**DBMS & SQL NOTES**

**Database:**  Database is a collection of inter-related data which helps in efficient retrieval, insertion and deletion of data from database and organizes the data in the form of tables, views, schemas, reports etc.

For Example, university database organizes the data about students, faculty, and admin staff etc. which helps in efficient retrieval, insertion and deletion of data from it.

## What is DBMS?

DBMS or Database Management System is a software application used to access, create, and manage databases. With the help of DBMS, you can easily create, retrieve and update data in databases acts as an interface between the end-users and the database. Refer below.

The following are a few characteristics of DBMS:

* To **limit** the **permissions** of the users
* Provide multiple **views** of the single database schema
* Facilitates **security** and removes data redundancy
* Allows**multi-user transaction** processing and sharing of data
* Follows the **ACID** property
* Offers both physical and logical data independence

RDBMS :

**RDBMS** stands for *Relational Database Management Systems.*.

All modern database management systems like SQL, MS SQL Server, IBM DB2, ORACLE, My-SQL and Microsoft Access are based on RDBMS

It is called Relational Data Base Management System (RDBMS) because it is based on relational model introduced by E.F. Codd.

The RDBMS database uses tables to store data. A table is a collection of related data entries and contains rows and columns to store data.

. A table is the simplest example of data storage in RDBMS.

Every table is broken up into smaller entities called fields. The fields in the CUSTOMERS table consist of ID, NAME, AGE, ADDRESS and SALARY.

A row of a table is also called record. It contains the specific information of each individual entry in the table. It is a horizontal entity in the table. For example: The above table contains 5 records.

A column is a vertical entity in the table which contains all information associated with a specific field in a table. For example: "name" is a column in the above table which contains all information about student's name.

## Data Integrity

There are the following categories of data integrity exist with each RDBMS:

**Entity integrity**: It specifies that there should be no duplicate rows in a table.

**Domain integrity**: It enforces valid entries for a given column by restricting the type, the format, or the range of values.

**Referential integrity**: It specifies that rows cannot be deleted, which are used by other records.

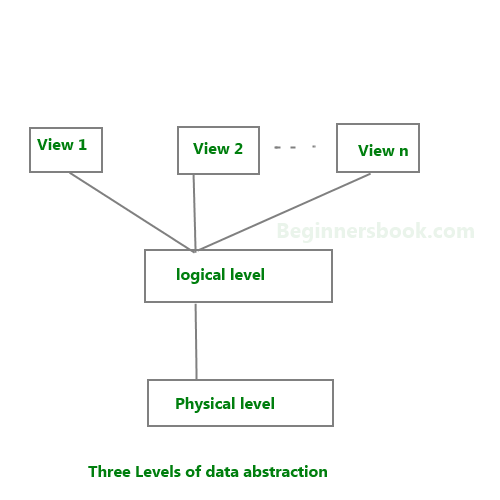
**User-defined integrity**: It enforces some specific business rules that are defined by users. These rules are different from entity, domain or referential integrity.

What is relation : A relation in a database means a ‘TABLE’. As we all know, tables are something which is made up of rows and columns, a relation in database will also have rows and columns. Each row is called a tuple. Each column is called an attribute or field.

. A domain is a unique set of values permitted for an attribute in a table. For example, a domain of month-of-year can accept January, February….December as possible values, a domain of integers can accept whole numbers that are negative, positive and zero.

DATABASE ABSTRACTION : <https://beginnersbook.com/2015/04/levels-of-abstraction-in-dbms/>

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



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

**Logical level**: This is the middle level of 3-level data abstraction architecture. It describes what data is stored in database. It also stores the relationship among the data entities in relatively simple structures

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

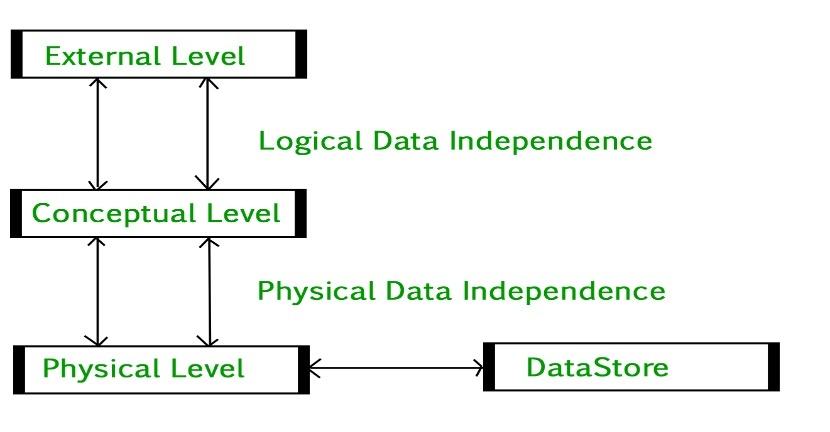
**Example**: Let’s say we are storing customer information in a customer table. At **physical level** these records can be described as blocks of storage (bytes, gigabytes, terabytes etc.) in memory. These details are often hidden from the programmers.

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

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

***Three Schema Architecture***

<https://www.geeksforgeeks.org/introduction-of-3-tier-architecture-in-dbms-set-2/>

DBMS 3-tier architecture divides the complete system into three inter-related but independent modules as shown below:  


1. **Physical Level:** At the physical level, the information about the location of database objects in the data store is kept. Various users of DBMS are unaware of the locations of these objects.In simple terms,physical level of a database describes how the data is being stored in secondary storage devices like disks and tapes and also gives insights on additional storage details.
2. **Conceptual Level:**At conceptual level, data is represented in the form of various database tables. For Example, STUDENT database may contain STUDENT and COURSE tables which will be visible to users but users are unaware of their storage.Also referred as logical schema,it describes what kind of data is to be stored in the database.
3. **External Level:** An external level specifies a view of the data in terms of conceptual level tables.  Each external level view is used to cater to the needs of a particular category of users. For Example, FACULTY of a university is interested in looking course details of students, STUDENTS are interested in looking at all details related to academics, accounts, courses and hostel details as well. So, different views can be generated for different users. The main focus of external level is data abstraction.

Data interdependency: Data independence means a change of data at one level should not affect another level. Two types of data independence are present in this architecture:

* **Physical Data Independence** If we do any changes in the storage size of the database system server, then the Conceptual structure of the database will not be affected.

For example, in case we want to change or upgrade the storage system itself − suppose we want to replace hard-disks with SSD − it should not have any impact on the logical data or schemas.

**Conceptual Data Independence:** The data at conceptual level schema and external level schema must be independent. This means a change in conceptual schema should not affect external schema. e.g.; Adding or deleting attributes of a table should not affect the user’s view of the table. But this type of independence is difficult to achieve as compared to physical data independence because the changes in conceptual schema are reflected in the user’s view.

Tier 2 and tier 3 architecture :

There are direct communication between a **client** and a **server**with no intermediary. It is divided into two parts;

1. ) Client Application.

2. ) Database.

The client in a Two-tier**architecture** application has the code written for saving data in the database. The client sends a request to the server, where it then processes the request and sends back the data. Meaning the client handles both the Presentation layer (application interface) and Application layer (logical operations), while the server system handles the database layer.

The **Three-tier Architecture** is divided into 3 parts;

1. ) Presentation Layer (Client tier)
2. ) Application Layer (Business tier)
3. ) Database Layer (Data tier)

The **Presentation Layer** is the topmost layer of an application. This is the layer seen when using the software(**Interface**, **web** **pages.**  Its main function is to communicate with the application layer. This layer passes information which is given by the user in terms of keyboard actions and mouse clicks to the **application** **layer**.

The **Application Layer**is also known as the **Business** **logic** **layer.**Here’s where we find logic controls and functionality that processes data received from the presentation layer and database layer. It acts as an intermediary between the **presentation** and **database** layer.

The **Database Layer**ere information is stored and retrieved. This layer consists of data access components to aid in resource sharing.

Tier 2 vs tier 3 <https://www.geeksforgeeks.org/difference-between-two-tier-and-three-tier-database-architecture/>

Intension and extention of database:

Intension is the permanent part of the relation and comprises of two things: relation schema and the integrity constraints. Relation schema defines the name and attributes of the relation, and integrity constraints define key constraints, referential constraints. etc. it time independent.

For example : Intension of student :

Student (RollNo Number(4) Not NULL, Name Char(20), Age Number(2), Course Char(15) )

The extension of a given relation is the set of tuples appearing in that relation at any given instance. The extension thus varies with time. It changes as tuples are created, destroyed, and updated.

**Relation: Employee at time= t1**

|  |  |  |  |
| --- | --- | --- | --- |
| **EmpNo** | **EmpName** | **Age** | **Dept** |
| 1001 | Jason | 23 | SD |
| 1002 | William | 24 | HR |
| 1003 | Jonathan | 28 | Fin |
| 1004 | Harry | 20 | Fin |

**Relation: Employee at time= t2 after adding more records**

|  |  |  |  |
| --- | --- | --- | --- |
| **EmpNo** | **EmpName** | **Age** | **Dept** |
| 1001 | Jason | 23 | SD |
| 1002 | William | 24 | HR |
| 1003 | Jonathan | 28 | Fin |
| 1004 | Harry | 20 | Fin |
| 1005 | Smith | 22 | HR |
| 1006 | Mary | 19 | HR |
| 1007 | Sarah | 23 | SD |

**Relation: Employee at time= t2 after adding more records**

|  |  |  |  |
| --- | --- | --- | --- |
| **EmpNo** | **EmpName** | **Age** | **Dept** |
| 1001 | Jason | 23 | SD |
| 1002 | William | 24 | HR |

**Database Management System (DBMS)** is a software for storing and retrieving users' data while considering appropriate security measures. It consists of a group of programs which manipulate the database. The DBMS accepts the request for data from an application and instructs the operating system to provide the specific data. In large systems, a DBMS helps users and other third-party software to store and retrieve data.

Database management systems were developed to handle the following difficulties of typical File-processing systems supported by conventional operating systems.

1. Data redundancy and inconsistency

2. Difficulty in accessing data

3. Data isolation – multiple files and formats

4. Integrity problems

5. Atomicity of updates

6. Concurrent access by multiple users

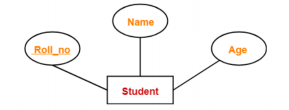
7. Security problems

**ER diagram:**

● ER diagram or **Entity Relationship diagram** is a conceptual model that gives the graphical representation of the logical structure of the database.

● It shows all the constraints and relationships that exist among the different components.

● An ER diagram is mainly composed of following three components- Entity Sets, Attributes and Relationship Set.



● Roll\_no is a primary key that can identify each entity uniquely.

● Thus, by using a student's roll number, a student can be identified uniquely.

**Entity Set:**

An entity set is a set of the same type of entities.

**● Strong Entity Set:**

**o** A strong entity set is an entity set that contains sufficient attributes to uniquely identify all its entities.

o In other words, a primary key exists for a strong entity set.

o Primary key of a strong entity set is represented by underlining it.

● **Weak Entity Set:**

o A weak entity set is an entity set that does not contain sufficient attributes to uniquely identify its entities.

o In other words, a primary key does not exist for a weak entity set.

o However, it contains a partial key called a discriminator.

o Discriminator can identify a group of entities from the entity set.

o Discriminator is represented by underlining with a dashed line.

**Relationship:**

A relationship is defined as an association among several entities.

● **Unary Relationship Set -** Unary relationship set is a relationship set where only one entity set participates in a relationship set.

● **Binary Relationship Set -** Binary relationship set is a relationship set where two entity sets participate in a relationship set.

● **Ternary Relationship Set -** Ternary relationship set is a relationship set where three entity sets participate in a relationship set.

● **N-ary Relationship Set -** N-ary relationship set is a relationship set where ‘n’ entity sets participate in a relationship set.

**Cardinality Constraint:**

Cardinality constraint defines the maximum number of relationship instances in which an entity can participate.

● **One-to-One Cardinality -** An entity in set A can be associated with at most one entity in set B. An entity in set B can be associated with at most one entity in set A.

**● One-to-Many Cardinality -** An entity in set A can be associated with any number (zero or more) of entities in set B. An entity in set B can be associated with at most one entity in set A.

**● Many-to-One Cardinality -** An entity in set A can be associated with at most one entity in set B. An entity in set B can be associated with any number of entities in set A. **● Many-to-Many Cardinality -** An entity in set A can be associated with any number (zero or more) of entities in set B. An entity in set B can be associated with any number (zero or more) of entities in set A.

**Attributes:**

Attributes are the descriptive properties which are owned by each entity of an Entity Set. **Types of Attributes:**

● **Simple Attributes -** Simple attributes are those attributes which cannot be divided further. Ex. Age

● **Composite Attributes -** Composite attributes are those attributes which are composed of many other simple attributes. Ex. Name, Address

● **Multi Valued Attributes -** Multi valued attributes are those attributes which can take more than one value for a given entity from an entity set. Ex. Mobile No, Email ID ● **Derived Attributes -** Derived attributes are those attributes which can be derived from other attribute(s). Ex. Age can be derived from DOB.

● **Key Attributes -** Key attributes are those attributes which can identify an entity uniquely in an entity set. Ex. Roll No.

**Constraints:**

Relational constraints are the restrictions imposed on the database contents and operations. They ensure the correctness of data in the database.

**● Domain Constraint -** Domain constraint defines the domain or set of values for an attribute. It specifies that the value taken by the attribute must be the atomic value from its domain.

**● Tuple Uniqueness Constraint -** Tuple Uniqueness constraint specifies that all the tuples must be necessarily unique in any relation.

● **Key Constraint -** All the values of the primary key must be unique. The value of the primary key must not be null.

● **Entity Integrity Constraint -** Entity integrity constraint specifies that no attribute of primary key must contain a null value in any relation.

● **Referential Integrity Constraint -** It specifies that all the values taken by the foreign key must either be available in the relation of the primary key or be null.

**Closure of an Attribute Set:**

The set of all those attributes which can be functionally determined from an attribute set is called a closure of that attribute set.

**Keys:**

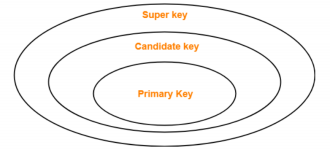
 key is an attribute or a set of attributes that help to uniquely identify a tuple (or row) in a relation (or table). Keys are also used to establish relationships between the different tables and columns of a relational database. Individual values in a key are called key values.

**. Types of Keys:**

**● Super Key -** A super key is a set of attributes that can identify each tuple uniquely in the given relation. A super key may consist of any number of attributes.

**● Candidate Key -** A set of minimal attribute(s) that can identify each tuple uniquely in the given relation is called a candidate key.

**● Primary Key -** A primary key is a candidate key that the database designer selects while designing the database. Primary Keys are unique and NOT NULL.



**● Alternate Key -** Candidate keys that are left unimplemented or unused after implementing the primary key are called as alternate keys.

**● Foreign Key -** An attribute ‘X’ is called as a foreign key to some other attribute ‘Y’ when its values are dependent on the values of attribute ‘Y’. The relation in which attribute ‘Y’ is present is called as the referenced relation. The relation in which attribute ‘X’ is present is called as the referencing relation.

A foreign key is an attribute value in a table that acts as the primary key in another table. Hence, the foreign key is useful in linking together two tables. Data should be entered in the foreign key column with great care, as wrongly entered data can invalidate the relationship between the two tables.

**● Composite Key -** A primary key composed of multiple attributes and not just a single attribute is called a composite key.

**● Unique Key -** It is unique for all the records of the table. Once assigned, its value cannot be changed i.e. it is non-updatable. It may have a NULL value.

* A prime attribute is an attribute that is part of any candidate key. It can also be used to uniquely identify a tuple in the schema. A prime attribute in DBMS is also known as a key attribute.
* A non-prime attribute is one that is not part of one of the candidate keys. It cannot be used to uniquely identify a tuple in the schema. A non-prime attribute in DBMS is also known as a non-key attribute.

**What is the difference between primary key and unique constraints?**   
The primary key cannot have NULL value, the unique constraints can have NULL values. There is only one primary key in a table, but there can be multiple unique constrains.

**Functional Dependency:**

[**https://www.geeksforgeeks.org/types-of-functional-dependencies-in-dbms/**](https://www.geeksforgeeks.org/types-of-functional-dependencies-in-dbms/)

In any relation, a functional dependency α → β holds if- Two tuples having same value of attribute α also have same value for attribute β.

**Types of Functional Dependency:**

● **Trivial Functional Dependencies –**

o A functional dependency X → Y is said to be trivial if and only if Y ⊆ X. o Thus, if RHS of a functional dependency is a subset of LHS, then it is called a trivial functional dependency.

● **Non-Trivial Functional Dependencies –**

o A functional dependency X → Y is said to be non-trivial if and only if Y ⊄ X. o Thus, if there exists at least one attribute in the RHS of a functional dependency that is not a part of LHS, then it is called a non-trivial functional dependency.

**Decomposition of a Relation:**

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

**Properties of Decomposition:**

**● Lossless Decomposition -** Lossless decomposition ensures

o No information is lost from the original relation during decomposition. **o** When the sub relations are joined back, the same relation is obtained that was decomposed.

**● Dependency Preservation -** Dependency preservation ensures

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

**Types of Decomposition:**

**● Lossless Join Decomposition:**

o Consider there is a relation R which is decomposed into sub relations R1, R2, …., Rn.

o This decomposition is called lossless join decomposition when the join of the sub relations results in the same relation R that was decomposed.

**o** For lossless join decomposition, we always have- R1 ⋈ R2 ⋈ R3 ……. ⋈ Rn = R where ⋈ is a natural join operator

**● Lossy Join Decomposition:**

o Consider there is a relation R which is decomposed into sub relations R1, R2, …., Rn.

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

o For lossy join decomposition, we always have- R1 ⋈ R2 ⋈ R3 ……. ⋈ Rn ⊃ R where ⋈ is a natural join operator

**Normalization:**

**Normalization** is a database design technique that reduces data redundancy and eliminates undesirable characteristics like Insertion, Update and Deletion Anomalies. Normalization rules divides larger tables into smaller tables and links them using relationships. The purpose of Normalisation in SQL is to eliminate redundant (repetitive) data and ensure data is stored logically.

In DBMS, database normalization is a process of making the database consistent by-

● Reducing the redundancies

● Ensuring the integrity of data through lossless decomposition

**Normal Forms:**

● **First Normal Form (1NF) -** A given relation is called in First Normal Form (1NF) if each cell of the table contains only an atomic value i.e. if the attribute of every tuple is either single valued or a null value.

● **Second Normal Form (2NF) -** A given relation is called in Second Normal Form (2NF) if and only if

o Relation already exists in 1NF.

o No partial dependency exists in the relation.

A → B is called a **partial dependency** if and only if- A is a subset of some candidate key and B is a non-prime attribute. Partial Dependency occurs when a non-prime attribute is functionally dependent on part of a candidate key.

● **Third Normal Form (3NF) -** A given relation is called in Third Normal Form (3NF) if and only if

o Relation already exists in 2NF.

o No transitive dependency exists for non-prime attributes.

A → B is called a **transitive dependency** if and only if- A is not a super key and B is a non-prime attribute.

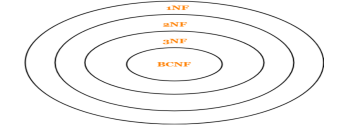
**Relation should not have a non-key attribute functionally determined by another non-key attribute (or by a sets of non-key attributes)**

● **Boyce-Codd Normal Form -** A given relation is called in BCNF if and only if o Relation already exists in 3NF.

o For each non-trivial functional dependency ‘A → B’, A is a super key of the relation.

**for every functional dependency , Make sure that the left side of every functional dependency is a candidate key.**

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

**Transaction:**

It is a set of operation used to perform logical unit of work.

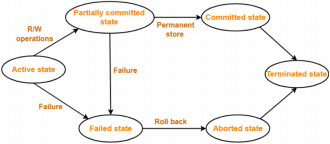
Ex->money withdraw from ATM machine. Here work is money withdrawn.

**Operations in Transaction:**

● **Read Operation - Read(A)** instruction will read the value of ‘A’ from the database and will store it in the buffer in main memory.

● **Write Operation – Write(A)** will write the updated value of ‘A’ from the buffer to the database.

**Transaction States:**

****

**● Active State –**

o This is the first state in the life cycle of a transaction.

o A transaction is called in an active state as long as its instructions are getting executed.

**o** All the changes made by the transaction now are stored in the buffer in main memory.

**● Partially Committed State –**

o After the last instruction of the transaction has been executed, it enters into a partially committed state.

o After entering this state, the transaction is considered to be partially committed.

**o** It is not considered fully committed because all the changes made by the transaction are still stored in the buffer in main memory.

**● Committed State –**

o After all the changes made by the transaction have been successfully stored into the database, it enters into a committed state.

**o** Now, the transaction is considered to be fully committed.

**● Failed State –**

o When a transaction is getting executed in the active state or partially committed state and some failure occurs due to which it becomes impossible to continue the execution, it enters into a failed state.

● **Aborted State –**

o After the transaction has failed and entered into a failed state, all the changes made by it have to be undone.

o To undo the changes made by the transaction, it becomes necessary to roll back the transaction.

o After the transaction has rolled back completely, it enters into an aborted state. ● **Terminated State –**

o This is the last state in the life cycle of a transaction.

o After entering the committed state or aborted state, the transaction finally enters into a terminated state where its life cycle finally comes to an end.

**ACID Properties:**

To ensure the consistency of the database, certain properties are followed by all the transactions occurring in the system. These properties are called as **ACID Properties** of a transaction.

● **Atomicity –**

o This property ensures that either the transaction occurs completely or it does not occur at all.

o In other words, it ensures that no transaction occurs partially.

● **Consistency –**

o This property ensures that integrity constraints are maintained.

o In other words, it ensures that the database remains consistent before and after the transaction.

● **Isolation –**

o This property ensures that multiple transactions can occur simultaneously without causing any inconsistency.

o The resultant state of the system after executing all the transactions is the same as the state that would be achieved if the transactions were executed serially one after the other.

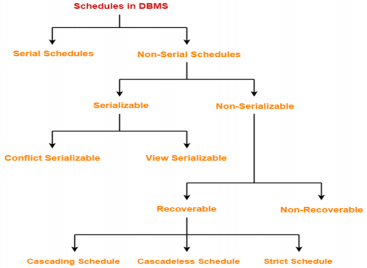
● **Durability –**

o This property ensures that all the changes made by a transaction after its successful execution are written successfully to the disk.

o It also ensures that these changes exist permanently and are never lost even if there occurs a failure of any kind.

**Schedules:**

The order in which the operations of multiple transactions appear for execution is called as a schedule.



**● Serial Schedules –**

o All the transactions execute serially one after the other.

o When one transaction executes, no other transaction is allowed to execute. **o** Serial schedules are always- Consistent, Recoverable, Cascadeless and Strict. **● Non-Serial Schedules –**

o Multiple transactions execute concurrently.

**o** Operations of all the transactions are inter leaved or mixed with each other.

**o** Non-serial schedules are **not** always- Consistent, Recoverable, Cascadeless and Strict.

**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](https://beginnersbook.com/2018/12/dbms-schedules/) are serializable. A serializable schedule always leaves the database in consistent state. A [serial schedule](https://beginnersbook.com/2018/12/dbms-schedules/) 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.

o If a given non-serial schedule of ‘n’ transactions is equivalent to some serial schedule of ‘n’ transactions, then it is called as a serializable schedule.

**o** Serializable schedules are always- Consistent, Recoverable, Cascadeless and Strict.

**Types of Serializability –**

**● Conflict Serializability** - If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called a conflict serializable schedule.

**● View Serializability -** If a given schedule is found to be viewed as equivalent to some serial schedule, then it is called a view serializable schedule.

**Non-Serializable Schedules –**

● A non-serial schedule which is not serializable is called a non-serializable schedule. ● A non-serializable schedule is not guaranteed to produce the same effect as produced by some serial schedule on any consistent database.

**●** Non-serializable schedules- may or may not be consistent, may or may not be recoverable.

**● Irrecoverable Schedules –**

If in a schedule,

o A transaction performs a dirty read operation from an uncommitted transaction o And commits before the transaction from which it has read the value then such a schedule is known as an Irrecoverable Schedule.

**● Recoverable Schedules –**

If in a schedule,

o A transaction performs a dirty read operation from an uncommitted transaction o And its commit operation is delayed till the uncommitted transaction either commits or roll backs

then such a schedule is known as a Recoverable Schedule.

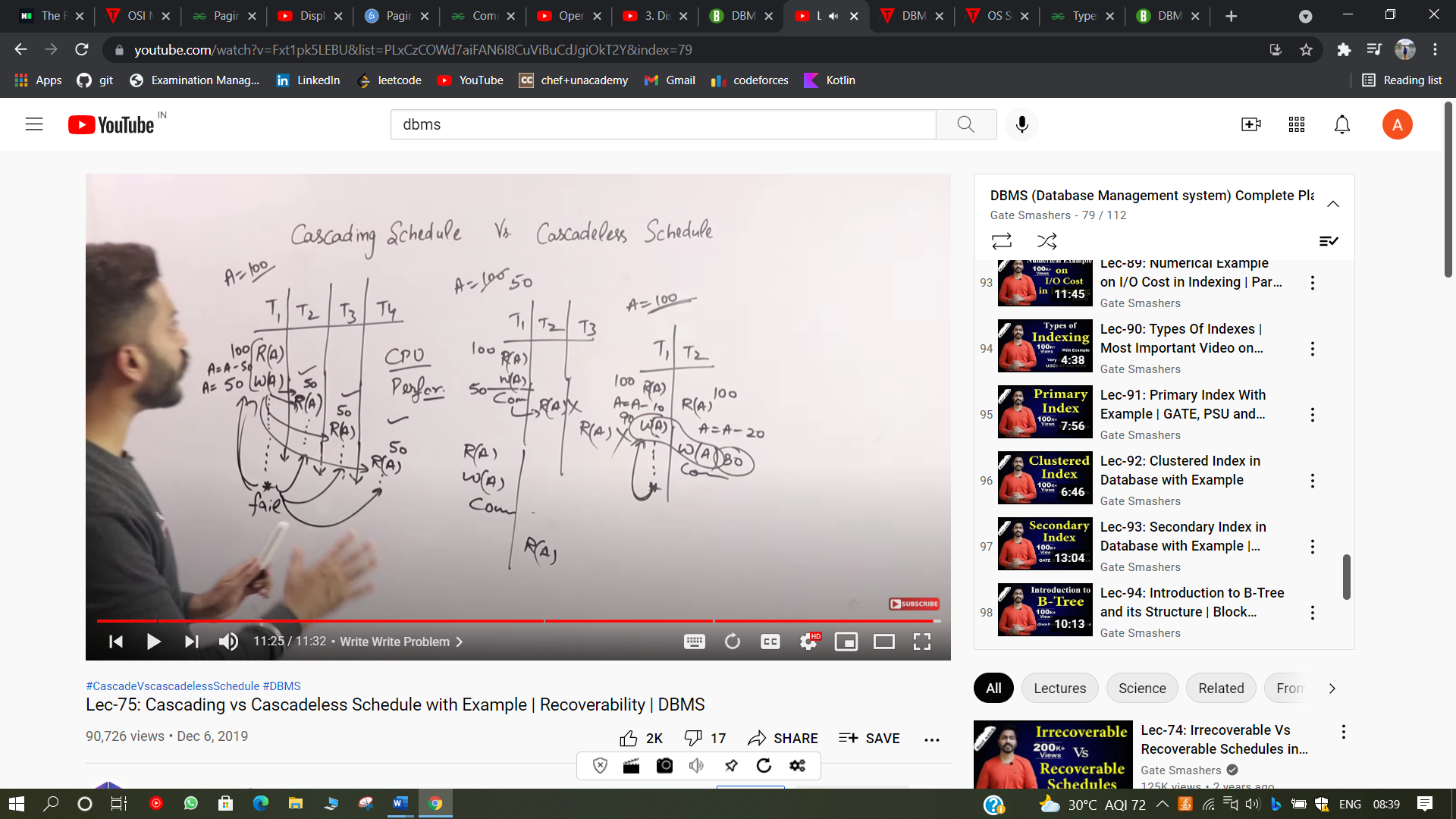
**Types of Recoverable Schedules –**

**● 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.

**Disadvantage -** > reduce cpu performance

**● Cascadeless Schedule** In Cascadeless Schedule, if a transaction is going to perform read operation on a value, it has to wait until the transaction who is performing write on that value commits.

**Disadvantage -> write write problem**



**● 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.

**File Structures:**

● **Primary Index:** A primary index is an ordered file, records of fixed length with two fields. First field is the same as the primary key as a data file and the second field is a pointer to the data block, where the key is available. The average number of block accesses using index = **log2 Bi + 1**, where Bi = number of index blocks.

* ● **Clustering Index(parse)** A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.
* In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
* The records which have similar characteristics are grouped, and indexes are created for these group.

● **Secondary Index:** Secondary index provides secondary means of accessing a file for which primary access already exists.

**B Trees**

At every level , we have Key and Data Pointer and data pointer points to either block or record.

**Properties of B-Trees:**

Root of B-tree can have children between **2** and **P**, where P is Order of tree. **Order of tree** – Maximum number of children a node can have.

Internal node can have children between ⌈ **P/2** ⌉ and **P**

Internal node can have keys between ⌈ **P/2** ⌉ **– 1** and **P-1**

**B+ Trees**

In B+ trees, the structure of leaf and non-leaf are different, so their order is. Order of non-leaf will be higher as compared to leaf nodes.

Searching time will be less in B+ trees, since it doesn’t have record pointers in non-leaf because of which depth will decrease.

**What is horizontal scaling and vertical scaling** : <https://www.geeksforgeeks.org/horizontal-and-vertical-scaling-in-databases/>

Sharding <https://www.geeksforgeeks.org/what-is-sharding/>

<https://nlogn.in/what-is-database-sharding-and-how-is-it-done/>

union and union all <https://www.javatpoint.com/mysql-union-vs-union-all>

**Relational Algebra:**

Relational Algebra is a procedural query language which takes a relation as an input and generates a relation as an output.

|  |  |
| --- | --- |
| **Basic Operator** | **Semantic** |
| **σ(Selection)** | Select rows based on given condition |
| **∏(Projection)** | Project some columns |
| **X (Cross Product)** | Cross product of relations, returns **m\*n** rows where m and n are number of rows in R1 and R2 respectively. |
| **U (Union)** | Return those tuples which are either in R1 or in R2. Max no. of rows returned **= m+n** and Min no. of rows returned = **max(m,n)** |
| **−(Minus)** | R1-R2 returns those tuples which are in R1 but not in R2. Max no. of rows returned = **m** and Min no. of rows returned = **m-n** |
| **ρ(Rename)** | Renaming a relation to another relation. |

|  |  |
| --- | --- |
| **Extended Operator** | **Semantic** |
| **∩ (Intersection)** | Returns those tuples which are in both R1 and R2. Max no. of rows returned = min(m,n) and Min no. of rows returned = 0 |
| ⋈**c(Conditional Join)** | Selection from two or more tables based on some condition (Cross product followed by selection) |
| ⋈**(Equi Join)** | It is a special case of conditional join when only equality conditions are applied between attributes. |
| ⋈**(Natural Join)** | In natural join, equality conditions on common attributes hold and duplicate attributes are removed by default.  **Note:** Natural Join is equivalent to cross product if two relations have no attribute in common and natural join of a relation R with itself will return R only. |

|  |  |
| --- | --- |
| ⟕**(Left Outer Join)** | When applying join on two relations R and S, some tuples of R or S do not appear in the result set which does not satisfy the join conditions. But Left Outer Joins gives all tuples of R in the result set. The tuples of R which do not satisfy the join condition will have values as NULL for attributes of S. |
| ⟖**(Right Outer Join)** | When applying join on two relations R and S, some tuples of R or S do not appear in the result set which does not satisfy the join conditions. But Right Outer Joins gives all tuples of S in the result set. The tuples of S which do not satisfy the join condition will have values as NULL for attributes of R. |
| ⟗(Full Outer Join)  /(Division Operator) | When applying join on two relations R and S, some tuples of R or S do not appear in the result set which does not satisfy the join conditions. But Full Outer Joins gives all tuples of S and all tuples of R in the result set. The tuples of S which do not satisfy the join condition will have values as NULL for attributes of R and vice versa.  Division operator A/B will return those tuples in A which are associated with every tuple of B. **Note:** Attributes of B should be a proper subset of attributes of A. The attributes in A/B will be Attributes of A- Attribute of B. |
|  |  |

**SQL**

**DDL:**

DDL is short name of **Data Definition Language,** which deals with database schemas and descriptions, of how the data should reside in the database.

● CREATE - to create a database and its objects like (table, index, views, store procedure, function, and triggers)

● ALTER - alters the structure of the existing database

● DROP - delete objects from the database

● TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed

● RENAME - rename an object

**DML:**

DML is short name of **Data Manipulation Language** which deals with data manipulation and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE, etc., and it is used to store, modify, retrieve, delete and update data in a database.

● SELECT - retrieve data from a database

● INSERT - insert data into a table

● UPDATE - updates existing data within a table

● DELETE - Delete all records from a database table

● MERGE - UPSERT operation (insert or update)

**DCL:**

DCL is short name of **Data Control Language** which includes commands such as GRANT and mostly concerned with rights, permissions and other controls of the database system.

● GRANT - allow users access privileges to the database

● REVOKE - withdraw users access privileges given by using the GRANT command

**TCL:**

TCL is short name of Transaction Control Language which deals with a transaction within a database.

● COMMIT - a Transaction commits

● ROLLBACK - rollback a transaction in case of any error occurs

● SAVEPOINT - to roll back the transaction making points within groups **SQL:**

SQL is a standard language for storing, manipulating and retrieving data in databases.

**SELECT:**

The SELECT statement is used to select data from a database.

**Syntax -**

● SELECT *column1*, *column2, ...*

FROM *table\_name*;

● Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax: ● SELECT \* FROM *table\_name*;

**Ex –**

● SELECT CustomerName, City FROM Customers;

**SELECT DISTINCT:**

The SELECT DISTINCT statement is used to return only distinct (different) values. **Syntax –**

● SELECT DISTINCT *column1*, *column2, ...*

FROM *table\_name*;

**Ex –**

● SELECT DISTINCT Country FROM Customers;

**WHERE:**

The WHERE clause is used to filter records.

**Syntax –**

● SELECT *column1*, *column2, ...*

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT \* FROM Customers

WHERE Country='Mexico';

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| <> | Not equal. **Note:** In some versions of SQL this operator may be written as != |

**AND, OR and NOT:**

The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

● The AND operator displays a record if all the conditions separated by AND are TRUE. ● The OR operator displays a record if any of the conditions separated by OR is TRUE.

The NOT operator displays a record if the condition(s) is NOT TRUE.

**Syntax –**

● SELECT *column1*, *column2, ...*

FROM *table\_name*

WHERE *condition1* AND *condition2* AND *condition3 ...*;

● SELECT *column1*, *column2, ...*

FROM *table\_name*

WHERE *condition1* OR *condition2* OR *condition3 ...*;

● SELECT *column1*, *column2, ...*

FROM *table\_name*

WHERE NOT *condition*;

**Ex –**

● SELECT \* FROM Customers

WHERE Country='Germany' AND City='Berlin';

● SELECT \* FROM Customers

WHERE Country='Germany' AND (City='Berlin' OR City='München');

**ORDER BY:**

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

**Syntax –**

● SELECT *column1*, *column2, ...*

FROM *table\_name*

ORDER BY *column1, column2, ...* ASC|DESC;

**Ex –**

● SELECT \* FROM Customers

ORDER BY Country;

● SELECT \* FROM Customers

ORDER BY Country ASC, CustomerName DESC;

**INSERT INTO:**

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

**Syntax –**

● INSERT INTO *table\_name* (*column1*, *column2*, *column3*, ...)

VALUES (*value1*, *value2*, *value3*, ...);

● INSERT INTO *table\_name*

VALUES (*value1*, *value2*, *value3*, ...);

\*In the second syntax, make sure the order of the values is in the same order as the columns in the table.

**Ex –**

● INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)

VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

**NULL Value:**

It is not possible to test for NULL values with comparison operators, such as =, <, or <>. We will have to use the IS NULL and IS NOT NULL operators instead.

**Syntax –**

● SELECT *column\_names*

FROM *table\_name*

WHERE *column\_name* IS NULL;

● SELECT *column\_names*

FROM *table\_name*

WHERE *column\_name* IS NOT NULL;

**Ex –**

● SELECT CustomerName, ContactName, Address

FROM Customers

WHERE Address IS NULL;

**UPDATE:**

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

● UPDATE *table\_name*

SET *column1* = *value1*, *column2* = *value2*, ...

WHERE *condition*;

**Ex –**

● UPDATE Customers

SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'

WHERE CustomerID = 1;

**DELETE:**

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

**Syntax –**

● DELETE FROM *table\_name* WHERE *condition*;

● DELETE FROM *table\_name*;

In 2ndsyntax, all rows are deleted. The table structure, attributes, and indexes will be intact **Ex –**

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

**SELECT TOP:**

The SELECT TOP clause is used to specify the number of records to return. **Syntax –**

● SELECT TOP *number*|*percent column\_name(s)*

FROM *table\_name*

WHERE *condition*;

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE *condition*

LIMIT *number*;

● SELECT *column\_name(s)*

FROM *table\_name*

ORDER BY *column\_name(s)*

FETCH FIRST *number* ROWS ONLY;

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE ROWNUM <= *number*;

\*In case the interviewer asks other than the TOP, rest are also correct. (Diff. DB Systems) **Ex –**

● SELECT TOP 3 \* FROM Customers;

● SELECT \* FROM Customers

LIMIT 3;

● SELECT \* FROM Customers

FETCH FIRST 3 ROWS ONLY;

**Aggregate Functions:**

**MIN():**

The MIN() function returns the smallest value of the selected column. **Syntax –**

● SELECT MIN(*column\_name*)

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT MIN(Price) AS SmallestPrice

FROM Products;

**MAX():**

The MAX() function returns the largest value of the selected column. **Syntax –**

● SELECT MAX(*column\_name*)

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT MAX(Price) AS LargestPrice

FROM Products;

**COUNT():**

The COUNT() function returns the number of rows that matches a specified criterion. **Syntax –**

● SELECT COUNT(*column\_name*)

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT COUNT(ProductID)

FROM Products;

**AVG():**

The AVG() function returns the average value of a numeric column.

**Syntax –**

● SELECT AVG(*column\_name*)

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT AVG(Price)

FROM Products;

**SUM():**

The SUM() function returns the total sum of a numeric column.

**Syntax –**

● SELECT SUM(*column\_name*)

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT SUM(Quantity)

FROM OrderDetails;

**LIKE Operator:**

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. There are two wildcards often used in conjunction with the LIKE operator:

● The percent sign (%) represents zero, one, or multiple characters

● The underscore sign (\_) represents one, single character

**Syntax –**

● SELECT *column1, column2, ...*

FROM *table\_name*

WHERE *columnN* LIKE *pattern*;

|  |  |
| --- | --- |
| **LIKE Operator** | **Description** |
| WHERE CustomerName LIKE 'a%' | Finds any values that start with "a" |
| WHERE CustomerName LIKE '%a' | Finds any values that end with "a" |
| WHERE CustomerName LIKE '%or%' | Finds any values that have "or" in any position |

|  |  |
| --- | --- |
| WHERE CustomerName LIKE '\_r%' | Finds any values that have "r" in the second position |
| WHERE CustomerName LIKE 'a\_%' | Finds any values that start with "a" and are at least 2 characters in length |
| WHERE CustomerName LIKE 'a\_ \_%' | Finds any values that start with "a" and are at least 3 characters in length |
| WHERE ContactName LIKE 'a%o' | Finds any values that start with "a" and ends with "o" |

LIKE is case sensitive, ILIKE is case insensitive.

**IN:**

The IN operator allows you to specify multiple values in a WHERE clause. The NI operator is a shorthand for multiple OR conditions.

**Syntax –**

● SELECT *column\_name(s)*

FROM  *\_name*

WHERE *column\_name* IN (*value1*, *value2*, ...);

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE *column\_name* IN (*SELECT STATEMENT*);

**Ex –**

● SELECT \* FROM Customers

WHERE Country IN ('Germany', 'France', 'UK');

● SELECT \* FROM Customers

WHERE Country IN (SELECT Country FROM Suppliers);

**BETWEEN:**

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

**Syntax –**

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE *column\_name* BETWEEN *value1* AND *value2;*

**Ex –**

● SELECT \* FROM Products

WHERE Price BETWEEN 10 AND 20;

**Joins:**

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

NATURAL JOIN:

Natural Join joins two tables based on same attribute name and datatypes. The resulting table will contain all the attributes of both the table but keep only one copy of each common column.

SYNTAX:

SELECT \*

FROM table1 NATURAL JOIN table2;

**INNER JOIN:**

Inner Join joins two table on the basis of the column which is explicitly specified in the ON clause. The resulting table will contain all the attributes from both the tables including common column also **Syntax –**

● SELECT *column\_name(s)*

FROM *table1*

INNER JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*;

**Ex –**

● SELECT Orders.OrderID, Customers.CustomerName

FROM Orders

INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

## JOIN Three Tables

SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName

FROM ((Orders

INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)

INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);

**LEFT (OUTER) JOIN:**

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

**Syntax –**

● SELECT *column\_name(s)*

FROM *table1*

LEFT JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*;

**Ex –**

● SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID

ORDER BY Customers.CustomerName;

**RIGHT (OUTER) JOIN:**

The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

**Syntax –**

● SELECT *column\_name(s)*

FROM *table1*

RIGHT JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*;

**Ex –**

● SELECT Orders.OrderID, Employees.LastName, Employees.FirstName FROM Orders

RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID ORDER BY Orders.OrderID;

**FULL (OUTER) JOIN:**

The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.

**Syntax:**

● SELECT *column\_name(s)*

FROM *table1*

FULL OUTER JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*

WHERE *condition*;

**Ex –**

● SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID ORDER BY Customers.CustomerName;

**SELF JOIN**: As the name signifies, in SELF JOIN a table is joined to itself. That is, each row of the table is joined with itself and all other rows depending on some conditions. In other words we can say that it is a join between two copies of the same table.**Syntax:**

SELECT a.coulmn1 , b.column2

FROM table\_name a, table\_name b

WHERE some\_condition;

**table\_name**: Name of the table.

**some\_condition**: Condition for selecting the rows.

**CARTESIAN JOIN: The CARTESIAN JOIN is also known as CROSS JOIN. In a CARTESIAN JOIN there is a join for each row of one table to every row of another tabl**

**SELECT table1.column1 , table1.column2, table2.column1...**

**FROM table1**

**CROSS JOIN table2;**

**UNION:**

The UNION operator is used to combine the result-set of two or more SELECT statements.

● Every SELECT statement within UNION must have the same number of columns ● The columns must also have similar data types

● The columns in every SELECT statement must also be in the same order

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL

**Syntax –**

● SELECT *column\_name(s)* FROM *table1*

UNION

SELECT *column\_name(s)* FROM *table2*;

● SELECT *column\_name(s)* FROM *table1*

UNION ALL

SELECT *column\_name(s)* FROM *table2*;

**Ex –**

● SELECT City FROM Customers

UNION

SELECT City FROM Suppliers

ORDER BY City;

**GROUP BY:**

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions

(COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns. **Syntax –**

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE *condition*

GROUP BY *column\_name(s)*

ORDER BY *column\_name(s);*

**Ex –**

● SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country

ORDER BY COUNT(CustomerID) DESC;

* **all non-aggregated columns must appear in the GROUP BY clause**.
* select name, phone\_number , address , max(salary) as max\_salary from Employee\_Table group by dept\_name; is it right or wrong
* Your SQL query is **❌ incorrect** — and here's why:
*  You're **selecting columns (name, phone\_number, address)** that are **not part of the GROUP BY** clause.

### Correct Approaches:

#### ✅ ****Option 1: Include all non-aggregated columns in**** GROUP BY

This works but doesn't make logical sense in this case, since MAX(salary) is per department, not per name.

SELECT name, phone\_number, address, dept\_name, MAX(salary) AS max\_salary

FROM Employee\_Table

GROUP BY name, phone\_number, address, dept\_name;

But this returns **row-level max per person**, **not per department**.

**2: Get the employee(s) with the maximum salary per department**

This is likely what you're trying to do:

SELECT name, phone\_number, address, salary AS max\_salary, dept\_name

FROM Employee\_Table e

WHERE salary = (

SELECT MAX(salary)

FROM Employee\_Table

WHERE dept\_name = e.dept\_name

);

**HAVING:**

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

\*WHERE is given priority over HAVING.

**Syntax –**

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE *condition*

GROUP BY *column\_name(s)*

HAVING *condition*

ORDER BY *column\_name(s);*

**Ex –**

● SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country

HAVING COUNT(CustomerID) > 5;

<https://www.geeksforgeeks.org/having-vs-where-clause-in-sql/>

**How to print duplicate rows in a table?**   
See <https://www.geeksforgeeks.org/how-to-print-duplicate-rows-in-a-table/>

**CREATE DATABASE:**

The CREATE DATABASE statement is used to create a new SQL database. **Syntax –**

● CREATE DATABASE *databasename*;

**DROP DATABASE:**

The DROP DATABASE statement is used to drop an existing SQL database. **Syntax –**

● DROP DATABASE *databasename*;

**CREATE TABLE:**

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*,

....

);

**DROP TABLE:**

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

● DROP TABLE *table\_name*;

**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*;

**ALTER TABLE:**

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.

**Syntax –**

● ALTER TABLE *table\_name*

ADD *column\_name datatype*;

● ALTER TABLE *table\_name*

DROP COLUMN *column\_name*;

● ALTER TABLE *table\_name*

MODIFY COLUMN *column\_name datatype*;

**Ex –**

● ALTER TABLE Customers

ADD Email varchar(255);

● ALTER TABLE Customers

DROP COLUMN Email;

● ALTER TABLE Persons

MODIFY COLUMN DateOfBirth year;

Store procedure : <https://www.w3schools.com/sql/sql_stored_procedures.asp>

**Show 1st row of the table**:

Select \* from (Select rownum r, empSalary from EMP) where r =1;

**Show last row of the table**:

Select \*from (select rownum r, empSalary from EMP) where r = (select count(\*) from EMP);

**Show last 2 row of the table**:

Select \* from EMP minus select \* from Emp where rownum <= (select count(\*)-2 from EMP)

Select \*from (select rownum r, emp.\* from emp) where r>(select count(\*)-2 from EMP)

Even number of row/alternate record

Select \* from (select rownum r , emp.\* from EMP) where mod(r,2) =0;

Odd number of row

Select \* from (select rownum r , emp.\* from EMP) where mod(r,2) =1;

**SQL INTERVIEW QUESTION**

# Display nth row in SQL (Top 50 SQL Questions)| GeeksforGeeks

[**https://www.youtube.com/watch?v=k7rg4CARxD8&list=PLqM7alHXFySGweLxxAdBDK1CcDEgF-Kwx&index=7**](https://www.youtube.com/watch?v=k7rg4CARxD8&list=PLqM7alHXFySGweLxxAdBDK1CcDEgF-Kwx&index=7)

Trigger <https://www.geeksforgeeks.org/sql-trigger-student-database/>

**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.\

1. **CREATE TRIGGER Trigger\_Name**
2. **[ BEFORE | AFTER ] [ Insert | Update | Delete]**
3. **ON [Table\_Name]**
4. **[ FOR EACH ROW | FOR EACH COLUMN ]**
5. **AS**
6. **Set of SQL Statement**

**CREATE** **TRIGGER** Student\_Table\_Marks

1. BEFORE **INSERT**
2. **ON**
3. Student\_Trigger
4. **FOR** EACH ROW
5. **SET** new.Student\_TotalMarks = new.Student\_EnglishMarks + new.Student\_PhysicsMarks + new.Student\_ChemistryMarks + new.Student\_MathsMarks,
6. new.Student\_Percentage = ( new.Student\_TotalMarks / 400) \* 100;

<https://www.geeksforgeeks.org/find-second-largest-element-array/>.

### . Write a query to get the current date.

You can write a query as follows in [MySQL](https://www.edureka.co/blog/mysql-tutorial/):

|  |  |
| --- | --- |
| 1 | SELECT SYSTDATE(); |

### Write a query to create a new table which consists of data and structure copied from the other table.

CREATE TABLE NewTable AS SELECT \* FROM EmployeeInfo;

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Description** | **Example** |
| % | Represents zero or more characters | bl% finds bl, black, blue, and blob |
| \_ | Represents a single character | h\_t finds hot, hat, and hit |
| [] | Represents any single character within the brackets | h[oa]t finds hot and hat, but not hit |
| ^ | Represents any character not in the brackets | h[^oa]t finds hit, but not hot and hat |
| - | Represents any single character within the specified range | c[a-b]t finds cat and cbt |

Difference between Minus and – operator in Power Bi

| **Feature** | **MINUS** | **- (Arithmetic Minus)** |
| --- | --- | --- |
| **Type** | **Set operation** | **Arithmetic operator** |
| **Works on** | **Two queries (result sets)** | **Numbers or numeric expressions** |
| **Removes duplicates?** | **Yes (like DISTINCT)** | **No (just does math)** |
| **Supported in** | **Oracle** | **All SQL dialects** |

Got it! If you want to **round a number to a specific number of decimal places (scale)** in MySQL, you can use the built-in ROUND() function.

**✅ Syntax:**

sql

CopyEdit

ROUND(number, decimal\_places)

* number: The number you want to round.
* decimal\_places: How many decimal places (scale) you want to round to.

**🔍 Examples:**

sql

CopyEdit

SELECT ROUND(123.4567, 2); -- Output: 123.46

SELECT ROUND(123.4567, 0); -- Output: 123

SELECT ROUND(123.4567, 3); -- Output: 123.457

**📌 Note:**

* If you **omit** the second argument, MySQL will round to **0 decimal places**.

sql

CopyEdit

SELECT ROUND(123.4567); -- Output: 123

Common ways to get the **last character**

SELECT RIGHT(CITY, 1) AS LastChar

FROM STATION;

Case statement :

The CASE statement in SQL is like a switch-case or if-else ladder — it's used to evaluate conditions and return values accordingly. Let's break it down step by step:

SELECT

CASE

WHEN condition1 THEN result1

WHEN condition2 THEN result2

...

ELSE resultN

END AS alias\_name

FROM table\_name;

SELECT

Marks,

CASE

WHEN Marks >= 90 THEN 'A'

WHEN Marks >= 75 THEN 'B'

WHEN Marks >= 60 THEN 'C'

ELSE 'Fail'

END AS Grade

FROM Students;

**What is SQL Subquery?**

In SQL, a subquery can be defined as a query **embedded**within**another query**. It is often used in the **WHERE**, **HAVING**, or **FROM** clauses of a statement. Subqueries are commonly used with **SELECT**, **UPDATE**, INSERT, and **DELETE** statements to achieve **complex filtering** and **data manipulation**. They are an essential tool when we need to perform operations like:

**Types of Subqueries**

1. **Single-Row Subquery**: Returns a single value (row). Useful with comparison operators like =, >, <.
2. **Multi-Row Subquery**: Returns multiple values (rows). Useful with operators like IN, ANY, ALL.
3. **Correlated Subquery**: Refers to columns from the outer query in the subquery. Unlike regular subqueries, the subquery depends on the outer query for its values.
4. **Non-Correlated Subquery**: Does not refer to the outer query and can be executed independently.

A **correlated subquery** is a subquery that **depends on the outer query** for its values. Unlike a regular subquery (which can be executed independently), a correlated subquery is executed **once for each row** processed by the outer query

**Key Characteristics:**

* It references **columns from the outer query**.
* It is evaluated **row by row**.
* Typically used in WHERE, SELECT, or HAVING clauses.

### Performance Consideration:

* Since correlated subqueries execute **once per row**, they can be slow on large datasets.
* In many cases, it's better to use **JOINs** or **common table expressions (CTEs)** for optimization.

### When to Use Correlated Subqueries:

* When you need **row-by-row comparisons**.
* When values from the outer query are needed inside the subquery.

**In SQL, both RANK() and DENSE\_RANK() are window functions used to assign a rank to rows within a partition of a result set, based on the order of a specified column. However, they handle ties differently.**

**1. RANK()**

* **Gives the same rank to tied rows, but skips the next ranks.**
* **Think of it as "gapped ranking."**

| **Score** | **RANK()** |
| --- | --- |
| **100** | **1** |
| **100** | **1** |
| **90** | **3** |
| **80** | **4** |

***Explanation*: Two rows tie at rank 1, so the next rank is 3 (it skips 2).**

**. DENSE\_RANK()**

* **Gives the same rank to tied rows, but does NOT skip the next rank.**
* **Think of it as "compact ranking."**

| **Score** | **DENSE\_RANK()** |
| --- | --- |
| **100** | **1** |
| **100** | **1** |
| **90** | **2** |
| **80** | **3** |

***Explanation*: No gaps in the ranking after a tie.**

**SELECT**

**Name,**

**Score,**

**RANK() OVER (ORDER BY Score DESC) AS Rank,**

**DENSE\_RANK() OVER (ORDER BY Score DESC) AS DenseRank**

**FROM**

**Students;**