A.bal

New_Bal = Old_Bal - 620

A.bal = New_Bal

Close_Acct(A)

The 2nd one is B's transaction:

Open_Acct(B)

Old_Bal = B.bal

New_Bal = Old_Bal + 620

B.bal = New_Bal

Close_Acct(B)

You can also see the DBMS Transactions page to get more information about DBMS transactions.It is to be noted that the transaction is very closely related to concurrency control.Transaction management is a logical unit of processing in a DBMS which entails one or more database access operation. It is a transaction is a program unit whose execution may or may not change the contents of a database. ... Active, Partially Committed, Committed, Failed & Terminate are important transaction states.A transaction can be defined as a group of tasks. single task is the minimum processing unit which cannot be divided further.

CONCURRENCY CONTROL

Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on adatabase.But before knowing about concurrency control, we should know about concurrent execution.

Concurrent Execution in DBMS

4. In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.

- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
- The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

  Problems with Concurrent Execution

  In a database transaction, the two main operations are READ and WRITE operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

  - Problem 1: Lost Update Problems (W - W Conflict)

  The problem occurs when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent

execution) that makes the values of the items incorrect hence making the database inconsistent.

2. Dirty Read Problems (W-R Conflict)

The dirty read problem occurs when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets
rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.

- Unrepeatable Read Problem (W-R Conflict)

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

Concurrency Control

Concurrency Control is the working concept that is required for controlling and managing the concurrent execution of database operations and thus avoiding the inconsistencies in the database. Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.

Concurrency Control Protocols

The concurrency control protocols ensure the atomicity, consistency, isolation, durability and serializability of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

- Lock Based Concurrency Control Protocol

4. Time Stamp Concurrency Control Protocol
4. Validation Based Concurrency Control Protocol

Lock-Based Protocol

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

1. Shared lock:

It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

2. Exclusive lock:

In the exclusive lock, the data item can be both reads as well as written by the transaction.This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

There are four types of lock protocols available:

1. Simplistic lock protocol

It is the simplest way of locking the data while transaction. Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it. It will unlock the data item after completing the transaction.

2. Pre-claiming Lock Protocol

- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
  - If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.

3. Two-phase locking (2PL)

- Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.
2. Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

4. Strict Two-phase locking (Strict-2PL)

  - The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
  - The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
  - Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock releas

Transaction in DBMS-

"Transaction is a set of operations which are all logically related."In a database, the transaction can be in one of the following states

Operations in Transaction-

The main operations in a transaction are-

- Read Operation
4. Write Operation

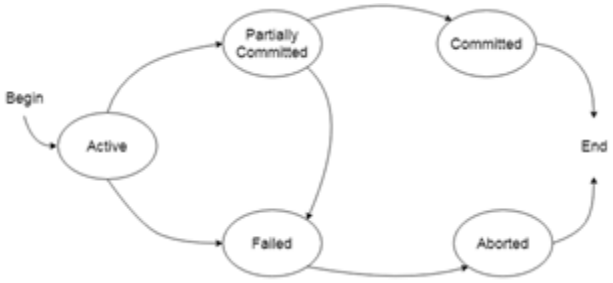
1. Read Operation-

Read operation reads the data from the database and then stores it in the buffer in main memory.For example- Read(A) instruction will read the value of A from the database and will store it in the buffer in main memory.

2. Write Operation-

Write operation writes the updated data value back to the database from the buffer.For example- Write(A) will write the updated value of A from the buffer to the database.

Transaction States-



Active state

6. The active state is the first state of every transaction. In this state, the transaction is being executed.

- For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

Partially committed

- In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.

7. In the total mark calculation example, a final display of the total marks step is executed in this state.

Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

Failed state

6. If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state

- .In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

Aborted

7. If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
4. If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.

- After aborting the transaction, the database recovery module will select one of the two operations:
  7. Re-start the transaction
  8. Kill the transaction

TRANSITION PROPERTIES

Any transaction must maintain the ACID properties, viz. Atomicity, Consistency, Isolation, and Durability.

- **Atomicity** − This property states that a transaction is an atomic unit of processing, that is, either it is performed in its entirety or not performed at all. No partial update should exist.

4. **Consistency** − A transaction should take the database from one consistent state to another consistent state. It should adversely affect any data item in the database.

6. **Isolation** − A transaction should be executed as if it is the only one in the system. There should not be any interference from the other concurrent transactions that are simultaneously running.

- **Durability** − If a committed transaction brings about a change, that change should be durable in the database and not lost in case of any failure.

SERIALIZABILITY

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. Serializability is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state.

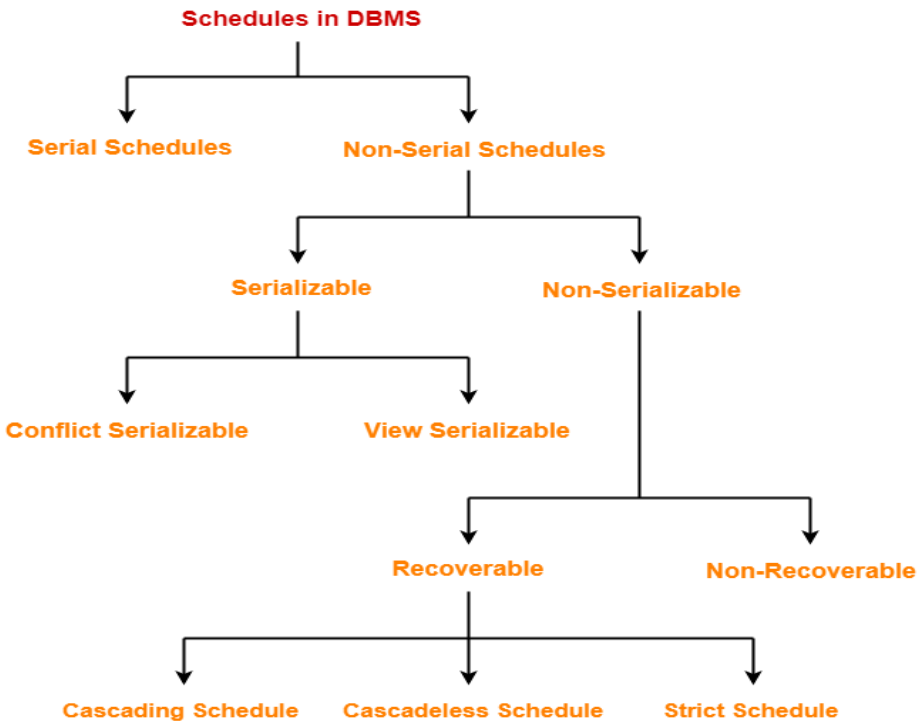
- A serializable schedule always leaves the database in consistent state. A serial schedule is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However a non-serial schedule needs to be checked for Serializability.

- A non-serial schedule of n number of transactions is said to be serializable schedule, if it is equivalent to the serial schedule of those n transactions. A serial schedule doesn't allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

SCHEDULES

The order in which the operations of multiple transactions appear for execution is called as a schedule.Serial Schedules-

Types of Schedules



serial schedules

All the transactions execute serially one after the other.When one transaction executes, no other transaction is allowed to executed

Serial schedules are always-

3. Consistent
- Recoverable

- Cascadeless
- Strict

non-serial schedules

- Multiple transactions execute concurrently.
- Operations of all the transactions are inter leaved or mixed with each other.

Non-serial schedules are NOT always-

- Consistent
- Recoverable
- Cascadeless
- Strict

Conflict Serializable Schedule

- A schedule is called conflict serializability if after swapping of non-conflicting operations, it can transform into a serial schedule.
- The schedule will be a conflict serializable if it is conflict equivalent to a serial schedule.

Conflicting Operations

- The two operations become conflicting if all conditions satisfy:
- Both belong to separate transactions.
- They have the same .

They contain at least one write operatio

View Serializability

A schedule will view serializable if it is view equivalent to a serial schedule.

If a schedule is conflict serializable, then it will be view serializable.

The view serializable which does not conflict serializable contains blind writes.

View Equivalent

Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

1. Initial Read
2. Updated Read

3. Final Write

Recoverability of Schedule

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

Cascading Schedule

If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is called as a Cascading Schedule or Cascading Rollback or Cascading Abort.

- It simply leads to the wastage of CPU time.

Cascadeless Schedule-

If in a schedule, a transaction is not allowed to read a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a Cascadeless Schedule.

- Cascadeless schedule allows only committed read operations.
- Therefore, it avoids cascading roll back and thus saves CPU time.

Strict Schedule-

If in a schedule, a transaction is neither allowed to read nor write a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a Strict Schedule

- Strict schedule allows only committed read and write operations.
- Clearly, strict schedule implements more restrictions than cascadeless schedule.

CONCURRENCY CONTROL

Concurrency Control in Database Management System is a procedure of managing simultaneous operations without conflicting with each other. It ensures that Database transactions are performed concurrently and accurately to produce correct results without violating data integrity of the respective Database.

Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical Database, it would have a mix of READ and WRITE operations and hence the concurrency is a challenge.
DBMS Concurrency Control is used to address such conflicts,
which mostly occur with a multi-user system. Therefore, Concurrency Control is the most important element for proper functioning of a Database Management System where two or more database transactions are executed simultaneously, which require access to the same data.
Concurrency Control Protocols
Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose. Following are the Concurrency Control techniques in DBMS:

- Lock-Based Protocols
- Two Phase Locking Protocol
- Timestamp-Based Protocols
- Validation-Based Protocols
- Lock-based Protocols

Lock Based Protocols in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.
A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks in DBMS help synchronize access to the database items by concurrent transactions.
All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Binary Locks: A Binary lock on a data item can either locked or unlocked states.
Shared/exclusive: This type of locking mechanism separates the locks in DBMS based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

1. Shared Lock (S):

A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.
For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

2. Exclusive Lock (X):
With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.
For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

3. Simplistic Lock Protocol
This type of lock-based protocols allows transactions to obtain a lock on every object before beginning operation. Transactions may unlock the data item after finishing the 'write' operation.
4. Pre-claiming Locking
Pre-claiming lock protocol helps to evaluate operations and create a list of required data items which are needed to initiate an execution process. In the situation when all locks are granted, the transaction executes. After that, all locks release when all of its operations are over.

Two Phase Locking Protocol

Two Phase Locking Protocol also known as 2PL protocol is a method of concurrency control in DBMS that ensures serializability by applying a lock to the transaction data which blocks other transactions to access the same data simultaneously. Two Phase Locking protocol helps to eliminate the concurrency problem in DBMS.
This locking protocol divides the execution phase of a transaction into three different parts.

- In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.

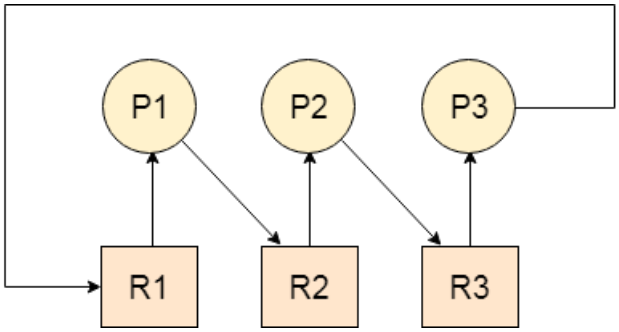
- The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.
- In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.

The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

- Growing Phase: In this phase transaction may obtain locks but may not release any locks.
- Shrinking Phase: In this phase, a transaction may release locks but not obtain any new lock

DEADLOCK

A deadlock is a condition where two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as no task ever gets finished and is in waiting state forever.



Necessary conditions for Deadlocks

- Mutual Exclusion
- A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.
- Hold and Wait

  A process waits for some resources while holding another resource at the same time.

- No preemption

  The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.

- Circular Wait

All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

Deadlock Avoidance

- When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restating the database. This is a waste of time and resource.
- Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

Deadlock Detection

In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait for the graph to detect the deadlock cycle in the database.

Deadlock Prevention

Deadlock prevention method is suitable for a large database. If the resources are allocated in such a way that deadlock never occurs, then the deadlock can be prevented.

The Database management system analyzes the operations of the transaction whether they can create a deadlock situation or not. If they do, then the DBMS never allowed that transaction to be executed.

GRANULARITY

It deals with the cost of implementing locks depending upon the space and time. Here, space refers to data structure in DBMS for each lock and time refers to handling of lock request and release.

The cost of implementing locks depends on the size of data items. There are two types of lock granularity:

• Fine granularity

• Coarse granularity

Fine granularity refers for small item sizes and coarse granularity refers for large item Sizes.

Here, Sizes decides on the basis:

- a database record
- a field value of a database record
- a disk block
- a whole file
- the whole database

If a typical transaction accesses a small number of records it is advantageous that the data item granularity is one record. If a transaction typically accesses many records of the same file it is better to have block or file granularity so that the transaction will consider all those records as one data item.

Timestamp Ordering Protocol

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.

BASICS OF QUERY PROCESSING

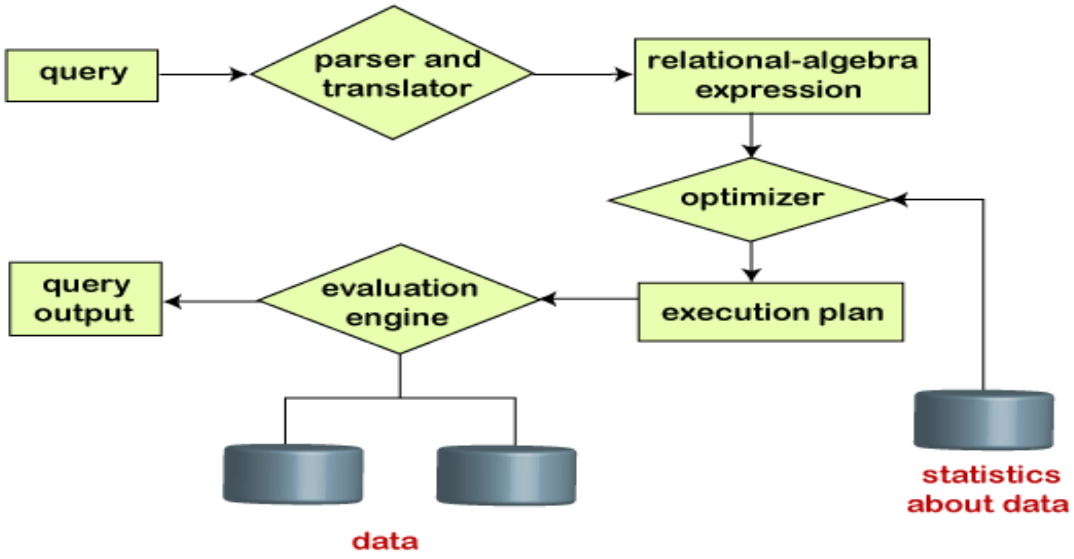
Query Processing is the activity performed in extracting data from the database. In processing, it takes various steps for fetching the data from the database. The steps involved are:

- Parsing and translation
- Optimization
- Evaluation

The query processing works in the following way:

Parsing and TranslationAs query processing includes certain activities for data retrieval. Initially, the given user queries get translated in high-level database languages such as SQL. It gets translated into expressions that can be further used at the physical level of the file system. After this, the actual evaluation of the queries and a variety of query -optimizing transformations and takes place. Thus before processing a query, a computer system needs to translate the query into a human-readable and understandable language. Consequently, SQL or

Structured Query Language is the best suitable choice for humans. But, it is not perfectly suitable for the internal representation of the query to the system. Relational algebra is well suited for the internal representation of a query. The translation process in query processing is similar to the parser of a query. When a user executes any query, for generating the internal form of the query, the parser in the system checks the syntax of the query, verifies the name of the relation in the database, the tuple, and finally the required attribute value. The parser creates a tree of the query, known as 'parse-tree.' Further, translate it into the form of relational algebra. With this, it evenly replaces all the use of the views when used in th



Steps in query processing

MODULE 5

Object Oriented Database Management Systems (OODBMS)

The **ODBMS** which is an abbreviation for **object oriented database management system**, is the data model in which data is stored in form of objects, which are instances of classes. These classes and objects together makes an object oriented data model.

Components of Object Oriented Data Model:

The OODBMS is based on three major components, namely: Object structure, Object classes, and Object identity. These are explained as following below.

1. **Object Structure:**

   The structure of an object refers to the properties that an object is made up of. These properties of an object are referred to as an attribute. Thus, an object is a real world entity with certain attributes that makes up the object structure. Also an object encapsulates the data code into a single unit which in turn provides data abstraction by hiding the implementation details from the user.

   The object structure is further composed of three types of components:

   Messages, Methods, and Variables. These are explained as following below.

   1. **Messages –**

      A message provides an interface or acts as a communication medium between an object and the outside world. A message can be of two types:

      - **Read-only message:** If the invoked method does not change the value of a variable, then the invoking message is said to be a read-only message.

- **Update message:** If the invoked method changes the value of a variable, then the invoking message is said to be an update message.

2. **Methods –**
   When a message is passed then the body of code that is executed is known as a method. Every time when a method is executed, it returns a value as output. A method can be of two types:
   - **Read-only method:** When the value of a variable is not affected by a method, then it is known as read-only method.
   - **Update-method:** When the value of a variable changes by a method, then it is known as an update method.

3. **Variables –**
   It stores the data of an object. The data stored in the variables makes the object distinguishable from one another.

2. Object Classes:

An object which is a real world entity is an instance of a class. Hence first we need to define a class and then the objects are made which differ in the values they store but share the same class definition. The objects in turn corresponds to various messages and variables stored in it.

Example –

```
class CLERK


 { //variables
```

```
    char name;
    string address;
    int id;
    int salary;

   //methods
    char get_name();
    string get_address();
    int annual_salary();
  };
```

In above example we can see, CLERK is a class that holds the object variables and messages.

An OODBMS also supports inheritance in an extensive manner as in a database there may be many classes with similar methods, variables and messages. Thus, the concept of class hierarchy is maintained to depict the similarities among various classes.

The concept of encapsulation that is the data or information hiding is also supported by object oriented data model. And this data model also provides the facility of abstract data types apart from the built-in data types like char, int, float. ADT's are the user defined data types that hold the values within it and can also have methods attached to it.

Thus, OODBMS provides numerous facilities to it's users, both built-in and user defined. It incorporates the properties of an object oriented data model with a database management system, and supports the concept of programming paradigms like classes and objects along with the support for other concepts like encapsulation, inheritance and the user defined ADT's (abstract data types).

Composite objects in OODBMS

Many applications in such domains as computer-aided design require the capability to define, store and retrieve as a single unit a collection of related objects known as a composite object. A composite object explicitly captures and enforces the IS-PART-OF integrity constraint between child and parent pairs of objects in a hierarchical collection of objects. Further, it can be used as a unit of storage and retrieval to enhance the performance of a database system

Advantages and Disadvantages of OODBMS

OODBMSs can provide appropriate solutions for many types of advanced database applications. However, there are also disadvantages.

Advantages

Enriched modeling capabilities

The object-oriented data model allows the ‘real world’ to be modeled more closely. The object, which encapsulates both state and behavior, is a more natural and realistic representation of real-world objects. An object can store all the relationships it has with other objects, including many-to-many relationships, and objects can be formed into complex objects that the traditional data models cannot cope with easily.

Extensibility

OODBMSs allow new data types to be built from existing types. The ability to factor out common properties of several classes and form them into a super-class that can be shared with sub-classes can greatly reduce redundancy within system is regarded as one of the main advantages of object orientation. Further, the reusability of classes promotes faster development and easier maintenance of the database and its applications.

Capable of handling a large variety of data types

Unlike traditional databases (such as hierarchical, network or relational), the object oriented database are capable of storing different types of data, for example, pictures, voice video, including text, numbers and so on.

Removal of impedance mismatch

A single language interface between the Data Manipulation Language (DML) and the programming language overcomes the impedance mismatch. This eliminates many of the efficiencies that occur in mapping a declarative language such as SQL to an imperative 'language such as 'C'. Most OODBMSs provide a DML that is computationally complete compared with SQL, the 'standard language of RDBMSs.

More expressive query language

Navigational access from the object is the most common form of data access in an OODBMS. This is in contrast to the associative access of SQL (that is, declarative statements with selection based on one or more predicates). Navigational access is more suitable for handling parts explosion, recursive queries, and so on.

Support for schema evolution

The tight coupling between data and applications in an OODBMS makes schema evolution more feasible.

Support for long-duration, transactions

Current relational DBMSs enforce serializability on concurrent transactions to maintain database consistency. OODBMSs use a different protocol to handle the types of long-duration transaction that are common in many advanced database application.

Applicability to advanced database applications

There are many areas where traditional DBMSs have not been particularly successful, such as, Computer-Aided Design (CAD), Computer-Aided Software Engineering (CASE), Office Information System (OIS), and Multimedia Systems. The enriched modeling capabilities of OODBMSs have made them suitable for these applications.

Improved performance

There have been a number of benchmarks that have suggested OODBMSs provide significant performance improvements over relational DBMSs. The results showed an average 30-fold performance improvement for the OODBMS over the RDBMS.

Disadvantages of OODBMS

Lack of universal data model: There is no universally agreed data model for an OODBMS, and most models lack a theoretical foundation. This disadvantage is seen as a significant drawback, and is comparable to per-relational systems.

Lack of experience: In comparison to RDBMSs the use of OODBMS is still relatively limited. This means that we do not yet have the level of experience that we have with traditional systems. OODBMSs are still very much geared towards the programmer, rather than the naïve end-user. Also there is a resistance to the acceptance of the technology. While the OODBMS is limited to a small niche market, this problem will continue to exist

Lack of standards: There is a general lack of standards of OODBMSs. We have already mentioned that there is not universally agreed data model. Similarly, there is no standard object-oriented query language.

Competition: Perhaps one of the most significant issues that face OODBMS vendors is the competition posed by the RDBMS and the emerging ORDBMS products. These products have an established user base with significant experience available. SQL is an approved standard

and the relational data model has a solid theoretical formation and relational products have many supporting tools to help .both end-users and developers.

Query optimization compromises encapsulations: Query optimization requires. An understanding of the underlying implementation to access the database efficiently. However, this compromises the concept of incrustation.

Locking at object level may impact performance Many OODBMSs use locking as the basis for concurrency control protocol. However, if locking is applied at the object level, locking of an inheritance hierarchy may be problematic, as well as impacting performance.

Complexity: The increased functionality provided by the OODBMS (such as the illusion of a single-level storage model, pointer sizzling, long-duration transactions, version management, and schema evolution–makes the system more complex than that of traditional DBMSs. In complexity leads to products that are more expensive and more difficult to use.

Lack of support for views: Currently, most OODBMSs do not provide a view mechanism, which, as we have seen previously, provides many advantages such as data independence, security, reduced complexity, and customization.

Lack of support for security: Currently, OODBMSs do not provide adequate security mechanisms. The user cannot grant access rights on individual objects or classes.

If OODBMSs are to expand fully into the business field, these deficiencies must be rectified.

DISTRIBUTED DATABASE

In a distributed database, there are a number of databases that may be geographically distributed all over the world. A distributed DBMS manages the distributed database in a manner so that it appears as one single database to users. A **distributed database** is a collection of multiple interconnected databases, which are spread physically across various locations that communicate via a computer network.

Features

- Databases in the collection are logically interrelated with each other. Often they represent a single logical database.
- Data is physically stored across multiple sites. Data in each site can be managed by a DBMS independent of the other sites.
- The processors in the sites are connected via a network. They do not have any multiprocessor configuration.
- A distributed database is not a loosely connected file system.
- A distributed database incorporates transaction processing, but it is not synonymous with a transaction processing system.

Distributed Database Management System

A distributed database management system (DDBMS) is a centralized software system that manages a distributed database in a manner as if it were all stored in a single location.

Features

- It is used to create, retrieve, update and delete distributed databases.
- It synchronizes the database periodically and provides access mechanisms by the virtue of which the distribution becomes transparent to the users.
- It ensures that the data modified at any site is universally updated.
- It is used in application areas where large volumes of data are processed and accessed by numerous users simultaneously.
- It is designed for heterogeneous database platforms.
- It maintains confidentiality and data integrity of the databases.

Advantages of Distributed Databases

Following are the advantages of distributed databases over centralized databases.

Modular Development − If the system needs to be expanded to new locations or new units, in centralized database systems, the action requires substantial efforts and disruption in the existing functioning. However, in distributed databases, the work simply requires adding new computers and local data to the new site and finally connecting them to the distributed system, with no interruption in current functions.

More Reliable − In case of database failures, the total system of centralized databases comes to a halt. However, in distributed systems, when a component fails, the functioning of the system continues may be at a reduced performance. Hence DDBMS is more reliable.

Better Response − If data is distributed in an efficient manner, then user requests can be met from local data itself, thus providing faster response. On the other hand, in centralized systems, all queries have to pass through the central computer for processing, which increases the response time.

Lower Communication Cost − In distributed database systems, if data is located locally where it is mostly used, then the communication costs for data manipulation can be minimized. This is not feasible in centralized systems.

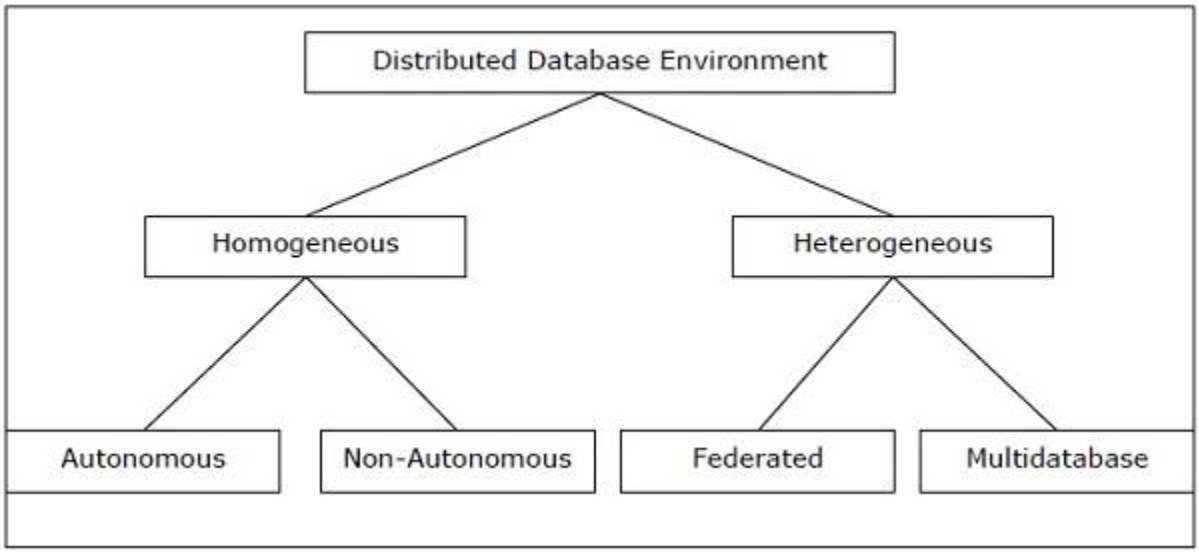
Characteristics of Distributed DDBMS :

A DDBMS has the following characteristics-

1. A collection of logically related shared data.
2. The data is split into a number of fragments.
3. Fragments may be duplicate.
4. Fragments are allocated to sites.
5. The data at each site is under the control of DBMS and managed by DBMS.

Types of Distributed Databases

Distributed databases can be broadly classified into homogeneous and heterogeneous distributed database environments, each with further sub-divisions, as shown in the following illustration.



Homogeneous Distributed Databases

In a homogeneous distributed database, all the sites use identical DBMS and operating systems. Its properties are −

- The sites use very similar software.
- The sites use identical DBMS or DBMS from the same vendor.
- Each site is aware of all other sites and cooperates with other sites to process user requests.
- The database is accessed through a single interface as if it is a single database.

Types of Homogeneous Distributed Database

There are two types of homogeneous distributed database −

- **Autonomous** − Each database is independent that functions on its own. They are integrated by a controlling application and use message passing to share data updates.

- **Non-autonomous** − Data is distributed across the homogeneous nodes and a central or master DBMS co-ordinates data updates across the sites.

Heterogeneous Distributed Databases

In a heterogeneous distributed database, different sites have different operating systems, DBMS products and data models. Its properties are −

- Different sites use dissimilar schemas and software.
- The system may be composed of a variety of DBMSs like relational, network, hierarchical or object oriented.
- Query processing is complex due to dissimilar schemas.
- Transaction processing is complex due to dissimilar software.
- A site may not be aware of other sites and so there is limited co-operation in processing user requests.

Types of Heterogeneous Distributed Databases

- **Federated** − The heterogeneous database systems are independent in nature and integrated together so that they function as a single database system.
- **Un-federated** − The database systems employ a central coordinating module through which the databases are accessed.

Distributed DBMS Architectures

DDBMS architectures are generally developed depending on three parameters −

- **Distribution** − It states the physical distribution of data across the different sites.
- **Autonomy** − It indicates the distribution of control of the database system and the degree to which each constituent DBMS can operate independently.

- **Heterogeneity** − It refers to the uniformity or dissimilarity of the data models, system components and databases.

Architectural Models

Some of the common architectural models are −

- Client - Server Architecture for DDBMS
- Peer - to - Peer Architecture for DDBMS
- Multi - DBMS Architecture

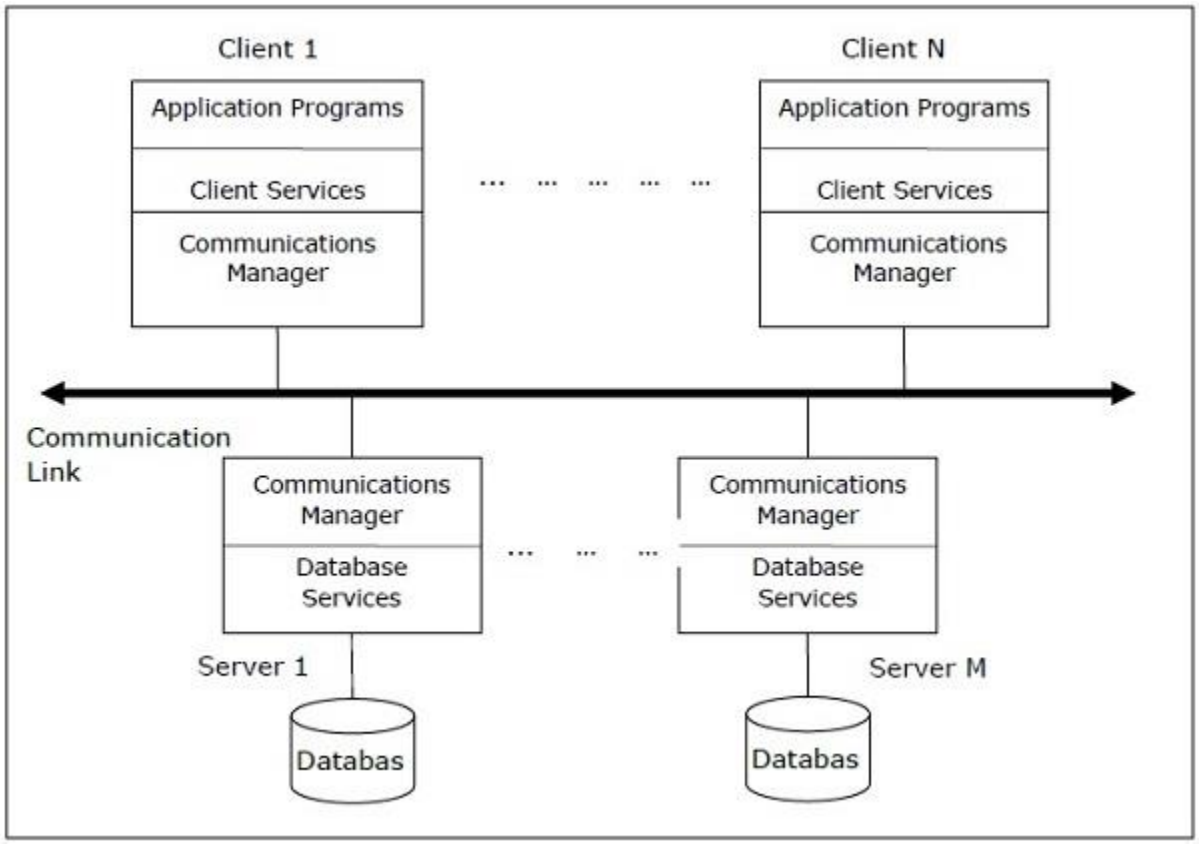
Client - Server Architecture for DDBMS

This is a two-level architecture where the functionality is divided into servers and clients. The server functions primarily encompass data management, query processing, optimization and transaction management. Client functions include mainly user interface. However, they have some functions like consistency checking and transaction management.

- A client server architecture has a number of clients and a few servers connected in a network.
- A client sends a query to one of the servers. The earliest available server solves it and replies.
- A Client-server architecture is simple to implement and execute due to centralized server system.

The two different client - server architecture are −

- Single Server Multiple Client
- Multiple Server Multiple Client (shown in the following diagram)

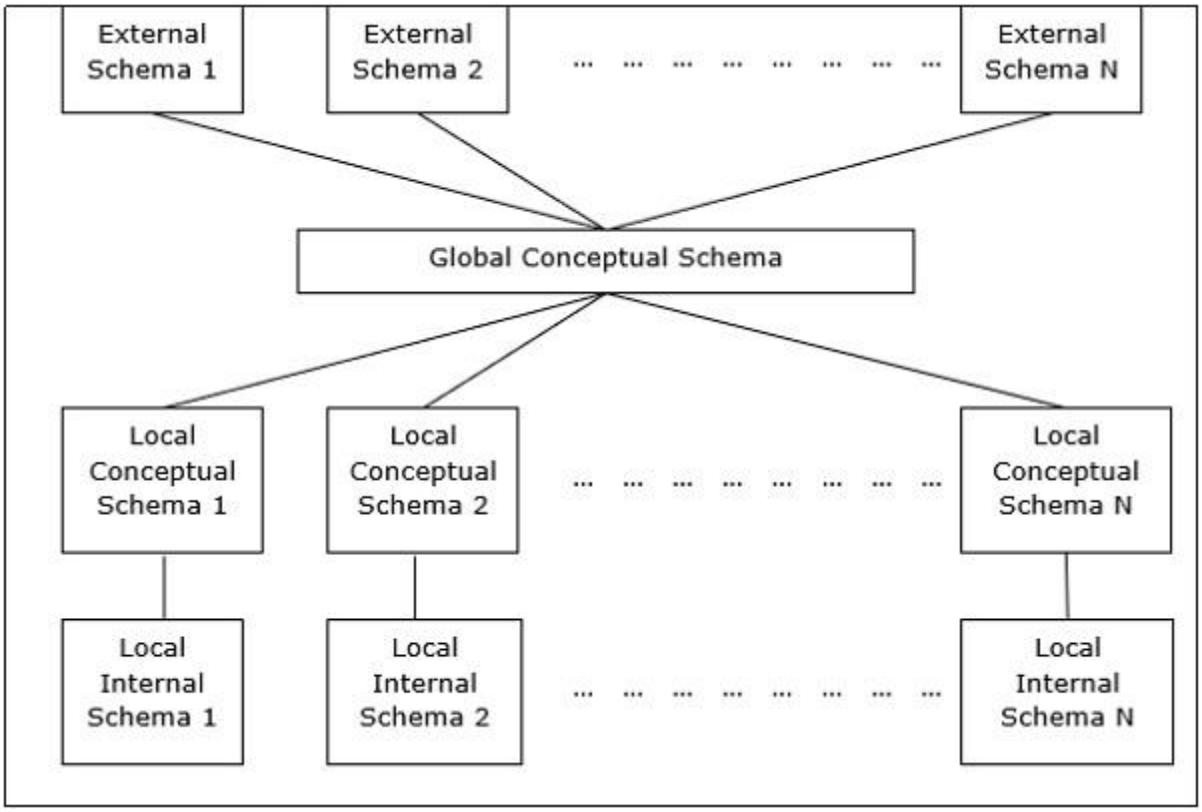


Peer- to-Peer Architecture for DDBMS

In these systems, each peer acts both as a client and a server for imparting database services. The peers share their resource with other peers and co-ordinate their activities.

This architecture generally has four levels of schemas −

- **Global Conceptual Schema** − Depicts the global logical view of data.
- **Local Conceptual Schema** − Depicts logical data organization at each site.
- **Local Internal Schema** − Depicts physical data organization at each site.
- **External Schema** − Depicts user view of data.



Multi - DBMS Architectures

This is an integrated database system formed by a collection of two or more autonomous database systems.

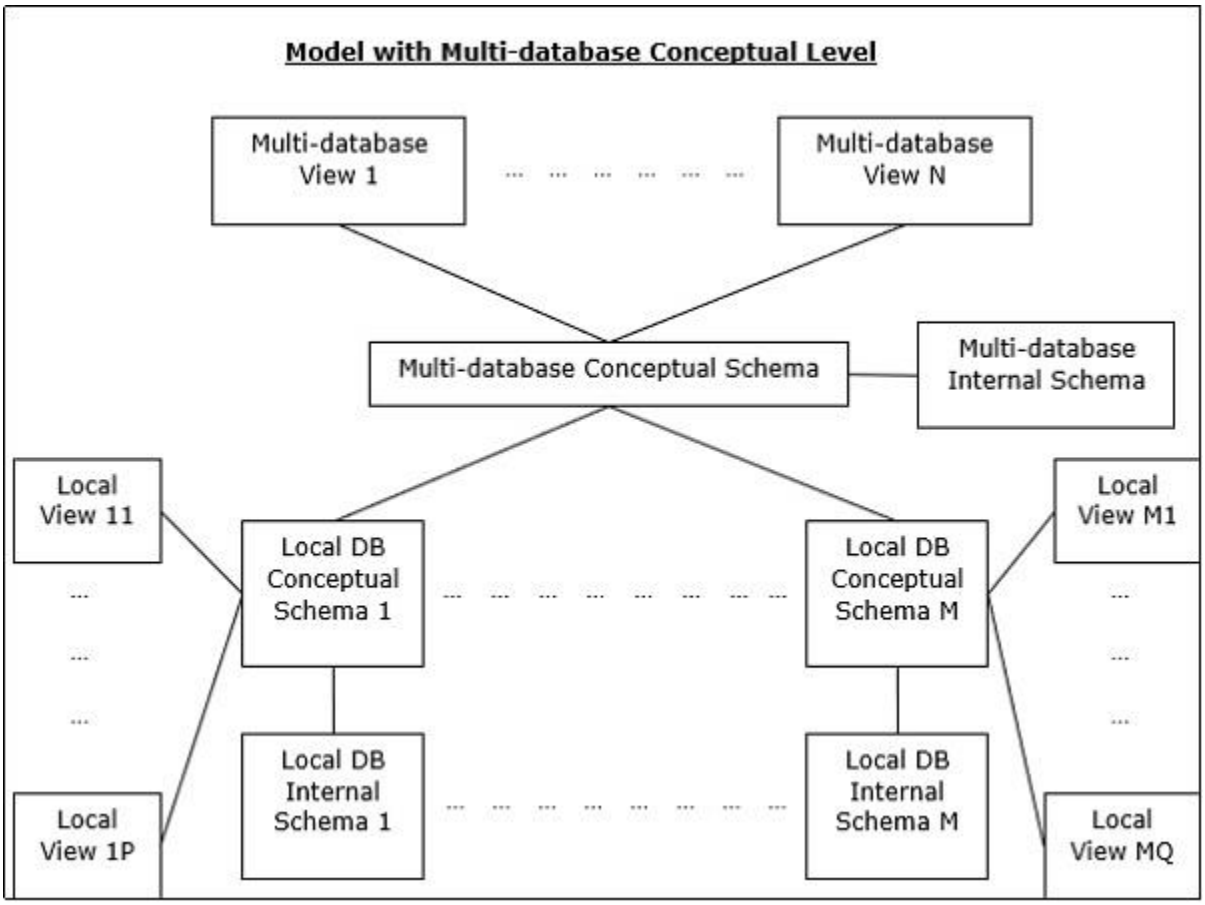
Multi-DBMS can be expressed through six levels of schemas −

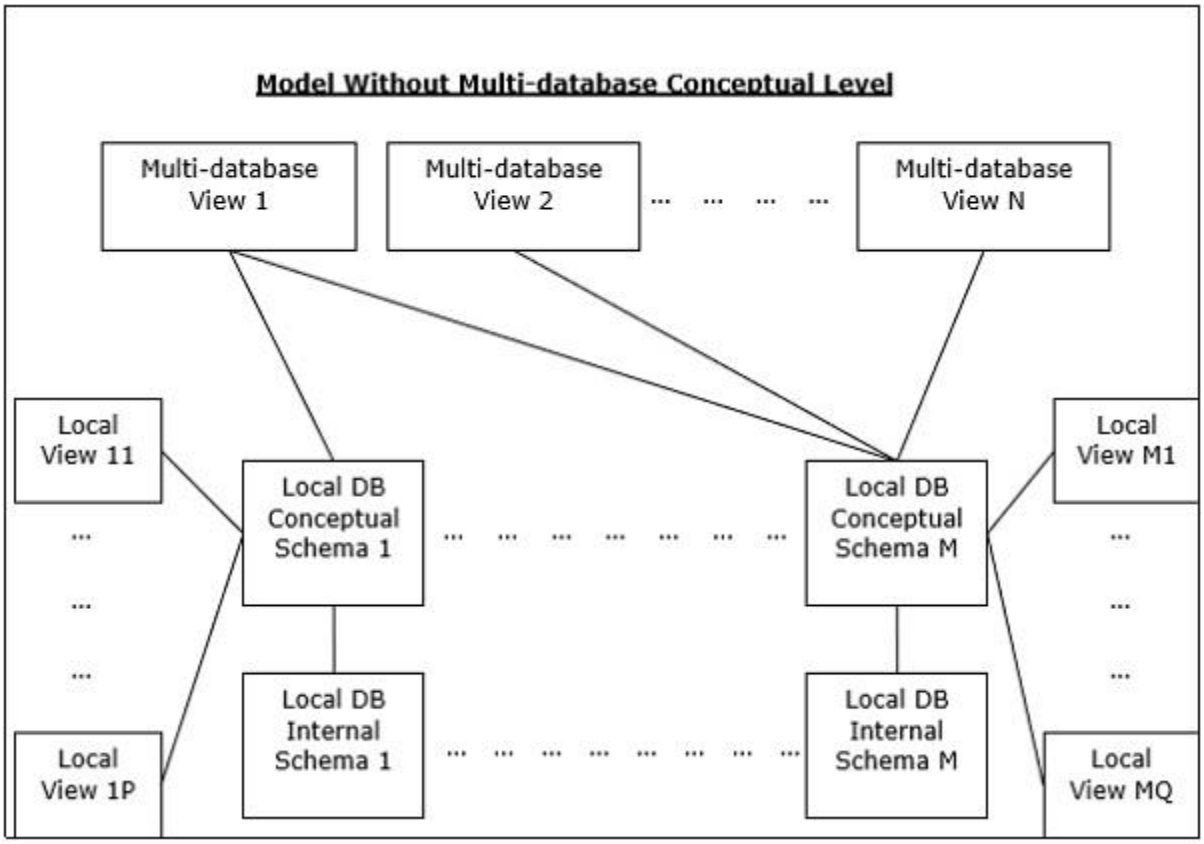
- **Multi-database View Level** − Depicts multiple user views comprising of subsets of the integrated distributed database.
- **Multi-database Conceptual Level** − Depicts integrated multi-database that comprises of global logical multi-database structure definitions.
- **Multi-database Internal Level** − Depicts the data distribution across different sites and multi-database to local data mapping.
- **Local database View Level** − Depicts public view of local data.
- **Local database Conceptual Level** − Depicts local data organization at each site.

- **Local database Internal Level** − Depicts physical data organization at each site.

There are two design alternatives for multi-DBMS −

- Model with multi-database conceptual level.
- Model without multi-database conceptual level.





Design Alternatives

The distribution design alternatives for the tables in a DDBMS are as follows −

- Non-replicated and non-fragmented
- Fully replicated
- Partially replicated
- Fragmented
- Mixed

Non-replicated & Non-fragmented

In this design alternative, different tables are placed at different sites. Data is placed so that it is at a close proximity to the site where it is used most. It is most suitable for database systems where the percentage of queries needed to join information in tables placed at different sites is

low. If an appropriate distribution strategy is adopted, then this design alternative helps to reduce the communication cost during data processing.

Fully Replicated

In this design alternative, at each site, one copy of all the database tables is stored. Since, each site has its own copy of the entire database, queries are very fast requiring negligible communication cost. On the contrary, the massive redundancy in data requires huge cost during update operations. Hence, this is suitable for systems where a large number of queries is required to be handled whereas the number of database updates is low.

Partially Replicated

Copies of tables or portions of tables are stored at different sites. The distribution of the tables is done in accordance to the frequency of access. This takes into consideration the fact that the frequency of accessing the tables vary considerably from site to site. The number of copies of the tables (or portions) depends on how frequently the access queries execute and the site which generate the access queries.

Fragmented

In this design, a table is divided into two or more pieces referred to as fragments or partitions, and each fragment can be stored at different sites. This considers the fact that it seldom happens that all data stored in a table is required at a given site. Moreover, fragmentation increases parallelism and provides better disaster recovery. Here, there is only one copy of each fragment in the system, i.e. no redundant data.

The three fragmentation techniques are −

- Vertical fragmentation
- Horizontal fragmentation
- Hybrid fragmentation

Mixed Distribution

This is a combination of fragmentation and partial replications. Here, the tables are initially fragmented in any form (horizontal or vertical), and then these fragments are partially replicated across the different sites according to the frequency of accessing the fragments.

Distributed transaction

A distributed transaction is a type of transaction with two or more engaged network hosts. Generally, hosts provide resources, and a transaction manager is responsible for developing and handling the transaction. Like any other transaction, a distributed transaction should include all four ACID properties (atomicity, consistency, isolation, durability). Given the nature of the work, atomicity is important to ensure an all-or-nothing outcome for the operations bundle (unit of work).

Databases are standard transactional resources, and transactions usually extend to a small number of such databases. In such cases, a distributed transaction may be viewed as a database transaction that should be synchronized between various participating databases allocated between various physical locations. The isolation property presents a unique obstacle for multi-database transactions.
For distributed transactions, each computer features a local transaction manager. If the transaction works at several computers, the transaction managers communicate with various other transaction managers by means of superior or subordinate relationships, which are accurate only for a specific transaction.

Resource managers handle consistent or resilient data and closely cooperate with the distributed transaction coordinator (DTC) to ensure an application's isolation and atomicity. In distributed transactions, every participating element should conform to committing a change action, such as a database update, prior to the transaction. The DTC coordinates the transaction for the participating components and works as a transaction manager for each computer that is meant to

manage the transactions. When distributing transactions between various computers, the transaction manager delivers, prepares, commits and aborts messages to each subordinate transaction manager.

In the DTC's two-phase commit algorithm, phase one involves the transaction manager prompting commitment preparation of each enlisted component, whereas in phase two, if all components are prepared to successfully commit, the transaction manager messages the decision to commit.

Commit protocols for distributed databases

In a local database system, for committing a transaction, the transaction manager has to only convey the decision to commit to the recovery manager. However, in a distributed system, the transaction manager should convey the decision to commit to all the servers in the various sites where the transaction is being executed and uniformly enforce the decision. When processing is complete at each site, it reaches the partially committed transaction state and waits for all other transactions to reach their partially committed states. When it receives the message that all the sites are ready to commit, it starts to commit. In a distributed system, either all sites commit or none of them does.

The different distributed commit protocols are −

- One-phase commit
- Two-phase commit
- Three-phase commit

Distributed One-phase Commit

Distributed one-phase commit is the simplest commit protocol. Let us consider that there is a controlling site and a number of slave sites where the transaction is being executed. The steps in distributed commit are −

- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site.
- The slaves wait for “Commit” or “Abort” message from the controlling site. This waiting time is called **window of vulnerability**.
- When the controlling site receives “DONE” message from each slave, it makes a decision to commit or abort. This is called the commit point. Then, it sends this message to all the slaves.
- On receiving this message, a slave either commits or aborts and then sends an acknowledgement message to the controlling site.

Distributed Two-phase Commit

Distributed two-phase commit reduces the vulnerability of one-phase commit protocols. The steps performed in the two phases are as follows −

Phase 1: Prepare Phase

- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site. When the controlling site has received “DONE” message from all slaves, it sends a “Prepare” message to the slaves.
- The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a “Ready” message.
- A slave that does not want to commit sends a “Not Ready” message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

Phase 2: Commit/Abort Phase

- After the controlling site has received “Ready” message from all the slaves −
    - The controlling site sends a “Global Commit” message to the slaves.

- The slaves apply the transaction and send a “Commit ACK” message to the controlling site.
- When the controlling site receives “Commit ACK” message from all the slaves, it considers the transaction as committed.

- After the controlling site has received the first “Not Ready” message from any slave −
  - The controlling site sends a “Global Abort” message to the slaves.
  - The slaves abort the transaction and send a “Abort ACK” message to the controlling site.
  - When the controlling site receives “Abort ACK” message from all the slaves, it considers the transaction as aborted.

Distributed Three-phase Commit

The steps in distributed three-phase commit are as follows −

Phase 1: Prepare Phase

The steps are same as in distributed two-phase commit.

Phase 2: Prepare to Commit Phase

- The controlling site issues an “Enter Prepared State” broadcast message.
- The slave sites vote “OK” in response.

Phase 3: Commit / Abort Phase

The steps are same as two-phase commit except that “Commit ACK”/”Abort ACK” message is not required.