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)
```

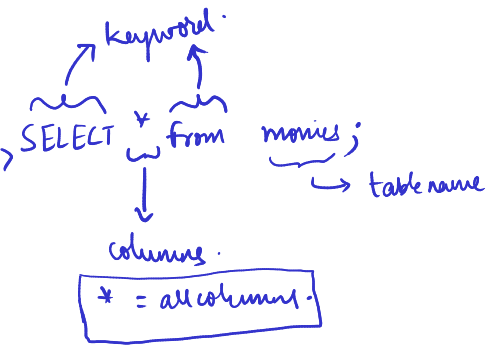
Press Esc to exit full screen

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

mysql>
```



→ 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 slower than SELECT — — specific columns.

(this order can be anything. Does it 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 value & 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 :-

column names — table name — First 10 results — descending order

```
mysql> SELECT name,rankscore,year FROM movies ORDER BY year DESC LIMIT 10;
+----------------------------------------+-----------+------+
| name                                   | rankscore | year |
+----------------------------------------+-----------+------+
| Harry Potter and the Half-Blood Prince |      NULL | 2008 |
| Tripoli                                |      NULL | 2007 |
| War of the Red Cliff, The              |      NULL | 2007 |
| Rapunzel Unbraided                     |      NULL | 2007 |
| Spider-Man 3                           |      NULL | 2007 |
| Untitled Star Trek Prequel             |      NULL | 2007 |
| DragonBall Z                           |      NULL | 2007 |
| Harry Potter and the Order of the Phoenix |   NULL | 2007 |
| Andrew Henry's Meadow                  |      NULL | 2006 |
| American Rain                          |      NULL | 2006 |
+----------------------------------------+-----------+------+
10 rows in set (0.15 sec)
```

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 conditional 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.

_ :- Atmost 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)
```

→ Took all the columns.

→ overall number of rows.

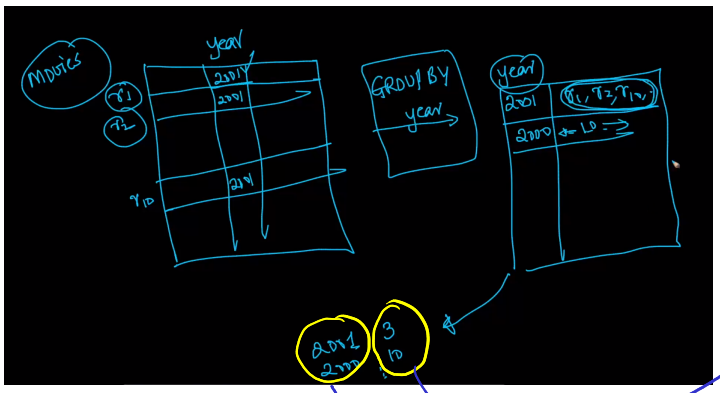
count(column_name) does not include NULL values.
COUNT(*) does.

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;

Calculates this & stores it in

Sorted based on that.

- GROUP BY groups 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 table space scan. WHERE is better when grouping is not required.

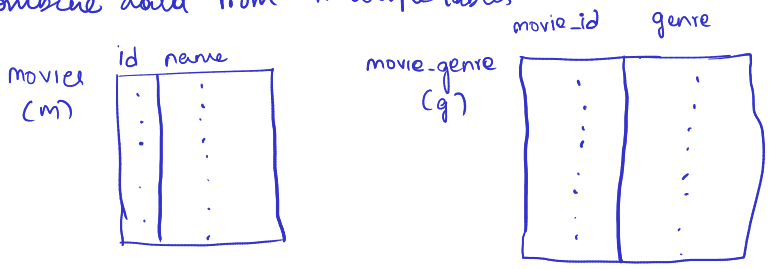
Order of keywords :-

```
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



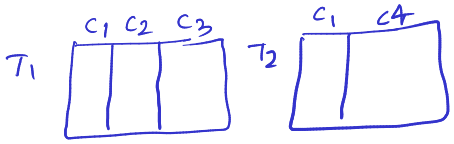

```
SELECT m.name, g.genre FROM movies m JOIN movie_genres g ON m.id=g.movie_id LIMIT 20;
```

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);
```