USE imdb; <DB-NAME>

SHOW TABLES;

DESCRIBE movies; <TABLE-NAME>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SELECT \* FROM movies; # \* means all columns selection from respective table

# more data transfer

#result-set: a set of rows that form the result of a query along with column-names and meta-data.

SELECT name,year FROM movies;

SELECT rankscore,name FROM movies;

#row order same as the one in the table

how to stop or exit from the query,is it possible - Ctrl + C or

kill "number from first col";

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LIMIT: No of rows we need the datas to limit to.

SELECT name,rankscore FROM movies LIMIT 20;

SELECT name,rankscore FROM movies LIMIT 20 OFFSET 40;

#offset means we need to ignore 1st 40 rows and then print next 20 rows after it. Used in app/website for listing down no of rows page wise.

Notes -  if I run this "SELECT id,name,rankscore FROM movies LIMIT 40;" - This gives me the first 40 rows with their ID's starting from 1.  
But when I run "SELECT id,name FROM movies LIMIT 40;" - This prints the 40 rows whose ID's starts from 376764. WHY?

Ans - The 'select' command is non-deterministic in the nature that it does not always return the same set of rows everytime you perform a query. The 'select' command can be used with 'order by' if an order is desired.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ORDER BY:

# list recent movies first

SELECT name,rankscore,year FROM movies ORDER BY year DESC LIMIT 10;

# default:ASC

SELECT name,rankscore,year FROM movies ORDER BY year LIMIT 10;

# the output row order maynot be same as the one in the table due to query optimzier and internal data-structres/indices.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DISTINCT:

# list all genres of

SELECT DISTINCT genre FROM movies\_genres;

# multiple-column DISTINCT

SELECT DISTINCT first\_name, last\_name FROM directors;

Notes –

Let's consider a simple example.  
SELECT DISTINCT first\_name,last\_name FROM directors ORDER BY first\_name;  
Assume first\_name contains ['A','B','C'] and last\_name contains ['D','E','D']  
1.) SELECT DISTINCT last\_name : Result : ['D','E"] (These are the distinct values)  
2.) SELCT DISTINCT first\_name : Result : ['A','B','C'] (These are the distinct values)  
3.) SELECT DISTINCT first\_name,last\_name : Result: [ ('A','D') , ('B','E') , ('C','D') ]  Note: ('A','D') and ('C','D') are distinct. To say a tuple (x,y) is equal to (a,b) it must satisfy x=a and y=b.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

WHERE:

# list all movies with rankscore>9

SELECT name,year,rankscore FROM movies WHERE rankscore>9 ;

SELECT name,year,rankscore FROM movies WHERE rankscore>9 ORDER BY rankscore DESC LIMIT 20;

# Condition's outputs: TRUE, FALSE, NULL

# Comparison Operators: = , <> or != , < , <= , >, >=

SELECT \* FROM movies\_genres WHERE genre = 'Comedy';

SELECT \* FROM movies\_genres WHERE genre <> 'Horror';

NULL => doesnot-exist/unknown/missing

# "=" doesnot work with NULL, will give you an empty result-set.

SELECT name,year,rankscore FROM movies WHERE rankscore = NULL;

SELECT name,year,rankscore FROM movies WHERE rankscore IS NULL LIMIT 20;

SELECT name,year,rankscore FROM movies WHERE rankscore IS NOT NULL LIMIT 20;

Notes –

In SQL, queries are processed in the following order -  
1. From  
2. Where  
3. Group By

---- Aggregate function if it exists then evaluated before having clause  
4. Having  
5. Select  
6. Order By  
7. Offset, Fetch, Limit  
  
The main query clauses appear in the figure in the order that you are required to type them, and the step numbers represent their logical query processing order. Even though you type the SELECT clause first, the FROM clause is the one that is logically processed first.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# LOGICAL OPERATORS: AND, OR, NOT, ALL, ANY, BETWEEN, EXISTS, IN, LIKE, SOME

# website search filters

SELECT name,year,rankscore FROM movies WHERE rankscore>9 AND year>2000;

SELECT name,year,rankscore FROM movies WHERE NOT year<=2000 LIMIT 20; # year >2000

SELECT name,year,rankscore FROM movies WHERE rankscore>9 OR year>2007;

# will discsuss about ANY and ALL when we discuss sub-queries

SELECT name,year,rankscore FROM movies WHERE year BETWEEN 1999 AND 2000;

#between is inclusive: year>=1999 and year<=2000

SELECT name,year,rankscore FROM movies WHERE year BETWEEN 2000 AND 1999;

#lowvalue <= highvalue else you will get an empty result set

SELECT director\_id, genre FROM directors\_genres WHERE genre IN ('Comedy','Horror');

# same as genre='Comedy' OR genre='Horror'

SELECT name,year,rankscore FROM movies WHERE name LIKE 'Tis%';

# % => wildcard character to imply zero or more characters

SELECT first\_name, last\_name FROM actors WHERE first\_name LIKE '%es';

# first name ending in 'es'

SELECT first\_name, last\_name FROM actors WHERE first\_name LIKE '%es%';

#first name contains 'es'

SELECT first\_name, last\_name FROM actors WHERE first\_name LIKE 'Agn\_s';

# '\_' implies exactly one character. Eg Agnes,Agnas etc.

# If we want to match % or \_, we should use the backslash as the escape character: \% and \\_

SELECT first\_name, last\_name FROM actors WHERE first\_name LIKE 'L%' AND first\_name NOT LIKE 'Li%';

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Aggregate functions: Computes a single value on a set of rows and returns the aggreagate

COUNT, MIN, MAX, SUM, AVG

SELECT MIN(year) FROM movies;

SELECT MAX(year) FROM movies;

SELECT COUNT(\*) FROM movies; # here ‘\*’ means take all the columns. And result give total no of rows.

SELECT COUNT(\*) FROM movies where year>2000;

SELECT COUNT(year) FROM movies;

Notes –

Count(\*) will consider all the rows in the table and return the count, when we use count(column\_name), it will return only not-null count . Please correct me.

Yes, COUNT counts values, since null is not a value it does not get counted. So,  
COUNT(\*)  
will count all rows  
COUNT(columnname)  
will count all rows, except those rows where columnname IS NULL.

The 1 in COUNT(1) is just a literal constant expression, it has nothing to do with Column 1 of the table. It will give the same answer as COUNT(\*) i.e. SELECT COUNT(1) counts the number of rows in a result set where a constant 1 is being returned from EACH and EVERY row, which is exactly the same as “SELECT COUNT(\*)"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

GROUP-BY – creates a resultset or view which looks something like this ex.

Year1 ={row1,row3,…} for which rows year1 is present in the year field as mentioned below in movies table.

# find number of movies released per year

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

# here group by will execute first which groups the rows against the year and then on this grouping the aggregate function like count will execute to count the number of rows. Eg.

2001={r1,r2,r3} -🡪 will return count(year)=3.

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

SELECT year, COUNT(year) year\_count FROM movies GROUP BY year ORDER BY year\_count;

# year\_count is an alias.

# often used with COUNT, MIN, MAX or SUM.

# if grouping columns contain NULL values, all null values are grouped together.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

HAVING:

# Print years which have >1000 movies in our DB [Data Scientist for Analysis]

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

# specify a condition on groups using HAVING.

Order of execution:

1. GROUP BY to create groups

2. apply the AGGREGATE FUNCTION

3. Apply HAVING condition.

# often used along with GROUP BY. Not Mandatory.

SELECT name, year FROM movies HAVING year>2000;

## HAVING without GROUP BY is same as WHERE

SELECT year, COUNT(year) year\_count FROM movies WHERE rankscore>9 GROUP BY year HAVING year\_count>20;

# HAVING vs WHERE

## WHERE is applied on individual rows while HAVING is applied on groups.

## HAVING is applied after grouping while WHERE is used before grouping.

Notes –

here when i use having instead of where it give error  
mysql> select name from movies where year=2008;  
+----------------------------------------+  
| name |  
+----------------------------------------+  
| Harry Potter and the Half-Blood Prince |  
+----------------------------------------+  
1 row in set (1.28 sec)  
  
mysql> select name from movies having year=2008;  
ERROR 1054 (42S22): Unknown column 'year' in 'having clause'

Ans –

If you want to write 'HAVING' clause, you have to get that column as well from table in SELECT statement.

select \* from the movies having year = 2008;  
to get all info.  
+--------+----------------------------------------+------+-----------+  
| id | name | year | rankscore |  
+--------+----------------------------------------+------+-----------+  
| 139653 | Harry Potter and the Half-Blood Prince | 2008 | NULL |  
+--------+----------------------------------------+------+-----------+

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

JOINs:

#combine data in multiple tables

# For each movie, print name and the genres

SELECT m.name, g.genre from movies m JOIN movies\_genres g ON m.id=g.movie\_id LIMIT 20;

# table aliases: m and g

# natural join: a join where we have the same column-names across two tables.

#T1: C1, C2

#T2: C1, C3, C4

SELECT \* FROM T1 JOIN T2;

SELECT \* FROM T1 JOIN T2 USING (C1);

# returns C1,C2,C3,C4

# no need to use the keyword "ON"

# Inner join (default) vs left outer vs right outer vs full-outer join.

T1: C1, C2, C3

SELECT m.name, g.genre from movies m LEFT JOIN movies\_genres g ON m.id=g.movie\_id LIMIT 20;

#LEFT JOIN or LEFT OUTER JOIN

#RIGHT JOIN or RIGHT OUTER JOIN

#FULL JOIN or FULL OUTER JOIN

#JOIN or INNER JOIN

# NULL for missing counterpart rows.

# 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';

#Practical note about joins: Joins can be expensive computationally when we have large tables.

Note –

In the video,u said indexing is used to speed up the joining process but as much as I know about indexing, it is used to accelerate the fetching process of the data based on some conditions, using some advanced data structure s like B/B++ tree. Also, indexing is done multiple attributes, so as to serve its purpose on the fly. Now coming to joining, I think there is NO way you could combine the tables without actually peeking across the tables because that's how the nature of the problem is!  
I think we can speed up join is do pre-joining the tables based on user's recent queries and caching the stuff & building indexing on top of it is the only way we can speed up the joining process

If your Database does not support FULL JOIN (MySQL does not support FULL JOIN), then you can use **UNION ALL** clause to combine these two JOINS as shown below.

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

UNION ALL

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Sub-Queries or Nested Queries or Inner Queries

# List all actors in the movie Schindler's List

#https://www.imdb.com/title/tt0108052/fullcredits/?ref\_=tt\_ov\_st\_sm

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)

);

# Syntax:

SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

WHERE column\_name OPERATOR

(SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

[WHERE])

# first the innner query is executed and then the outer query is executed using the output values in the inner query

# IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL, Comparison operators

#EXISTS returns true if the subquery returns one or more records or NULL

# ANY operator returns TRUE if any of the subquery values meet the condition.

# ALL operator returns TRUE if all of the subquery values 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.

# e.g: rankscore <> ALL(...)

# <https://en.wikipedia.org/wiki/Correlated_subquery> -

**SELECT** employee\_number, name

**FROM** employees emp

**WHERE** salary > (

**SELECT** **AVG**(salary)

**FROM** employees

**WHERE** department = emp.department)

May increase computational cost if query optimizer doesn’t keep the avg once calculated in some temporary location for each department calculated once.

Note –

In most cases, JOINs are faster than sub-queries and it is very rare for a sub-query to be faster.  
In JOINs RDBMS can create an execution plan that is better for your query and can predict what data should be loaded to be processed and save time, unlike the sub-query where it will run all the queries and load all their data to do the processing.  
The good thing in sub-queries is that they are more readable than JOINs that's why most new SQL people prefer them. it is the easy way but when it comes to performance, JOINS are better in most cases even though they are not hard to read too.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Data Manipulation Language: SELECT, INSERT, UPDATE, DELETE

INSERT INTO movies(id, name, year, rankscore) VALUES (412321, 'Thor', 2011, 7);

INSERT INTO movies(id, name, year, rankscore) VALUES (412321, 'Thor', 2011, 7), (412322, 'Iron Man', 2008, 7.9), (412323, 'Iron Man 2', 2010, 7);

# INSERT FROM one table to another using nested sub query: <https://en.wikipedia.org/wiki/Insert_(SQL)#Copying_rows_from_other_tables>

**INSERT** **INTO** phone\_book2

**SELECT** \*

**FROM** phone\_book

**WHERE** name **IN** ('John Doe', 'Peter Doe')

**INSERT** **INTO** phone\_book

**SELECT** 'John Doe', '555-1212' **FROM** DUAL

**UNION** **ALL**

**SELECT** 'Peter Doe','555-2323' **FROM** DUAL

Notes –

Another way to insert value using ('&') in SQL,  
> insert into dept\_student\_17 values('&student\_id','&student\_name','&joindate  
','&fees');  
###only press ENTER button to automatically inserted values.  
Enter value for student\_id: 1  
Enter value for student\_name: niddhi  
Enter value for joindate: 01-june-2014  
Enter value for fees: 10000

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# UPDATE Command

UPDATE <TableName> SET col1=val1, col2=val2 WHERE condition

UPDATE movies SET rankscore=9 where id=412321;

# Update multiple rows also.

# Can be used along with Sub-queries.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#DELETE

DELETE FROM movies WHERE id=412321;

# Remove all rows: TRUNCATE TABLE TableName;

# Same as selete without a WHERE Clause.

Note –

a DML statement can affect only one table. So, you'd need to run multiple statements to delete certain IDs from multiple tables

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Data Definition Language - all DDL commands include the keyword ‘TABLE’ in it.

CREATE TABLE language ( id INT PRIMARY, lang VARCHAR(50) NOT NULL);

# Datatypes: https://www.journaldev.com/16774/sql-data-types

# Constraints: https://www.w3schools.com/sql/sql\_constraints.asp

NOT NULL - Ensures that a column cannot have a NULL value

UNIQUE - Ensures that all values in a column are different

PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table

FOREIGN KEY - Uniquely identifies a row/record in another table

CHECK - Ensures that all values in a column satisfies a specific condition

DEFAULT - Sets a default value for a column when no value is specified

INDEX - Used to create and retrieve data from the database very quickly

Notes –

the Syntax that is used to Create TABLE is giving Error,  
The Correct Syntax is:  
CREATE TABLE language (id INT PRIMARY KEY, lang VARCHAR(50) NOT NULL);

Just adding information for knowledge:--  
  
  
  
CREATE TABLE Language (id int(10) **UNSIGNED AUTO\_INCREMENT**PRIMARY KEY,lang VARCHAR(50) NOT NULL);

#### **UNSIGNED :-**Used for number types, limits the stored data to positive numbers and zero

### **AUTO\_INCREMENT:-** automatically increases the value of the field by 1 each time a new record is added ( So no data duplication problem).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ALTER: ADD, MODIFY, DROP

ALTER TABLE language ADD country VARCHAR(50);

ALTER TABLE language MODIFY country VARCHAR(60);

ALTER TABLE langauge DROP country;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Removes both the table and all of the data permanently.

DROP TABLE Tablename;

DROP TABLE TableName IF EXISTS; #means we will drop the table only when it exists.

#https://dev.mysql.com/doc/refman/8.0/en/drop-table.html

TRUNCATE TABLE TableName;

# as discussed earlier same as DELETE FROM TableName;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Data Control Language for DB Admins.

https://en.wikipedia.org/wiki/Data\_control\_language

https://dev.mysql.com/doc/refman/8.0/en/grant.html

https://dev.mysql.com/doc/refman/8.0/en/revoke.html