SQL:-

USE <DB_NAME> → Use a database

SHOW TABLES; → A list of all tables in the database

DESCRIBE <table-name> → Shows a summary of the table's columns. Includes datatypes, length & names

```
mysql> DESCRIBE directors_genres;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| director_id | int(11)      | NO   | PRI | NULL    |       |
| genre       | varchar(100) | NO   | PRI | NULL    |       |
| prob        | float        | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```

Press Esc to exit full screen

→ Two primary keys. Genre is not unique but combination of genre & dir_id gives a unique value (probability). This type of primary key is called composite primary key.

→ Apps/Websites all of them use a database

```
Terminal - yash@yash-xub: ~
| 412312 | "nica noche, La"          | 1996 |      NULL |
| 412313 | "nica Verdade, A"         | 1985 |      NULL |
| 412314 | "nica Verdade, A"         | 1958 |      NULL |
| 412315 | "pa el nimo"              | 1962 |      NULL |
| 412316 | "zem blch krlu"           | 2002 |      NULL |
| 412317 | "rgammk"                  | 1991 |      NULL |
| 412318 | "zgnm Leyla"              | 1995 |      NULL |
| 412319 | " Istanbul"               | 2002 |      NULL |
| 412320 | "sterreich"               | 1983 |      NULL |
                                    | 1958 |      NULL |
388269 rows in set (0.90 sec)

mysql>
```

SELECT * from movies;

- SELECT → keyword
- * → all columns
- from → keyword
- movies → table name

→ IRL tables have 100s of columns. So * may not be needed all the time.

In that case:

SELECT name, year FROM movies;

will display only these two columns

```
mysql> describe movies;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| id        | int          | NO   | PRI | 0       |       |
| name      | varchar(100) | YES  | MUL | NULL    |       |
| year      | int          | YES  |     | NULL    |       |
| rankscore | float        | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

→ We don't tell SQL how to print it. SQL is a declarative language. Not a procedural language.

→ The tabular output generated after running a command is called a 'result-set'.

→ SELECT * is always going to be slower than SELECT — — specific columns.

(this order can be anything. Doesn't necessarily have to be the same order as in the original table.

But the order in which the rows appear in the result-set will be the same.

This is called row order preservation.

There is no proper guarantee that the order will be the same unless ORDER BY is used

→ Using backticks(`) allows us to add special characters in column names.

LIMIT :- when we don't want all rows to be displayed at once.

OFFSET :-

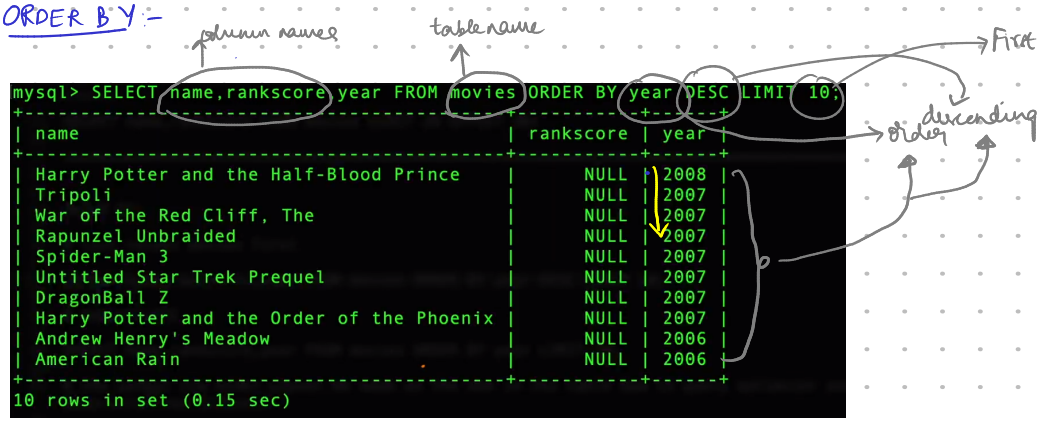
ex :- SELECT name, year FROM movies LIMIT 20; → displays only 20 values.

→ SELECT name, year FROM movies LIMIT 20 OFFSET 20; → ignore/offset first 20 values & print the next 20 values.

→ SELECT name, year FROM movies LIMIT 20 OFFSET 40; → ignore/offset the first 40 values & print the next 20 values.

Similar to page 1, page 2 etc, in google search results

ORDER BY :-



Similar to 'sort by' in websites.

Default is Ascending order.

Sorting by multiple columns :- SELECT ___ FROM ___ ORDER BY column_1 ASC, column_2 DESC;

↳ First sorts by column_1 in Ascending order & then sorts by column_2 in descending order.

DISTINCT :- Get all unique values in the column

ex :- SELECT DISTINCT genre FROM movies;

↑ All genres will be pointed.

Distinct can also be used on multiple columns

SELECT DISTINCT first_name, last_name FROM directors;

↑ Will list all unique name combinations.

(Apply distinct on one column but display multiple columns ?)

WHERE :- Apply condition to the query.

ex :- SELECT name, rankscore FROM movies WHERE rankscore > 9;

↑ Only movies with rating > 9

SELECT * FROM MOVIES WHERE rankscore > 9 ORDER BY rankscore;

↑ Same query but ordered by rankscore in Ascending order.

→ The conditions output can be (i) TRUE (ii) FALSE (iii) NULL

→ != & < > both imply not equal to in SQL.

SELECT * FROM movies WHERE rating <> 1.0;

↑ all movies except those whose rating is equal to 1.

NULL → unknown / missing / does not exist

↳ keyword

'=' does not work with NULL.

Ex:- Select * from movies where rankscore = NULL;

↳ will return empty set.

IS NULL & IS NOT NULL should be used instead.

SELECT * FROM movies WHERE RankScore IS NOT NULL;

IS NULL;

→ SELECT * FROM movies WHERE rankscore LIKE 9.8;

LIKE tries to match that pattern whereas = 9.8 tries to find the exact score value. Sometimes LIKE performs better.

Logical Operators:-

AND, OR, NOT, ALL, ANY, BETWEEN, EXISTS, IN, LIKE, SOME

AND :- Both cond1 & cond2

NOT :- SELECT * from movie WHERE NOT year <=2000;

↳ years >2000 only.

OR :- only one cond has to be true.

BETWEEN :- SELECT * FROM movies WHERE year BETWEEN 1999 AND 2000;

↳ Same as (year >= 1999 AND year <= 2000;)

Inclusive range ie., 1999 & 2000 are included.

BETWEEN a and b;

a should always be ≤ b otherwise it will result in an empty set.

IN :- SELECT * FROM movies where genre IN ('Comedy', 'Horror');

↳ Same as (genre = 'Comedy' OR genre = 'Horror';)

LIKE :- If we want names like "Bat..."

SELECT * FROM movies where name LIKE 'Bat%';

↳ Indicates 0 or more characters (Regular Expression)

↳ It's called a wildcard character.

_ :- Almost one character.

'_atman'

↳ Batman ✓ Bratman (2) ✗

Catman ✓

Ratman ✓

⋮

→ Backslash is the escape character.

Ex:- if working with percentages.

'\%.9%'

↳ Shows all %.9_ marks.

Aggregate functions :- They compute a single value over a set of rows.

COUNT | MIN | MAX | SUM | AVG → All work similarly.

→ SELECT MIN(year) FROM movies;

↑ Prints the minimum year from the column

COUNT :-

```
mysql> SELECT COUNT(*) FROM movies;
+----------+
| COUNT(*) |
+----------+
|   388269 |
+----------+
1 row in set (0.05 sec)
```

→ overall number of rows

→ Took all the columns.

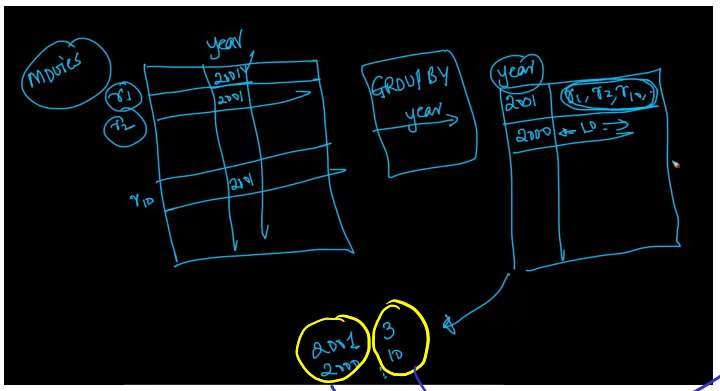
count(column_name) does not include NULL values.
COUNT(*) does.

SELECT COUNT(*) FROM movies WHERE year>2000;

↳ Returns number of movies released after 2000.

→ COUNT(1) is better than COUNT(*) because it only iterates over one column instead of all.

GROUP-BY :-



SELECT year, COUNT(year) FROM movies GROUP BY year;

using alias :- SELECT year, COUNT(year) year_count FROM movies GROUP BY year ORDER BY year_count;

Calculate this & store it in

Sorted based on that.

- GROUP BY group all the null values into one row.

Grouping odd year numbers :- SELECT year, COUNT(year) year_count FROM movies GROUP BY year WHERE year%2!=0 ORDER BY year_count;

HAVING :-

SELECT year, COUNT(year) year_count FROM movies GROUP BY year HAVING year_count>1000;

order of execution :- ① ② ③

→ HAVING is often used with GROUP BY, but it's not mandatory

SELECT name, movies FROM movies HAVING year>2000;

Same as

SELECT name, movies FROM movies WHERE year>2000;

→ HAVING is applied on groups, WHERE is applied on rows. HAVING is applied after grouping, WHERE is applied before grouping

→ HAVING will do tableSpace scan & WHERE will do tableSpace scan. WHERE is better when grouping is not required.

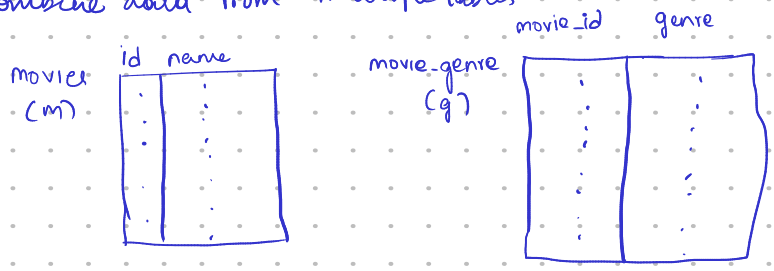
Order of keywords:-

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
      [HIGH_PRIORITY]
      [STRAIGHT_JOIN]
      [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
      SQL_NO_CACHE [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr ...]
    [FROM table_references
      [PARTITION partition_list]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}, ... [WITH ROLLUP]]
    [HAVING where_condition]
    [WINDOW window_name AS (window_spec)
        [, window_name AS (window_spec)] ...]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [INTO OUTFILE 'file_name'
        [CHARACTER SET charset_name]
        export_options
      | INTO DUMPFILE 'file_name'
      | INTO var_name [, var_name]]
    [FOR {UPDATE | SHARE} [OF tbl_name [, tbl_name] ...] [NOWAIT | SKIP LOCK
      | LOCK IN SHARE MODE]]
```

MySQL

JOINS :-

→ Combine data from multiple tables



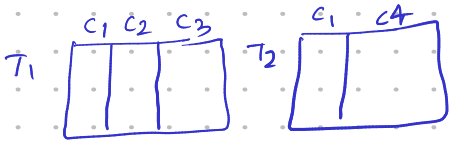

```
SELECT m.name, g.genre FROM movies m JOIN movie_genres g ON m.id=g.movie_id LIMIT 20;
```

m, g → alias

'id' is the common row in both tables.

The above query is called an inner join.

Natural Join :- When the column names are same in both tables, ON can be skipped.




```
SELECT * FROM T1 JOIN T2;
    is same as
SELECT * FROM T1 JOIN T2 ON T1.C1 = T2.C1;
    is same as
SELECT * FROM T1 JOIN T2 USING(C1);
```

Outer Joins :- Three types. (i) Left Outer Join (ii) Right outer join (iii) Full Outer Join.

Left :-

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1 = T2.C2;
```

↳ If values from C1 are missing in C2, they will be replaced with NULL.

Right :-

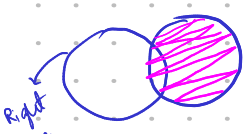
```
SELECT * FROM T1 RIGHT OUTER JOIN ON T1.C1 = T2.C2;
```

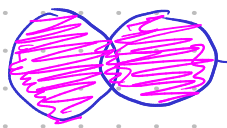
↳ At the end of result-set, leftover rows from T2 are added with NULL T1 values.

Full Outer Join :-

```
SELECT * FROM T1 FULL JOIN T2 ON T1.C1 = T2.C2;
```

All rows from both tables are added with NULLs





→ Full Outer Join.

→ INNER JOIN = JOIN | FULL OUTER JOIN = FULL JOIN | LEFT OUTER JOIN = LEFT JOIN | RIGHT OUTER JOIN = RIGHT JOIN

K-Way Join :-

```
# 3-way joins and k-way joins
SELECT(a.first_name, a.last_name FROM actors a JOIN roles r ON a.id=r.actor_id)(JOIN movies m on
m.id=r.movie_id) AND m.name='Officer 444';
```

• → 1 • → 2

→ Joins can be computationally expensive (Time consuming)

SubQueries :-

```
mysql> describe actors;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| id         | int(11)      | NO   | PRI | 0       |       |
| first_name | varchar(100) | YES  | MUL | NULL    |       |
| last_name  | varchar(100) | YES  | MUL | NULL    |       |
| gender     | char(1)      | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
4 rows in set (0.15 sec)

mysql> describe roles;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| actor_id | int(11)      | NO   | PRI | NULL    |       |
| movie_id | int(11)      | NO   | PRI | NULL    |       |
| role     | varchar(100) | NO   | PRI | NULL    |       |
+----------+--------------+------+-----+---------+-------+
3 rows in set (0.01 sec)

mysql> describe movies;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| id        | int(11)      | NO   | PRI | 0       |       |
| name      | varchar(100) | YES  | MUL | NULL    |       |
| year      | int(11)      | YES  |     | NULL    |       |
| rankscore | float        | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
4 rows in set (0.01 sec)
```

actors

roles

movies

```
SELECT first_name, last_name from actors WHERE id IN
    ( SELECT actor_id from roles WHERE movie_id IN
            (SELECT id FROM movies where name='Schindler's List)
);
```

③ ② ①

→ Order just like loops in programming.

General Syntax :-

```
# Syntax:
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
   (SELECT column_name [, column_name ]
   FROM table1 [, table2 ]
   [WHERE])
```

Operators that can be used :-

IN, NOT IN,

EXISTS (returns true if query returns one or more records or NULL)

ANY, ALL

ANY → if any of the subqueries meet the condition.

ALL → if all the subqueries meet the condition.

```
SELECT * FROM movies where rankscore >= ALL (SELECT MAX(rankscore) from movies);
# get all movies whose rankscore is same as the maximum rankscore.
```

9.9

example.

→ Nested Subqueries are much more readable than joins.

Correlated Sub query :-

(wikipedia)

```
SELECT employee_number, name
  FROM employees emp
  WHERE salary > (
    SELECT AVG(salary)
      FROM employees
      WHERE department = emp.department);
```

→ For every employee, the inner query is run.
As a result, the inner query is run multiple times.

→ inner query.
→ outer query.

→ The query evaluation time of JOIN is very high compared to that of subquery. But after join is performed, RDBMS performs indexing & it becomes faster.

→ Exists is intended to be used as a way to avoid counting.

```
--this statement needs to check the entire table
select count(*) from [table] where ...

--this statement is true as soon as one match is found
exists ( select * from [table] where ... )
```

Data Manipulation :-

SQL → Data Manipulations Language
→ Data Definitions Language.
→ Data Control Language.

→ SELECT , INSERT , UPDATE , DELETE.

→ INSERT INTO table_name(col_1, col_2, col_3) VALUES (val_1, val_2, val_3), (val_1_1, val_2_1, val_3_1), (val_1_2, val_2_2, val_3_2);

↳ multiple rows being inserted.
→ if col_1 is a primary key, we can't insert a new row with an already existing primary key value. It'll throw an error.

We can use nested subqueries to insert data from another table.

Sub-Q

```
INSERT INTO phone_book2
SELECT *
FROM   phone_book
WHERE  name IN ('John Doe', 'Peter Doe')
```

→ UPDATE table_name SET col1 = val1, col2 = val2 WHERE <CONDITION>;
Can be used with subqueries just like INSERT.
Multiple rows can be updated as well.

→ DELETE FROM table_name WHERE CONDITION;

→ TRUNCATE TABLE table_name;
↳ Delete all rows from table
Same as DELETE without WHERE clause.
Delete uses row lock whereas Truncate locks the whole table.
Rollback is possible in both cases.

Data Definitions Language :-

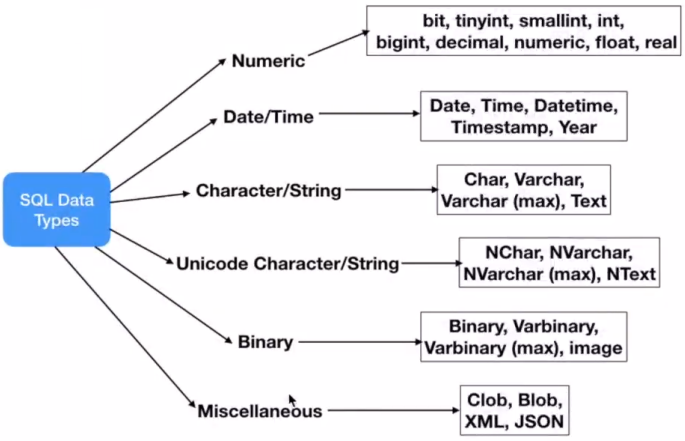
CREATE TABLE table_name (col_1 INT PRIMARY col_2 VARCHAR(50) NOT NULL);

- INT → col_1 INT
- column names → col_1, col_2
- PRIMARY → col_1 is the primary key.
- VARCHAR(50) → It is a char sequence of size ≤ 50.
- NOT NULL → Column value can't be NULL

Constraints :-

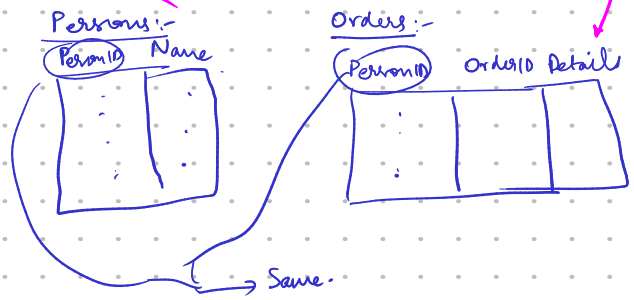
→ NOT NULL : Column value can't be NULL
→ UNIQUE :- Column values have to be unique. Can contain NULL values.
→ PRIMARY :- Combination of NOT NULL & UNIQUE.
→ FOREIGN :- Uniquely identifies a row in another table.
→ CHECK :- Ensures that all values follow a specific condition.
→ DEFAULT :- Sets a default value if a value isn't specified.
→ INDEX :- Used to create & retrieve data from the database very quickly. Indexes a column & allows us to access fast & perform ops fast.

SQL Datatypes :-



Foreign key :-

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



Modify Existing table :-

(ALTER) :- ADD , MODIFY , DROP

ALTER TABLE language ADD country VARCHAR(50);

ALTER TABLE language MODIFY country VARCHAR(60);

ALTER TABLE language DROP country;

Changing column name :- ALTER TABLE language CHANGE country origin VARCHAR(60);

old name ↑ new name ↑

Drop :-

DROP TABLE tablename;

↑ whole table is deleted. (including table structure)

DROP TABLE IF EXISTS tablename;

↑ drops table iff it exists.

TRUNCATE TABLE tablename; is same as DELETE FROM tablename;

↑ Deletes contents but not the table structure.

Data Control Language :-

→ for DB Admins.

* DCL control the access to a DB
* Not all RDBMS have DCL.
SQLite used in smartphones does not have DCL as it does not have user management

GRANT → Gives permissions

REVOKE → Revokes permissions.