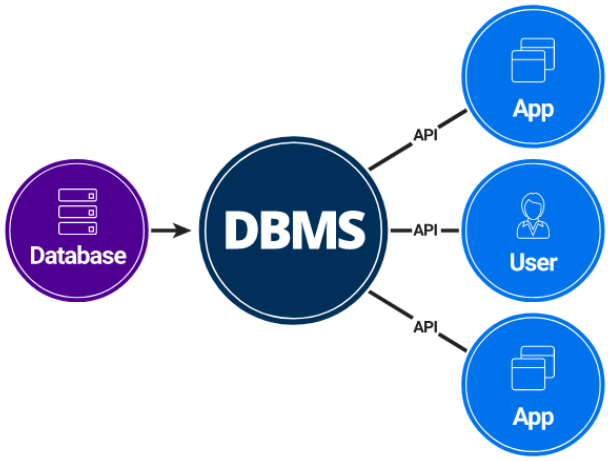
LEC-1: Introduction to DBMS

1. **What is Data?**
    a. Data is a collection of raw, unorganized facts and details like text, observations, figures, symbols, and descriptions of things etc.
    In other words, **data does not carry any specific purpose and has no significance by itself.**
    Moreover, data is measured in terms of bits and bytes – which are basic units of information in the context of computer storage and processing.
    b. Data can be recorded and doesn't have any meaning unless processed.
2. **Types of Data**
    a. Quantitative
        i. Numerical form
        ii. Weight, volume, cost of an item.
    b. Qualitative
        i. Descriptive, but not numerical.
        ii. Name, gender, hair color of a person.
3. What is **Information**?
    a. Info. Is **processed, organized, and structured data.**
    b. It provides **context of the data and enables decision making.**
    c. Processed data that make **sense** to us.
    d. Information is extracted from the data, by **analyzing and interpreting** pieces of data.
    e. E.g., you have data of all the people living in your locality, its Data, when you analyze and interpret the data and come to some conclusion that:
        i. There are 100 senior citizens.
        ii. The sex ratio is 1.1.
        iii. Newborn babies are 100.
        These are information.
4. **Data vs Information**
    a. Data is a collection of facts, while information puts those facts into context.
    b. While data is raw and unorganized, information is organized.
    c. Data points are individual and sometimes unrelated. Information maps out that data to provide a big-picture view of how it all fits together.
    d. Data, on its own, is meaningless. When it's analyzed and interpreted, it becomes meaningful information.
    e. Data does not depend on information; however, information depends on data.
    f. Data typically comes in the form of graphs, numbers, figures, or statistics. Information is typically presented through words, language, thoughts, and ideas.
    g. Data isn't sufficient for **decision-making**, but you can make decisions based on information.
5. What is **Database**?
    a. Database is an electronic place/system where data is stored in a way that it can be **easily accessed, managed, and updated.**
    b. To make real use Data, we need **Database management systems. (DBMS)**
6. What is **DBMS**?
    a. A database-management system (DBMS) is a collection of **interrelated data** and **a set of programs to access those data**. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to **store and retrieve database information** that is both convenient and efficient.
    b. A DBMS is the database itself, along with all the software and functionality. It is used to perform different operations, like **addition**, **access**, **updating**, and **deletion** of the data.

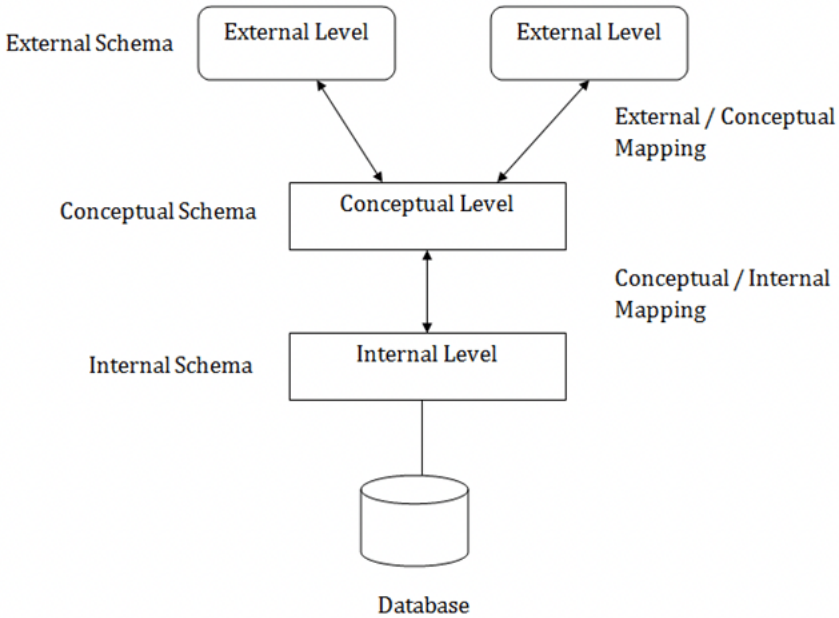


7.

8. **DBMS vs File Systems**
    a. **File-processing systems** has major **disadvantages**.
        i. Data Redundancy and inconsistency
        ii. Difficulty in accessing data
        iii. Data isolation
        iv. Integrity problems
        v. Atomicity problems
        vi. Concurrent-access anomalies
        vii. Security problems
    b. Above 7 are also the **Advantages of DBMS** (answer to "**Why to use DBMS?**")

LEC-2: DBMS Architecture

1. **View of Data (Three Schema Architecture)**
    a. The major purpose of DBMS is to provide users with an **abstract view** of the data. That is, the system hides certain details of how the data is stored and maintained.
    b. To simplify user interaction with the system, abstraction is applied through several levels of abstraction.
    c. The **main objective** of three level architecture is to enable multiple users to access the same data with a personalized view while storing the underlying data only once
    d. Physical level / Internal level
        i. The lowest level of abstraction describes how the data are stored.
        ii. Low-level data structures used.
        iii. It has **Physical schema** which describes physical storage structure of DB.
        iv. Talks about: Storage allocation (N-ary tree etc), Data compression & encryption etc.
        v. **Goal**: We must define algorithms that allow efficient access to data.
    e. Logical level / Conceptual level:
        i. The **conceptual schema** describes the design of a database at the conceptual level, describes ***what*** data are stored in DB, and what ***relationships*** exist among those data.
        ii. User at logical level does not need to be aware about physical-level structures.
        iii. **DBA**, who must decide what information to keep in the DB use the logical level of abstraction.
        iv. **Goal**: ease to use.
    f. View level / External level:
        i. Highest level of abstraction aims to simplify users' interaction with the system by providing different view to different **end**-user.
        ii. Each **view schema** describes the database part that a particular user group is interested and hides the remaining database from that user group.
        iii. At the external level, a database contains several schemas that sometimes called as **subschema**. The subschema is used to describe the different view of the database.
        iv. At views also provide a **security** mechanism to prevent users from accessing certain parts of DB.
    g.



2. **Instances and Schemas**
    a. The collection of information stored in the DB at a particular moment is called an **instance** of DB.

b. The overall design of the DB is called the DB **schema**.
c. Schema is **structural** description of data. Schema **doesn't change frequently**. Data may change frequently.
d. **DB schema** corresponds to the variable declarations (along with type) in a program.
e. We have 3 types of **Schemas**: **Physical, Logical**, several **view schemas** called subschemas.
f. Logical schema is most **important** in terms of its effect on application programs, as programmers construct apps by using logical schema.
g. **Physical data independence**, physical schema change should not affect logical schema/application programs.

3. Data Models:

a. Provides a way to describe the **design** of a DB at **logical level.**
b. Underlying the structure of the DB is the Data Model; a collection of conceptual tools for describing **data, data relationships, data semantics & consistency constraints**.
c. E.g., **ER** model, **Relational** Model, **object-oriented** model, **object-relational** data model etc.

4. **Database Languages**:

a. **Data definition language (DDL)** to specify the database schema.
b. **Data manipulation language (DML)** to express database queries and updates.
c. **Practically**, both language features are present in a single DB language, e.g., SQL language.
d. DDL
    i. We specify consistency constraints, which must be checked, every time DB is updated.
e. DML
    i. Data manipulation involves
        1. **Retrieval** of information stored in DB.
        2. **Insertion** of new information into DB.
        3. **Deletion** of information from the DB.
        4. **Updating** existing information stored in DB.
    ii. **Query language**, a part of DML to specify statement requesting the retrieval of information.

5. How is Database accessed from Application programs?

a. Apps (written in host languages, C/C++, Java) interacts with DB.
b. E.g., Banking system's module generating payrolls access DB by executing DML statements from the host language.
c. API is provided to send DML/DDL statements to DB and retrieve the results.
    i. Open Database Connectivity (**ODBC**), Microsoft "*C*".
    ii. Java Database Connectivity (**JDBC**), Java.

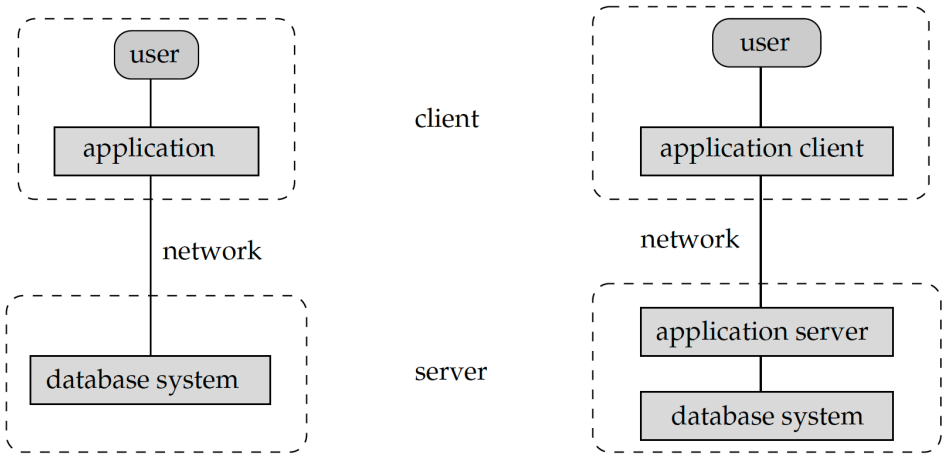
6. Database Administrator (DBA)

a. A person who has **central control** of both the data and the programs that access those data.
b. **Functions** of DBA
    i. Schema Definition
    ii. Storage structure and access methods.
    iii. Schema and physical organization modifications.
    iv. Authorization control.
    v. Routine maintenance
        1. Periodic backups.
        2. Security patches.
        3. Any upgrades.

7. **DBMS Application Architectures**: Client machines, on which remote DB users work, and server machines on which DB system runs.

a. **T1** Architecture
    i. The client, server & DB all present on the same machine.

b. **T2** Architecture
    i. App is partitioned into 2-components.
    ii. Client machine, which invokes DB system functionality at server end through query language statements.
    iii. API standards like **ODBC** & **JDBC** are used to interact between client and server.
c. **T3** Architecture
    i. App is partitioned into 3 logical components.
    ii. Client machine is just a frontend and doesn't contain any direct DB calls.
    iii. Client machine communicates with App server, and App server communicated with DB system to access data.
    iv. **Business** logic, what action to take at that condition is in App server itself.
    v. T3 architecture are best for **WWW** Applications.
    vi. **Advantages**:
        1. **Scalability** due to distributed application servers.
        2. **Data integrity**, App server acts as a middle layer between client and DB, which minimize the chances of data corruption.
        3. **Security**, client can't directly access DB, hence it is more secure.



a. two-tier architecture

b. three-tier architecture

LEC-9: SQL in 1-Video

1. **SQL**: Structured Query Language, used to access and manipulate data.
2. SQL used **CRUD** operations to communicate with DB.
    1. **CREATE** - execute INSERT statements to insert new tuple into the relation.
    2. **READ** - Read data already in the relations.
    3. **UPDATE** - Modify already inserted data in the relation.
    4. **DELETE** - Delete specific data point/tuple/row or multiple rows.
3. **SQL is not DB, is a query language.**
4. What is **RDBMS**? (Relational Database Management System)
    1. Software that enable us to implement designed relational model.
    2. e.g., MySQL, MS SQL, Oracle, IBM etc.
    3. Table/Relation is the simplest form of data storage object in R-DB.
    4. **MySQL** is open-source RDBMS, and it uses SQL for all CRUD operations
5. **MySQL** used client-server model, where client is CLI or frontend that used services provided by MySQL server.
6. **Difference between SQL and MySQL**
    1. SQL is Structured Query language used to perform CRUD operations in R-DB, while MySQL is a RDBMS used to store, manage and administrate DB (provided by itself) using SQL.

SQL DATA TYPES (Ref: https://www.w3schools.com/sql/sql_datatypes.asp)

1. In SQL DB, data is stored in the form of tables.
2. Data can be of different types, like INT, CHAR etc.

DATATYPE	Description
CHAR	string(0-255), string with size = (0, 255], e.g., CHAR(251)
VARCHAR	string(0-255)
TINYTEXT	String(0-255)
TEXT	string(0-65535)
BLOB	string(0-65535)
MEDIUMTEXT	string(0-16777215)
MEDIUMBLOB	string(0-16777215)
LONGTEXT	string(0-4294967295)
LONGBLOB	string(0-4294967295)
TINYINT	integer(-128 to 127)
SMALLINT	integer(-32768 to 32767)
MEDIUMINT	integer(-8388608 to 8388607)
INT	integer(-2147483648 to 2147483647)
BIGINT	integer (-9223372036854775808 to 9223372036854775807)
FLOAT	Decimal with precision to 23 digits
DOUBLE	Decimal with 24 to 53 digits

DATATYPE	Description
DECIMAL	Double stored as string
DATE	YYYY-MM-DD
DATETIME	YYYY-MM-DD HH:MM:SS
TIMESTAMP	YYYYMMDDHHMMSS
TIME	HH:MM:SS
ENUM	One of the preset values
SET	One or many of the preset values
BOOLEAN	0/1
BIT	e.g., BIT(n), n upto 64, store values in bits.

3. Size: TINY < SMALL < MEDIUM < INT < BIGINT.
4. **Variable length Data types** e.g., VARCHAR, are better to use as they occupy space equal to the actual data size.
5. Values can also be unsigned e.g., INT UNSIGNED.
6. **Types of SQL commands:**
    1. **DDL** (data definition language): defining relation schema.
        1. **CREATE**: create table, DB, view.
        2. **ALTER TABLE**: modification in table structure. e.g, change column datatype or add/remove columns.
        3. **DROP**: delete table, DB, view.
        4. **TRUNCATE**: remove all the tuples from the table.
        5. **RENAME**: rename DB name, table name, column name etc.
    2. **DRL/DQL** (data retrieval language / data query language): retrieve data from the tables.
        1. **SELECT**
    3. **DML** (data modification language): use to perform modifications in the DB
        1. **INSERT**: insert data into a relation
        2. **UPDATE**: update relation data.
        3. **DELETE**: delete row(s) from the relation.
    4. **DCL** (Data Control language): grant or revoke authorities from user.
        1. **GRANT**: access privileges to the DB
        2. **REVOKE**: revoke user access privileges.
    5. **TCL** (Transaction control language): to manage transactions done in the DB
        1. **START TRANSACTION**: begin a transaction
        2. **COMMIT**: apply all the changes and end transaction
        3. **ROLLBACK**: discard changes and end transaction
        4. **SAVEPOINT**: checkout within the group of transactions in which to rollback.

MANAGING DB (DDL)

1. **Creation of DB**
    1. **CREATE DATABASE IF NOT EXISTS db-name;**
    2. **USE** db-name; //need to execute to choose on which DB CREATE TABLE etc commands will be executed. //make switching between DBs possible.
    3. **DROP** DATABASE IF EXISTS db-name; //dropping database.
    4. **SHOW** DATABASES; //list all the DBs in the server.
    5. **SHOW** TABLES; //list tables in the selected DB.

DATA RETRIEVAL LANGUAGE (DRL)

1. Syntax: SELECT <set of column names> FROM <table_name>;
2. Order of execution from RIGHT to LEFT.
3. Q. Can we use SELECT keyword without using FROM clause?
    1. Yes, using DUAL Tables.
    2. Dual tables are dummy tables created by MySQL, help users to do certain obvious actions without referring to user defined tables.
    3. e.g., SELECT 55 + 11;
       SELECT now();
       SELECT ucase(); etc.
4. **WHERE**
    1. Reduce rows based on given conditions.
    2. E.g., SELECT * FROM customer WHERE age > 18;
5. **BETWEEN**
    1. SELECT * FROM customer WHERE age between 0 **AND** 100;
    2. In the above e.g., 0 and 100 are inclusive.
6. **IN**
    1. Reduces **OR** conditions;
    2. e.g., SELECT * FROM officers WHERE officer_name IN ('Lakshay', 'Maharana Pratap', 'Deepika');
7. **AND/OR/NOT**
    1. AND: WHERE cond1 AND cond2
    2. OR: WHERE cond1 OR cond2
    3. NOT: WHERE col_name NOT IN (1,2,3,4);
8. **IS NULL**
    1. e.g., SELECT * FROM customer WHERE prime_status is NULL;
9. **Pattern Searching / Wildcard ('%', '_')**
    1. '%', any number of character from 0 to n. Similar to '*' asterisk in regex.
    2. '_', only one character.
    3. SELECT * FROM customer WHERE name **LIKE** '%p_';
10. **ORDER BY**
    1. Sorting the data retrieved using **WHERE** clause.
    2. ORDER BY <column-name> DESC;
    3. DESC = Descending and ASC = Ascending
    4. e.g., SELECT * FROM customer ORDER BY name DESC;
11. **GROUP BY**
    1. GROUP BY Clause is used to collect data from multiple records and group the result by one or more column. It is generally used in a SELECT statement.
    2. Groups into category based on column given.
    3. SELECT c1, c2, c3 FROM sample_table WHERE cond GROUP BY c1, c2, c3.
    4. All the column names mentioned after SELECT statement shall be repeated in GROUP BY, in order to successfully execute the query.
    5. Used with aggregation functions to perform various actions.
        1. COUNT()
        2. SUM()
        3. AVG()
        4. MIN()
        5. MAX()
12. **DISTINCT**
    1. Find distinct values in the table.
    2. SELECT DISTINCT(col_name) FROM table_name;
    3. GROUP BY can also be used for the same
        1. "Select col_name from table GROUP BY col_name;" same output as above DISTINCT query.

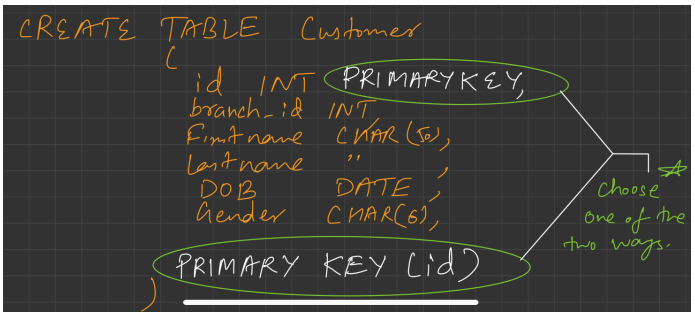
2. SQL is smart enough to realise that if you are using GROUP BY and not using any aggregation function, then you mean "DISTINCT".

13. GROUP BY HAVING

1. Out of the categories made by GROUP BY, we would like to know only particular thing (cond).
2. Similar to WHERE.
3. Select COUNT(cust_id), country from customer GROUP BY country HAVING COUNT(cust_id) > 50;
4. WHERE vs HAVING
    1. Both have same function of filtering the row base on certain conditions.
    2. WHERE clause is used to filter the rows from the table based on specified condition
    3. HAVING clause is used to filter the rows from the groups based on the specified condition.
    4. HAVING is used after GROUP BY while WHERE is used before GROUP BY clause.
    5. If you are using HAVING, GROUP BY is necessary.
    6. WHERE can be used with SELECT, UPDATE & DELETE keywords while GROUP BY used with SELECT.

CONSTRAINTS (DDL)

1. **Primary Key**



    1. PK is not null, unique and only one per table.
2. **Foreign Key**
    1. FK refers to PK of other table.
    2. Each relation can having any number of FK.
    3. CREATE TABLE ORDER (
    id INT PRIMARY KEY,
    delivery_date DATE,
    order_placed_date DATE,
    cust_id INT,
    FOREIGN KEY (cust_id) REFERENCES customer(id)
    );
3. **UNIQUE**
    1. Unique, can be null, table can have multiple unique attributes.
    2. CREATE TABLE customer (
    ...
    email VARCHAR(1024) UNIQUE,
    ...
    );
4. **CHECK**
    1. CREATE TABLE customer (
    ...
    CONSTRAINT age_check CHECK (age > 12),
    ...
    );
    2. "age_check", can also avoid this, MySQL generates name of constraint automatically.

5. **DEFAULT**
    1. Set default value of the column.
    2. CREATE TABLE account (
       …
       saving-rate DOUBLE NOT NULL DEFAULT 4.25,
       …
       );
6. An attribute can be **PK and FK both** in a table.
7. **ALTER OPERATIONS**
    1. Changes schema
    2. **ADD**
        1. **Add new column.**
        2. ALTER TABLE table_name ADD new_col_name datatype ADD new_col_name_2 datatype;
        3. e.g., ALTER TABLE customer ADD age INT NOT NULL;
    3. **MODIFY**
        1. **Change datatype of an attribute.**
        2. ALTER TABLE table-name MODIFY col-name col-datatype;
        3. E.g., VARCHAR TO CHAR
           ALTER TABLE customer MODIFY name CHAR(1024);
    4. **CHANGE COLUMN**
        1. **Rename column name.**
        2. ALTER TABLE table-name CHANGE COLUMN old-col-name new-col-name new-col-datatype;
        3. e.g., ALTER TABLE customer CHANGE COLUMN name customer-name VARCHAR(1024);
    5. **DROP COLUMN**
        1. **Drop a column completely.**
        2. ALTER TABLE table-name DROP COLUMN col-name;
        3. e.g., ALTER TABLE customer DROP COLUMN middle-name;
    6. **RENAME**
        1. **Rename table name itself.**
        2. ALTER TABLE table-name RENAME TO new-table-name;
        3. e.g., ALTER TABLE customer RENAME TO customer-details;

DATA MANIPULATION LANGUAGE (DML)

1. **INSERT**
    1. INSERT INTO table-name(col1, col2, col3) VALUES (v1, v2, v3), (val1, val2, val3);
2. **UPDATE**
    1. UPDATE table-name SET col1 = 1, col2 = 'abc' WHERE id = 1;
    2. Update multiple rows e.g.,
        1. UPDATE student SET standard = standard + 1;
    3. **ON UPDATE CASCADE**
        1. Can be added to the table while creating constraints. Suppose there is a situation where we have two tables such that primary key of one table is the foreign key for another table. if we update the primary key of the first table then using the ON UPDATE CASCADE foreign key of the second table automatically get updated.
3. **DELETE**
    1. DELETE FROM table-name WHERE id = 1;
    2. DELETE FROM table-name; //all rows will be deleted.
    3. **DELETE CASCADE** - (**to overcome DELETE constraint of Referential constraints**)
        1. What would happen to child entry if parent table's entry is deleted?
        2. CREATE TABLE ORDER (
           order_id int PRIMARY KEY,
           delivery_date DATE,
           cust_id INT,

FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE CASCADE
);

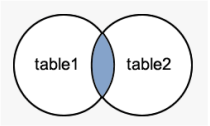
3. **ON DELETE NULL** - (**can FK have null values?**)
    1. CREATE TABLE ORDER (
    order_id int PRIMARY KEY,
    delivery_date DATE,
    cust_id INT,
    FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE SET NULL
    );

4. REPLACE

1. Primarily used for already present tuple in a table.
2. As UPDATE, using REPLACE with the help of WHERE clause in PK, then that row will be replaced.
3. As INSERT, if there is no duplicate data new tuple will be inserted.
4. REPLACE INTO student (id, class) VALUES(4, 3);
5. REPLACE INTO table SET col1 = val1, col2 = val2;

JOINING TABLES

1. All **RDBMS** are relational in nature, we refer to other tables to get meaningful outcomes.
2. FK are used to do reference to other table.
3. **INNER JOIN**
    1. Returns a resultant table that has matching values from both the tables or all the tables.
    2. SELECT column-list FROM table1 INNER JOIN table2 ON condition1
    INNER JOIN table3 ON condition2
    ...;



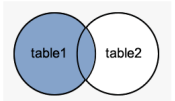
3. Alias in MySQL (AS)

1. Aliases in MySQL is used to give a temporary name to a table or a column in a table for the purpose of a particular query. It works as a nickname for expressing the tables or column names. It makes the query short and neat.
2. SELECT col_name AS alias_name FROM table_name;
3. SELECT col_name1, col_name2,... FROM table_name AS alias_name;

4. **OUTER JOIN**

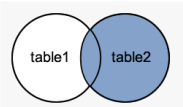
1. LEFT JOIN

1. This returns a resulting table that all the data from left table and the matched data from the right table.
2. SELECT columns FROM table LEFT JOIN table2 ON Join_Condition;



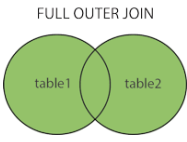
2. RIGHT JOIN

1. This returns a resulting table that all the data from right table and the matched data from the left table.
2. SELECT columns FROM table RIGHT JOIN table2 ON join_cond;



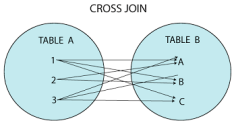
3. FULL JOIN

1. This returns a resulting table that contains all data when there is a match on left or right table data.
2. **Emulated** in MySQL using LEFT and RIGHT JOIN.
3. LEFT JOIN UNION RIGHT JOIN.
4. SELECT columns FROM table1 as t1 LEFT JOIN table2 as t2 ON t1.id = t2.id
    UNION
    SELECT columns FROM table1 as t1 RIGHT JOIN table2 as t2 ON t1.id = t2.id;
5. UNION ALL, can also be used this will duplicate values as well while UNION gives unique values.



5. CROSS JOIN

1. This returns all the cartesian products of the data present in both tables. Hence, all possible variations are reflected in the output.
2. Used rarely in practical purpose.
3. Table-1 has 10 rows and table-2 has 5, then resultant would have 50 rows.
4. SELECT column-lists FROM table1 CROSS JOIN table2;



6. SELF JOIN

1. It is used to get the output from a particular table when the same table is joined to itself.
2. Used very less.
3. Emulated using INNER JOIN.
4. SELECT columns FROM table as t1 INNER JOIN table as t2 ON t1.id = t2.id;

7. Join without using join keywords.

1. SELECT * FROM table1, table2 WHERE condition;
2. e.g., SELECT artist_name, album_name, year_recordedFROM artist, albumWHERE artist.id = album.artist_id;

SET OPERATIONS

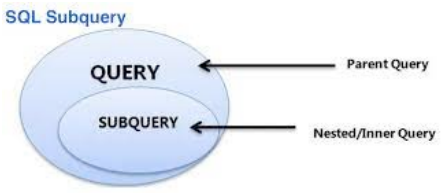
1. Used to combine multiple select statements.
2. Always gives distinct rows.

JOIN	SET Operations
Combines multiple tables based on matching condition.	Combination is resulting set from two or more SELECT statements.
Column wise combination.	Row wise combination.
Data types of two tables can be different.	Datatypes of corresponding columns from each table should be the same.
Can generate both distinct or duplicate rows.	Generate distinct rows.
The number of column(s) selected may or may not be the same from each table.	The number of column(s) selected must be the same from each table.
Combines results horizontally.	Combines results vertically.

3. **UNION**
   1. Combines two or more SELECT statements.
   2. SELECT * FROM table1
      UNION
      SELECT * FROM table2;
   3. Number of column, order of column must be same for table1 and table2.
4. **INTERSECT**
   1. Returns common values of the tables.
   2. Emulated.
   3. SELECT DISTINCT column-list FROM table-1 INNER JOIN table-2 USING(join_cond);
   4. SELECT DISTINCT * FROM table1 INNER JOIN table2 ON USING(id);
5. **MINUS**
   1. This operator returns the distinct row from the first table that does not occur in the second table.
   2. Emulated.
   3. SELECT column_list FROM table1 LEFT JOIN table2 ON condition WHERE table2.column_name IS NULL;
   4. e.g., SELECT id FROM table-1 LEFT JOIN table-2 USING(id) WHERE table-2.id IS NULL;

SUB QUERIES

1. Outer query depends on inner query.
2. Alternative to joins.
3. Nested queries.
4. SELECT column_list (s) FROM table_name WHERE column_name OPERATOR (SELECT column_list (s) FROM table_name [WHERE]);
5. e.g., SELECT * FROM table1 WHERE col1 IN (SELECT col1 FROM table1);
6. Sub queries exist mainly in 3 clauses
   1. Inside a WHERE clause.



2. Inside a FROM clause.
3. Inside a SELECT clause.

7. **Subquery using FROM clause**
   1. SELECT MAX(rating) FROM (SELECT * FROM movie WHERE country = 'India') as temp;
8. **Subquery using SELECT**
   1. SELECT (SELECT column_list(s) FROM T_name WHERE condition), columnList(s) FROM T2_name WHERE condition;
9. **Derived Subquery**
   1. SELECT columnLists(s) FROM (SELECT columnLists(s) FROM table_name WHERE [condition]) as new_table_name;
10. **Co-related sub-queries**
    1. With a normal nested subquery, the inner SELECT query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

```
SELECT column1, column2, ....
FROM table1 as outer
WHERE column1 operator
                    (SELECT column1, column2
                     FROM table2
                     WHERE expr1 =
                                outer.expr2);
```

JOIN VS SUB-QUERIES

JOINS	SUBQUERIES
Faster	Slower
Joins maximise calculation burden on DBMS	Keeps responsibility of calculation on user.
Complex, difficult to understand and implement	Comparatively easy to understand and implement.
Choosing optimal join for optimal use case is difficult	Easy.

MySQL VIEWS

1. A view is a database object that has no values. Its contents are based on the base table. It contains rows and columns similar to the real table.
2. In MySQL, the View is a **virtual table** created by a query by joining one or more tables. It is operated similarly to the base table but does not contain any data of its own.
3. The View and table have one main difference that the views are definitions built on top of other tables (or views). If any changes occur in the underlying table, the same changes reflected in the View also.
4. CREATE VIEW view_name AS SELECT columns FROM tables [WHERE conditions];
5. ALTER VIEW view_name AS SELECT columns FROM table WHERE conditions;
6. DROP VIEW IF EXISTS view_name;
7. CREATE VIEW Trainer AS SELECT c.course_name, c.trainer, t.email FROM courses c, contact t WHERE c.id = t.id; (View using Join clause).

NOTE: We can also import/export table schema from files (.csv or json).

LEC-11: Normalisation

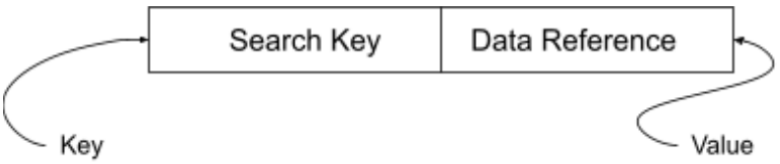
1. **Normalisation** is a step towards DB optimisation.
2. **Functional Dependency (FD)**
    1. It's a relationship between the primary key attribute (usually) of the relation to that of the other attribute of the relation.
    2. X -> Y, the left side of FD is known as a **Determinant**, the right side of the production is known as a **Dependent**.
    3. **Types of FD**
        1. **Trivial FD**
            1. A → B has trivial functional dependency if B is a subset of A. A->A, B->B are also Trivial FD.
        2. **Non-trivial FD**
            1. A → B has a non-trivial functional dependency if B is not a subset of A. [A intersection B is NULL].
    4. **Rules of FD (Armstrong's axioms)**
        1. **Reflexive**
            1. If 'A' is a set of attributes and 'B' is a subset of 'A'. Then, A→ B holds.
            2. If A ⊇ B then A → B.
        2. **Augmentation**
            1. If B can be determined from A, then adding an attribute to this functional dependency won't change anything.
            2. If A→ B holds, then AX→ BX holds too. 'X' being a set of attributes.
        3. **Transitivity**
            1. If A determines B and B determines C, we can say that A determines C.
            2. if A→ B and B→ C then A→ C.
3. **Why Normalisation?**
    1. To avoid redundancy in the DB, not to store redundant data.
4. **What happen if we have redundant data?**
    1. Insertion, deletion and updation anomalies arises.
5. **Anomalies**
    1. Anomalies means abnormalities, there are three types of anomalies introduced by data redundancy.
    2. **Insertion anomaly**
        1. When certain data (attribute) can not be inserted into the DB without the presence of other data.
    3. **Deletion anomaly**
        1. The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
    4. **Updation anomaly** (or modification anomaly)
        1. The update anomaly is when an update of a single data value requires multiple rows of data to be updated.
        2. Due to updation to many places, may be **Data inconsistency** arises, if one forgets to update the data at all the intended places.
    5. Due to these anomalies, **DB size increases** and **DB performance become very slow.**
    6. To rectify these anomalies and the effect of these of DB, we use **Database optimisation technique** called **NORMALISATION**.
6. **What is Normalisation?**
    1. Normalisation is used to minimise the redundancy from a relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
    2. Normalisation divides the composite attributes into individual attributes OR larger table into smaller and links them using relationships.
    3. The normal form is used to reduce redundancy from the database table.
7. **Types of Normal forms**
    1. **1NF**
        1. Every relation cell must have atomic value.
        2. Relation must not have multi-valued attributes.

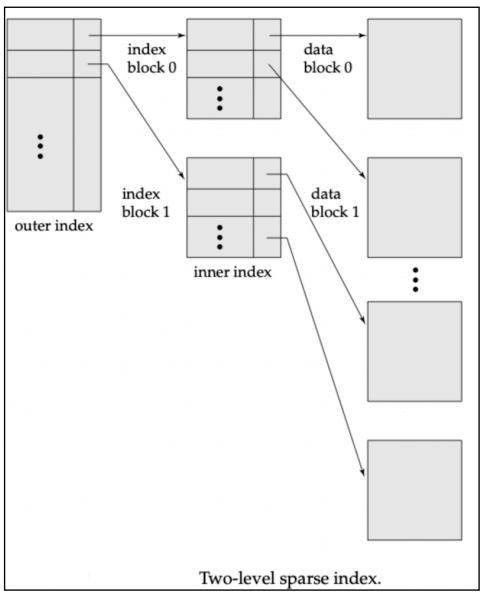
2. **2NF**
    1. Relation must be in 1NF.
    2. There should not be any partial dependency.
        1. All non-prime attributes must be fully dependent on PK.
        2. Non prime attribute can not depend on the part of the PK.
3. **3NF**
    1. Relation must be in 2NF.
    2. No transitivity dependency exists.
        1. Non-prime attribute should not find a non-prime attribute.
4. **BCNF (Boyce-Codd normal form)**
    1. Relation must be in 3NF.
    2. FD: A -> B, A must be a super key.
        1. We must not derive prime attribute from any prime or non-prime attribute.

8. **Advantages of Normalisation**
    1. Normalisation helps to minimise data redundancy.
    2. Greater overall database organisation.
    3. Data consistency is maintained in DB.

LEC-14: Indexing in DBMS

1. **Indexing** is used to **optimise the performance** of a database by minimising the number of disk accesses required when a query is processed.
2. The index is a type of **data structure.** It is used to locate and access the data in a database table quickly.
3. **Speeds up operation** with read operations like **SELECT** queries, **WHERE** clause etc.
4. **Search Key**: Contains copy of primary key or candidate key of the table or something else.
5. **Data Reference**: Pointer holding the address of disk block where the value of the corresponding key is stored.
6. Indexing is **optional**, but increases access speed. It is not the primary mean to access the tuple, it is the secondary mean.
7. **Index file is always sorted.**
8. **Indexing Methods**
    1. **Primary Index (Clustering Index)**
        1. A file may have several indices, on different search keys. If the data file containing the records is sequentially ordered, a Primary index is an index whose search key also defines the sequential order of the file.
        2. **NOTE**: The term primary index is sometimes used to mean an index on a primary key. However, such usage is **nonstandard** and **should be avoided.**
        3. All files are ordered sequentially on some search key. It could be Primary Key or non-primary key.
        4. **Dense And Sparse Indices**
            1. **Dense Index**
                1. The dense index contains an index record for every search key value in the data file.
                2. The index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search-key value would be stored sequentially after the first record.
                3. It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.
            2. **Sparse Index**
                1. An index record appears for only some of the search-key values.
                2. Sparse Index helps you to resolve the issues of dense Indexing in DBMS. In this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrieved, the block address will be fetched.
        5. Primary Indexing can be based on Data file is sorted w.r.t Primary Key attribute or non-key attributes.
        6. **Based on Key attribute**
            1. Data file is sorted w.r.t primary key attribute.
            2. PK will be used as search-key in Index.
            3. Sparse Index will be formed i.e., no. of entries in the index file = no. of blocks in datafile.
        7. **Based on Non-Key attribute**
            1. Data file is sorted w.r.t non-key attribute.
            2. No. Of entries in the index = unique non-key attribute value in the data file.
            3. This is dense index as, all the unique values have an entry in the index file.
            4. E.g., Let's assume that a company recruited many employees in various departments. In this case, clustering indexing in DBMS should be created for all employees who belong to the same dept.
        8. **Multi-level Index**
            1. Index with two or more levels.
            2. If the single level index become enough large that the binary search it self would take much time, we can break down indexing into multiple levels.





    2. **Secondary Index (Non-Clustering Index)**
        1. Datafile is unsorted. Hence, Primary Indexing is not possible.
        2. Can be done on key or non-key attribute.
        3. Called secondary indexing because normally one indexing is already applied.
        4. No. Of entries in the index file = no. of records in the data file.
        5. It's an example of Dense index.

```
-- Q-1. Write an SQL query to fetch “FIRST_NAME” from Worker table using the alias
name as <WORKER_NAME>.
select first_name AS WORKER_NAME from worker;

-- Q-2. Write an SQL query to fetch “FIRST_NAME” from Worker table in upper case.
select UPPER(first_name) from worker;

-- Q-3. Write an SQL query to fetch unique values of DEPARTMENT from Worker table.
SELECT distinct department from worker;

-- Q-4. Write an SQL query to print the first three characters of  FIRST_NAME from
Worker table.
select substring(first_name, 1, 3) from worker;

-- Q-5. Write an SQL query to find the position of the alphabet (‘b’) in the first
name column ‘Amitabh’ from Worker table.
select INSTR(first_name, 'B') from worker where first_name = 'Amitabh';

-- Q-6. Write an SQL query to print the FIRST_NAME from Worker table after
removing white spaces from the right side.
select RTRIM(first_name) from worker;

-- Q-7. Write an SQL query to print the DEPARTMENT from Worker table after
removing white spaces from the left side.
select LTRIM(first_name) from worker;

-- Q-8. Write an SQL query that fetches the unique values of DEPARTMENT from
Worker table and prints its length.
select distinct department, LENGTH(department) from worker;

-- Q-9. Write an SQL query to print the FIRST_NAME from Worker table after
replacing ‘a’ with ‘A’.
select REPLACE(first_name, 'a', 'A')  from worker;

-- Q-10. Write an SQL query to print the FIRST_NAME and LAST_NAME from Worker
table into a single column COMPLETE_NAME.
-- A space char should separate them.
select CONCAT(first_name, ' ', last_name) AS COMPLETE_NAME from worker;

-- Q-11. Write an SQL query to print all Worker details from the Worker table
order by FIRST_NAME Ascending.
select * from worker ORDER by first_name;

-- Q-12. Write an SQL query to print all Worker details from the Worker table
order by
-- FIRST_NAME Ascending and DEPARTMENT Descending.
select * from worker order by first_name, department DESC;

-- Q-13. Write an SQL query to print details for Workers with the first name as
“Vipul” and “Satish” from Worker table.
```

```
select * from worker where first_name IN ('Vipul', 'Satish');

-- Q-14. Write an SQL query to print details of workers excluding first names,
“Vipul” and “Satish” from Worker table.
select * from worker where first_name NOT IN ('Vipul', 'Satish');

-- Q-15. Write an SQL query to print details of Workers with DEPARTMENT name as
“Admin*”.
select * from worker where department LIKE 'Admin%';

-- Q-16. Write an SQL query to print details of the Workers whose FIRST_NAME
contains ‘a’.
select * from worker where first_name LIKE '%a%';

-- Q-17. Write an SQL query to print details of the Workers whose FIRST_NAME ends
with ‘a’.
select * from worker where first_name LIKE '%a';

-- Q-18. Write an SQL query to print details of the Workers whose FIRST_NAME ends
with ‘h’ and contains six alphabets.
select * from worker where first_name LIKE '_____h';

-- Q-19. Write an SQL query to print details of the Workers whose SALARY lies
between 100000 and 500000.
select * from worker where salary between 100000 AND 500000;

-- Q-20. Write an SQL query to print details of the Workers who have joined in
Feb’2014.
select * from worker where YEAR(joining_date) = 2014 AND MONTH(joining_date) = 02;

-- Q-21. Write an SQL query to fetch the count of employees working in the
department ‘Admin’.
select department, count(*) from worker where department = 'Admin';

-- Q-22. Write an SQL query to fetch worker full names with salaries >= 50000 and
<= 100000.
select concat(first_name, ' ', last_name) from worker
where salary between 50000 and 100000;

-- Q-23. Write an SQL query to fetch the no. of workers for each department in the
descending order.
select department, count(worker_id) AS no_of_worker from worker group by
department
ORDER BY no_of_worker desc;

-- Q-24. Write an SQL query to print details of the Workers who are also Managers.
select w.* from worker as w inner join title as t on w.worker_id = t.worker_ref_id
where t.worker_title = 'Manager';

-- Q-25. Write an SQL query to fetch number (more than 1) of same titles in the
```

```
ORG of different types.
select worker_title, count(*) as count from title group by worker_title having
count > 1;

-- Q-26. Write an SQL query to show only odd rows from a table.
-- select * from worker where MOD (WORKER_ID, 2) != 0;
select * from worker where MOD (WORKER_ID, 2) <> 0;

-- Q-27. Write an SQL query to show only even rows from a table.
select * from worker where MOD (WORKER_ID, 2) = 0;

-- Q-28. Write an SQL query to clone a new table from another table.
CREATE TABLE worker_clone LIKE worker;
INSERT INTO worker_clone select * from worker;
select * from worker_clone;

-- Q-29. Write an SQL query to fetch intersecting records of two tables.
select worker.* from worker inner join worker_clone using(worker_id);

-- Q-30. Write an SQL query to show records from one table that another table does
not have.
-- MINUS
select worker.* from worker left join worker_clone using(worker_id) WHERE
worker_clone.worker_id is NULL;

-- Q-31. Write an SQL query to show the current date and time.
-- DUAL
select curdate();
select now();

-- Q-32. Write an SQL query to show the top n (say 5) records of a table order by
descending salary.
select * from worker order by salary desc LIMIT 5;

-- Q-33. Write an SQL query to determine the nth (say n=5) highest salary from a
table.
select * from worker order by salary desc LIMIT 4,1;

-- Q-34. Write an SQL query to determine the 5th highest salary without using
LIMIT keyword.
select salary from worker w1
WHERE 4 = (
SELECT COUNT(DISTINCT (w2.salary))
from worker w2
where w2.salary >= w1.salary
);

-- Q-35. Write an SQL query to fetch the list of employees with the same salary.
select w1.* from worker w1, worker w2 where w1.salary = w2.salary and w1.worker_id
!= w2.worker_id;
```

```
-- Q-36. Write an SQL query to show the second highest salary from a table using
sub-query.
select max(salary) from worker
where salary not in (select max(salary) from worker);

-- Q-37. Write an SQL query to show one row twice in results from a table.
select * from worker
UNION ALL
select * from worker ORDER BY worker_id;

-- Q-38. Write an SQL query to list worker_id who does not get bonus.
select worker_id from worker where worker_id not in (select worker_ref_id from
bonus);

-- Q-39. Write an SQL query to fetch the first 50% records from a table.
select * from worker where worker_id <= ( select count(worker_id)/2 from worker);

-- Q-40. Write an SQL query to fetch the departments that have less than 4 people
in it.
select department, count(department) as depCount from worker group by department
having depCount < 4;

-- Q-41. Write an SQL query to show all departments along with the number of
people in there.
select department, count(department) as depCount from worker group by department;

-- Q-42. Write an SQL query to show the last record from a table.
select * from worker where worker_id = (select max(worker_id) from worker);

-- Q-43. Write an SQL query to fetch the first row of a table.
select * from worker where worker_id = (select min(worker_id) from worker);

-- Q-44. Write an SQL query to fetch the last five records from a table.
(select * from worker order by worker_id desc limit 5) order by worker_id;

-- Q-45. Write an SQL query to print the name of employees having the highest
salary in each department.
select w.department, w.first_name, w.salary from
 (select max(salary) as maxsal, department from worker group by department) temp
inner join worker w on temp.department = w.department and temp.maxsal = w.salary;

-- Q-46. Write an SQL query to fetch three max salaries from a table using
co-related subquery
select distinct salary from worker w1
where 3 >= (select count(distinct salary) from worker w2 where w1.salary <=
w2.salary) order by w1.salary desc;
-- DRY RUN AFTER REVISING THE CORELATED SUBQUERY CONCEPT FROM LEC-9.
select distinct salary from worker order by salary desc limit 3;
```

```
-- Q-47. Write an SQL query to fetch three min salaries from a table using
co-related subquery
select distinct salary from worker w1
where 3 >= (select count(distinct salary) from worker w2 where w1.salary >=
w2.salary) order by w1.salary desc;

-- Q-48. Write an SQL query to fetch nth max salaries from a table.
select distinct salary from worker w1
where n >= (select count(distinct salary) from worker w2 where w1.salary <=
w2.salary) order by w1.salary desc;

-- Q-49. Write an SQL query to fetch departments along with the total salaries
paid for each of them.
select department , sum(salary) as depSal from worker group by department order by
depSal desc;

-- Q-50. Write an SQL query to fetch the names of workers who earn the highest
salary.
select first_name, salary from worker where salary = (select max(Salary) from
worker);
```