SQL (by Mark Philip)

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.
            - **Syntax:** CREATE DATABASE db_name;
            - **Syntax:** CREATE TABLE table_name(
              table_col col attributes;
              );
        2. **ALTER TABLE**: modification in table structure. e.g, change column datatype or add/remove columns.
            - Add a column:
                - **Syntax:** ALTER TABLE table_name
                  ADD column_name datatype;
            - Drop a column:
                - **Syntax:** ALTER TABLE table_name
                  DROP COLUMN column_name;
            - Modify a column's data type:
                - **Syntax:** ALTER TABLE table_name
                  MODIFY COLUMN column_name new_datatype;
            - Rename a column:
                - **Syntax:** ALTER TABLE table_name
                  RENAME COLUMN old_column_name TO new_column_name;
            - Add a primary key constraint:
                - **Syntax:** ALTER TABLE table_name
                  ADD CONSTRAINT constraint_name PRIMARY KEY (column_name);
            - Drop a primary key constraint:
                - **Syntax:** ALTER TABLE table_name
                  DROP CONSTRAINT constraint_name;
            - Add a foreign key constraint:
                - **Syntax:** ALTER TABLE table_name
                  ADD CONSTRAINT constraint_name FOREIGN KEY (column_name)
                  REFERENCES referenced_table (referenced_column);
            - Drop a foreign key constraint:
                - **Syntax:** ALTER TABLE table_name
                  DROP CONSTRAINT constraint_name;

- Add a unique constraint:
  - **Syntax:** ALTER TABLE table_name
    ADD CONSTRAINT constraint_name UNIQUE (column_name);
- Drop a unique constraint:
  - **Syntax:** ALTER TABLE table_name
    DROP CONSTRAINT constraint_name;
- Rename a table:
  - **Syntax:** ALTER TABLE old_table_name
    RENAME TO new_table_name;

3. **DROP**: delete table, DB, view.
   - **Syntax:** DROP DATABASE db_name;
   - **Syntax:** DROP TABLE table_name;
4. **TRUNCATE**: remove all the tuples from the table.
   - **Syntax:** TRUNCATE TABLE table_name;
5. **RENAME**: rename DB name, table name, column name etc.
   - **Syntax:** RENAME TABLE older_table_name TO new_table_name;
   - RENAME to change db_name has been removed.

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
   6. **INSERT:** insert data into a relation
      - INSERT INTO TITLE (WORKER_REF_ID, WORKER_TITLE, AFFECTED_FROM) VALUES
        (001,'Manager', '2026-02-20 00:00:00'),
        (005,'Executive', '2026-02-11 00:00:00');

   7. **UPDATE**: update relation data.
   8. **DELETE:** delete row(s) from the relation
      - DELETE FROM TITLE WHERE WORKER_REF_ID = 3 AND WORKER_TITLE = 'Manager' AND AFFECTED_FROM = '2026-02-20 00:00:00';
      - DELETE FROM TITLE WHERE
        (WORKER_REF_ID, WORKER_TITLE, AFFECTED_FROM) IN (
        (1, 'Manager', '2026-02-20 00:00:00'),
        (5, 'Executive', '2026-02-11 00:00:00')
        );

9. **DCL** (Data Control language): grant or revoke authorities from user.
   1. **GRANT:** access privileges to the DB
   2. **REVOKE:** revoke user access privileges.
10. **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. Syntax: **CREATE DATABASE IF NOT EXISTS db-name;**
    2. **Syntax:** USE db-name; //need to execute to choose on which DB CREATE TABLE etc commands will be executed. //make switching between DBs possible.
    3. **Syntax:** DROP DATABASE IF EXISTS db-name; //dropping database.
        - **Drop can also be used for tables.**
        - **Syntax:** DROP TABLE table_name;
    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>;
    - SELECT * FROM table_name; (to view complete table)
    - SELECT col_name FROM tablle_name; (to view a specific column)

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**
    - Reduce rows based on given conditions.
    - **Syntax:** SELECT * FROM customer WHERE Condition;
    - SELECT * FROM WORKER WHERE SALARY > 1000;
5. **BETWEEN**
    - SELECT * FROM WORKER WHERE SALARY BETWEEN 1000 AND 10000;
6. **IN**
    - Reduces **OR** conditions;
    - SELECT * FROM WORKER WHERE DEPARTMENT = 'HR' OR DEPARTMENT ='Admin';
    - SELECT * FROM WORKER WHERE FIRST_NAME IN ('Mark', 'Hanok', 'Titus');
    - IN will give output containing all these names
7. **AND/OR/NOT**
    1. AND: WHERE cond1 AND cond2
    2. OR: WHERE cond1 OR cond2
        - SELECT * FROM WORKER WHERE DEPARTMENT = 'HR' OR DEPARTMENT ='Admin';
    3. NOT: WHERE col_name NOT IN (1,2,3,4);
        - SELECT * FROM WORKER WHERE DEPARTMENT NOT IN ('HR');
8. **IS NULL**
    - SELECT * FROM BONUS WHERE BONUS_AMOUNT IS NULL;

9. **Pattern Searching / Wildcard ('%', '_')**
   - '%', any number of character from 0 to n. Similar to '*' asterisk in regex.
   - '_', only one character.
   - SELECT * FROM WORKER WHERE FIRST_NAME LIKE '%a';
   - '%a%' - "mark"(n no. of char in start and last)
     '%a_' - "mmak" (n no. of char before but only 1 at last)
     '_a%' - "marrrrk" (only 1 before anf n no. of at last)
     '_a_' - "mam" (1 at start and 1 at end)
     '__a__' - "mmamm" (2 at start and 2 at end)
     'a__' - " amm" (non before and only 2 at last)
     ' %%a%%%' / '%a% - mam/mmmammm
10. **ORDER BY**
    - Sorting the data retrieved using **WHERE** clause.
    - SELECT * FROM WORKER ORDER BY SALARY;
    - By default ascending,or can also add ASC at last.
    - **Syntax:** ORDER BY <column-name> DESC;
    - SELECT * FROM WORKER ORDER BY SALARY DESC;
    - DESC = Descending and ASC = Ascending
    - DESC WORKER;(it will give all the details about the column.)

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.
       - SELECT DEPARTMENT FROM WORKER GROUP BY DEPARTMENT;
       - no aggregation added so it will work as distinct
    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()** - SELECT DEPARTMENT, COUNT(DEPARTMENT) FROM WORKER GROUP BY DEPARTMENT;
       2. **SUM()** - SELECT DEPARTMENT, SUM(SALARY) FROM WORKER GROUP BY DEPARTMENT;
       3. **AVG()** - SELECT DEPARTMENT, AVG(SALARY) FROM WORKER GROUP BY DEPARTMENT; (avg salary based on department)
       4. **MIN() - SELECT DEPARTMENT ,MIN(SALARY) FROM WORKER GROUP BY DEPARTMENT;**
       5. **MAX() - SELECT DEPARTMENT ,MAX(SALARY) FROM WORKER GROUP BY DEPARTMENT;**
12. **DISTINCT**
    1. Find distinct values in the table.
    2. **Syntax:** SELECT DISTINCT(col_name) FROM table_name;
    3. SELECT DISTINCT SALARY FROM WORKER;
    4. GROUP BY can also be used for the same
       1. **Syntax:** 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 DEPARTMENT, COUNT(DEPARTMENT) FROM WORKER GROUP BY DEPARTMENT HAVING COUNT(DEPARTMENT)=2;
    4. Select COUNT(cust_id), country from customer GROUP BY country HAVING COUNT(cust_id) > 50;

5. 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 -** good practice is to keep primary key value which is INT, because then searching n other operations are faster.
    - ```
      CREATE table A(
      id INT ,
      name VARCHAR(255),
      PRIMARY KEY(id)
      );
      ```
    - ```
      create table A(
      id INT PRIMARY KEY,
      name VARCHAR(255),
      );
      ```
    - PK is not null, unique and only one per table.

2. **Foreign Key**
    - FK refers to PK of other table.
    - Each relation can having any number of FK.
    - ```
      CREATE TABLE M (
          id INTEGER PRIMARY KEY,
          name VARCHAR(255),
          worker_id INT,
          WORKER_REF_ID INT,
          amount INT(10),
          FOREIGN KEY (worker_id) REFERENCES WORKER(WORKER_ID),
          FOREIGN KEY (WORKER_REF_ID) REFERENCES TITLE(WORKER_REF_ID)
      );
      ```
3. **UNIQUE**
    - Unique, can be null, table can have multiple unique attributes.
    - ```
      CREATE TABLE M (
          id INT PRIMARY KEY,
          name VARCHAR(255) UNIQUE
      );
      ```
4. **CHECK**
    - ```
      CREATE TABLE M (
        id INT PRIMARY KEY
        balance INT(10),
        CONSTRAINT check_balance CHECK(balance>1000),
      );
      ```
    - “check_balance”, can also avoid this, MySQL generates name of constraint automatically.
      ```

5. **DEFAULT**
   - Set default value of the column.
   - CREATE TABLE M (
     age INT DEFAULT 15
     );
   - An attribute can be **PK and FK both** in a table.
6. **ALTER OPERATIONS**
   - Changes schema

   1. **ADD:**
      - **Add new column.**
      - ALTER TABLE table_name ADD new_col_name datatype ADD new_col_name_2 datatype;
      - ALTER TABLE M ADD sex char(1);
   2. **MODIFY:**
      - **Change datatype of an attribute.**
      - ALTER TABLE table-name MODIFY col-name col-datatype;
        ALTER TABLE M MODIFY sex VARCHAR(1) DEFAULT 0;
   3. **CHANGE COLUMN:**
      - **Rename column name.**
      - ALTER TABLE table-name CHANGE COLUMN old-col-name new-col-name new-col-datatype;
        ALTER TABLE M CHANGE COLUMN sex S char(1);
   4. **DROP COLUMN:**
      - **Drop a column completely.**
      - ALTER TABLE table-name DROP COLUMN col-name;
      - ALTER TABLE M DROP COLUMN S;
   5. **RENAME:**
      - **Rename table name itself.**
      - ALTER TABLE table-name RENAME TO new-table-name;
      - ALTER TABLE M RENAME Details;
      - ALTER TABLE M RENAME TO Details;

DATA MANIPULATION LANGUAGE (DML)

1. **INSERT**
   - **Syntax:** INSERT INTO table-name(col1, col2, col3) VALUES (v1, v2, v3), (val1, val2, val3);
   - (1) INSERT INTO Details
     (id,name,worker_id,WORKER_REF_ID,balance,age)
     VALUES (1,'m',1,1,10000,12),
     (2,'a',2,2,12332,20);
   - (2) INSERT INTO Details
     (id,name,worker_id,WORKER_REF_ID,balance,age)
     VALUES (3,'d',3,3,222332,22);
   - (3) INSERT INTO Details VALUES (4,'g',4,5,100000,50);
   - (4) INSERT INTO Details (id,name) VALUES (5,'dd');

2. **UPDATE**
   - **Syntax:** UPDATE table-name SET col1 = 1, col2 = 'abc' WHERE id = 1;
   - UPDATE Details SET Worker_id = 5,WORKER_REF_ID=5, balance=11111 WHERE id =5;
   - Update multiple rows:
     - SET SQL_SAFE_UPDATES=0;
       UPDATE Details SET age=100; (if directly runs 2nd line it wont run , because due to safety concerns, it doesnt allows to change all data at once, thatswhy there is need of 1st line.)
       (if value back to 1 then safety mode is on again)
   - **ON UPDATE CASCADE**
     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 firsttable then using the ON UPDATE CASCADE foreign key of the second table automatically get updated.
3. **DELETE**

   - **Syntax:** DELETE FROM table_name WHERE Condition;
     - DELETE FROM Details WHERE age = 50;
   - **Syntax:** DELETE FROM table-name; //all rows will be deleted.
     - DELETE FROM Details;
   - **DELETE CASCADE - (to overcome DELETE constraint of Referential constraints)**
     - What would happen to child entry if parent table's entry is deleted?
     - When deleting from parent, those inherited value will also get deleted from child.
     - CREATE TABLE A(
       id INT PRIMARY KEY,
       age INT DEFAULT 100
       );

       INSERT INTO A(id,age) VALUES
       (1,12),
       (2,23);

       SELECT * FROM A;

       CREATE TABLE B(
       b_id INT,
       age INT DEFAULT 100,
       FOREIGN KEY(b_id) references A(id) ON DELETE CASCADE
       );

       INSERT INTO B(b_id,age) VALUES
       (1,122),
       (2,223);

       SELECT * FROM B;

       DELETE FROM A WHERE id = 1;

- **ON DELETE NULL -** (can FK have null values?)
  - ```
    CREATE TABLE C(
    id INT PRIMARY KEY,
    age INT DEFAULT 100
    );

    INSERT INTO C(id,age) VALUES
    (1,12),
    (2,23);

    SELECT * FROM C;

    CREATE TABLE D(
    d_id INT,
    age INT DEFAULT 100,
    FOREIGN KEY(d_id) references C(id) ON DELETE SET NULL
    );


    INSERT INTO D(d_id,age) VALUES
    (1,122),
    (2,223);

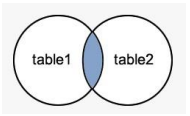
    SELECT * FROM D;

    DELETE FROM C WHERE id = 1;
    ```

4. **REPLACE**
   - Primarily used for already present tuple in a table.
     - ```
       REPLACE INTO C( id,age) VALUES(1,99);
       REPLACE INTO C SET id = 1, age = 9999;
       ```
       (we already had entry with id as 1 , so replaced)
   - As UPDATE, using REPLACE with the help of WHERE clause in PK, then that row will be replaced.
     - ```
       REPLACE INTO C(id) VALUES (0);
       REPLACE INTO C SET id = 5;
       ```
       (we didn't had entry with id as 0 , so inserted)
     - If row is not present REPLACE will insert new row while UPDATE will do nothing in such cases.
   - As INSERT, if there is no duplicate data new tuple will be inserted.
   - If all column details not given values will be null, but if we have a PK, then it has to be given
       ```

JOINING TABLES

- All **RDBMS** are relational in nature, we refer to other tables to get meaningful outcomes.
- FK are used to do reference to other table and these relations are used to fetch data using JOINS.

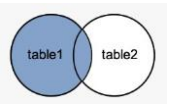


1. **INNER JOIN**
   - Returns a resultant table that has matching values from both the tables or all the tables.
   - **Syntax:** SELECT column-list FROM table1 INNER JOIN table2 ON condition1 INNER JOIN table3 ON condition2 ...;

a_id	First_name	b_id	Last_name
1	mark	1	philip
2	Hanok	2	philip

   - SELECT A1.*,B1.* FROM A1 INNER JOIN B1 ON A1.a_id = B1.b_id; (. * means all columns)
   - SELECT A1.*,B1.* FROM A1 INNER JOIN B1 ON A1.a_id = B1.b_id INNER JOIN C1 ON B1.b_id = C1.c_id;
   - If table1 has 5 entries and table2 has 3 , then will only display the common once.
   - **Alias in MySQL (AS)**
     - 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 shortand neat.
     - SELECT a_id AS id ,First_name AS name FROM A1;
     - SELECT a_id,First_name FROM A1 AS a;

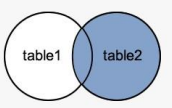
1. **OUTER JOIN**
   1. **LEFT JOIN**
      - This returns a resulting table that all the data from left table and the matched data from the right table.



      - **Syntax:** SELECT columns FROM table LEFT JOIN table2 ON Join_Condition;
      - SELECT A1.*,B1.* FROM A1 LEFT JOIN B1 ON A1.a_id = B1.b_id; (If left table has 5 entries and right table has 3 then rest 2 will have null as values.)

a_id	First_name	b_id	Last_name
1	mark	1	philip
2	Hanok	2	philip
3	sheelu	NULL	NULL
4	titus	NULL	NULL

      - SELECT a_id AS id, First_name, Last_name FROM A1 LEFT JOIN B1 ON A1.a_id = B1.b_id; (both the tables have individual id's , so to avoid multiple of those use this)
   2. **RIGHT JOIN**
      - This returns a resulting table that all the data from right table and the matched data from the left table.

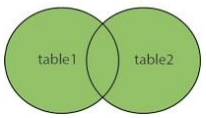


      - **Syntax:** SELECT columns FROM table RIGHT JOIN table2 ON join_cond;
      - SELECT A1.*, B1.* FROM A1 AS a RIGHT JOIN B1 AS b ON A1.a_id = B1.b_id; (we can replace both tables , all the values from the right table will be taken and matching from the left,(logic remains the same)

a_id	First_name	b_id	Last_name
1	mark	1	philip
2	Hanok	2	philip
3	sheelu	3	philip
4	titus	4	philip
NULL	NULL	5	philip
NULL	NULL	6	philip
NULL	NULL	7	philip
NULL	NULL	8	philip

   3. **FULL JOIN**
      - This returns a resulting table that contains all data when there is a match on left or right table data. (will find left join then right join then will give only unique values.)
      - **Emulated** in MySQL using LEFT and RIGHT JOIN.
      - LEFT JOIN UNION RIGHT JOIN.
      - **Syntax:** 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;

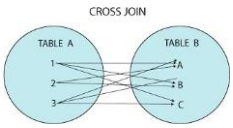
FULL OUTER JOIN



a_id	First_name	b_id	Last_name
1	mark	1	philip
2	Hanok	2	philip
3	sheelu	3	philip
4	titus	4	philip
NULL	NULL	5	philip
NULL	NULL	6	philip
NULL	NULL	7	philip
NULL	NULL	8	philip

      - SELECT A1.*,B1.* FROM A1 LEFT JOIN B1 ON A1.a_id = B1.b_id UNION SELECT A1.*,B1.* FROM A1 RIGHT JOIN B1 ON A1.a_id = B1.b_id; (will find left join then right join then will give just directly merge them.)
      - UNION ALL, can also be used this will duplicate values as well while UNION gives unique values.
      - SELECT A1.*,B1.* FROM A1 LEFT JOIN B1 ON A1.a_id = B1.b_id UNION ALL SELECT A1.*,B1.* FROM A1 RIGHT JOIN B1 ON A1.a_id = B1.b_id;

a_id	First_name	b_id	Last_name
1	mark	1	philip
2	Hanok	2	philip
3	sheelu	3	philip
4	titus	4	philip
1	mark	1	philip
2	Hanok	2	philip
3	sheelu	3	philip
4	titus	4	philip
NULL	NULL	5	philip
NULL	NULL	6	philip
NULL	NULL	7	philip
NULL	NULL	8	philip



a_id	First_name	b_id	Last_name
4	titus	1	philip
3	sheelu	1	philip
2	Hanok	1	philip
1	mark	1	philip
4	titus	2	philip
3	sheelu	2	philip
2	Hanok	2	philip
1	mark	2	philip
4	titus	3	philip
3	sheelu	3	philip
2	Hanok	3	philip
1	mark	3	philip
4	titus	4	philip
3	sheelu	4	philip
2	Hanok	4	philip
1	mark	4	philip
4	titus	5	philip
3	sheelu	5	philip
2	Hanok	5	philip
1	mark	5	philip
4	titus	6	philip
3	sheelu	6	philip
2	Hanok	6	philip
1	mark	6	philip
4	titus	7	philip

2. **CROSS JOIN**
   - This returns all the cartesian products of the data present in both tables. Hence, all possible variations are reflected in the output.
   - Used rarely in practical purpose.
   - Table-1 has 10 rows and table-2 has 5, then resultant would have 50 rows.
   - **Syntax:** SELECT column-lists FROM table1 CROSS JOIN table2;
   - SELECT A1.*,B1.* FROM A1 CROSS JOIN B1;

3. **SELF JOIN**
   - It is used to get the output from a particular table when the same table is joined to itself.
     (can use when we need a data from table 2 with respect to data from table 1)
   - Used very less.
   - Emulated using INNER JOIN.
   - **Syntax:** SELECT columns FROM table as t1 INNER JOIN table as t2 ON t1.id = t2.id;
   - SELECT A1.*,B1.* FROM A1 INNER JOIN B1 on A1.a_id = B1.b_id;
4. **Join without using join keywords.**
   - **Syntax:** SELECT * FROM table1, table2 WHERE condition;
   - SELECT * FROM A1,B1 WHERE A1.a_id = B1.b_id;

SET OPERATIONS

- Used to combine multiple select statements.
- 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.

1. **UNION**

col1	col2
A	1
B	2
C	3

col1	col2
A	1
B	1
D	3

col1	col2
A	1
B	2
C	3
B	1
D	3

- Combines two or more SELECT statements.(only unique rows)
- **Syntax:** SELECT * FROM table1
  UNION
  SELECT * FROM table2;
- SELECT* FROM table1 UNION SELECT* FROM table2;
- Number of column, order of column must be same for table1 and table2.

2. **INTERSECT**

col1	col2	col1	col2
A	1	A	1
B	2	B	1

- Returns common values of the tables.
- Emulated.
- **Syntax:** SELECT DISTINCT column-list FROM table-1 INNER JOIN table-2 USING(join_cond);
- SELECT DISTINCT col1 FROM table1 INNER JOIN table2 USING(col1);
- **Syntax:** SELECT DISTINCT * FROM table1 INNER JOIN table2 ON USING(id);

col1
A
B

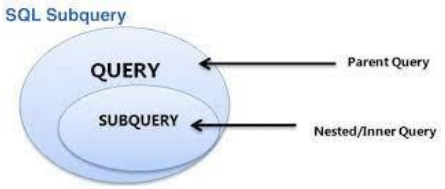
- SELECT DISTINCT* FROM table1 INNER JOIN table2 WHERE table1.col1 = table2.col1;

2. **MINUS**

- This operator returns the distinct row from the first table that does not occur in the second table.
- Emulated.
- **Syntax:** SELECT column_list FROM table1 LEFT JOIN table2 ON condition WHERE table2.column_name IS NULL;
- SELECT table1.* FROM table1 LEFT JOIN table2 USING(col1) WHERE table2.col1 IS NULL;

col1	col2
C	3

SUB QUERIES



- Outer query depends on inner query. [q1(q2)]
- Alternative to joins.
- Nested queries.
- **Syntax:** SELECT column_list (s) FROM table_name WHERE column_name OPERATOR(SELECT column_list (s) FROM table_name [WHERE]);
- q1(q2) = all values from col1 in table2(all values from col1 in table2 equal to 1)
  SELECT * FROM table1 WHERE col1 IN (SELECT col1 FROM table2 WHERE col2=1);

col1	col2
A	1
B	2

(In the subquery, SELECT col1 FROM table2 WHERE col2 = 1 retrieves values from col1 in table2 where col2 is equal to 1, resulting in values 'A' and 'B'.
The outer query selects all columns from table1 where col1 has values "a" and "b" (present in the result of the subquery)).

- q1(q2) = all values from col1 in table1 not equal to 'A'(all values from col1 in table2 equal to 1)
  SELECT *FROM table1 WHERE col1 IN (SELECT col1 FROM table2 WHERE col2 = 1) AND col1 != 'A';

col1	col2
B	2

In the subquery, SELECT col1 FROM table2 WHERE col2 = 1 retrieves values from col1 in table2 where col2 is equal to 1, resulting in values 'A' and 'B'.
The outer query selects all columns from table1 where col1 has values other than 'A' (present in the result of the subquery).

- Sub queries exist mainly in 3 clauses
  - Inside a WHERE clause.
  - Inside a FROM clause.
  - Inside a SELECT clause.
- **Subquery using FROM clause**

  SELECT col1 FROM(SELECT * FROM table2 WHERE col2 = 3) AS unique_values;

col1
D

  (here since we are not giving any instruction like SELECT col1 from table1,we dont know what to name it , so giving it a temprory name unique_values.)

- **Subquery using SELECT**
  - **Syntax:** SELECT (SELECT column_list(s) FROM T_name WHERE condition), columnList(s) FROM T2_name WHEREcondition;
  - SELECT (SELECT col1 FROM table2 WHERE col2 = 1 AND col1 = table1.col1) AS col1 FROM table1;

    (trying to fetch values from col1 in table2 where col2 equals 1, and then select those values in table1 where col1 matches the value in col1 of table2.)

col1
A
B
NULL

- **Derived Subquery**
  - SELECT columnLists(s) FROM (SELECT columnLists(s) FROM table_name WHERE [condition]) as new_table_name;
- **Co-related sub-queries(both the queries depends on each other)**
  - 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);
```

  - Lets find value which is 3rd largest :

    SELECT * FROM table1 t1 WHERE 3 = (SELECT COUNT(t2.col1) FROM table1 t2 WHERE t2.col1 >= t1.col1);

col1	col2
A	1

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

- 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.
- 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.
- 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.

- Creating a VIEW:
  - **Syntax:** CREATE VIEW view_name AS SELECT columns FROM tables [WHERE conditions];
  - CREATE VIEW name_list AS SELECT col1 FROM table1;

- Viewing the VIEW:
  - **Syntax:** SELECT * FROM view_name;
  - SELECT * FROM name_list;

- Altering the VIEW:
  - **Syntax:** ALTER VIEW view_name AS SELECT columns FROM table WHERE conditions;
  - ALTER VIEW name_list AS SELECT col1,col2 FROM table1;

- Droping the VIEW:
  - **Syntax:** DROP VIEW IF EXISTS view_name;
  - DROP VIEW IF EXISTS name_list;

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