

Why SQL?

(Why) SQL?

→ SQL: Structured Query language (1970's IBM)
 { Relational DB

→ Standard way to query/obtain/add/delete/modify
list •

DB ↗ T₁ →
 ↗ T₂ →
 ↗ T₃ →
 ↗ T₄ →



A

(Why) SQL?

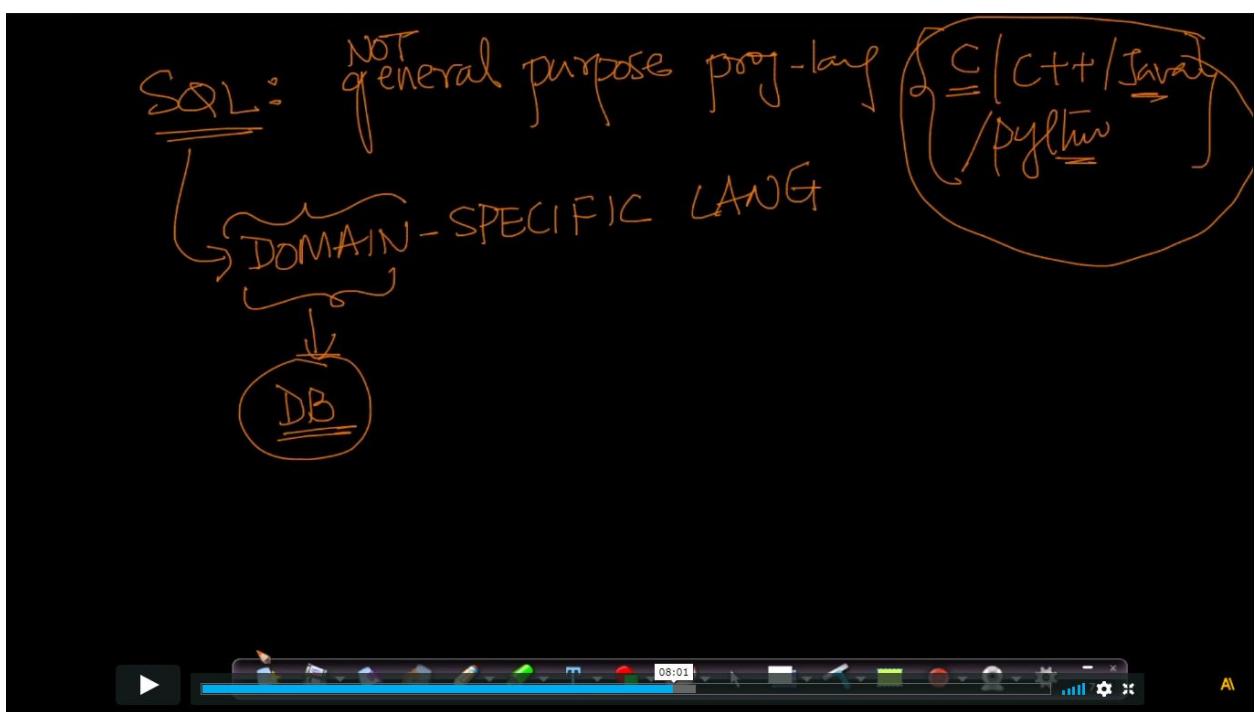
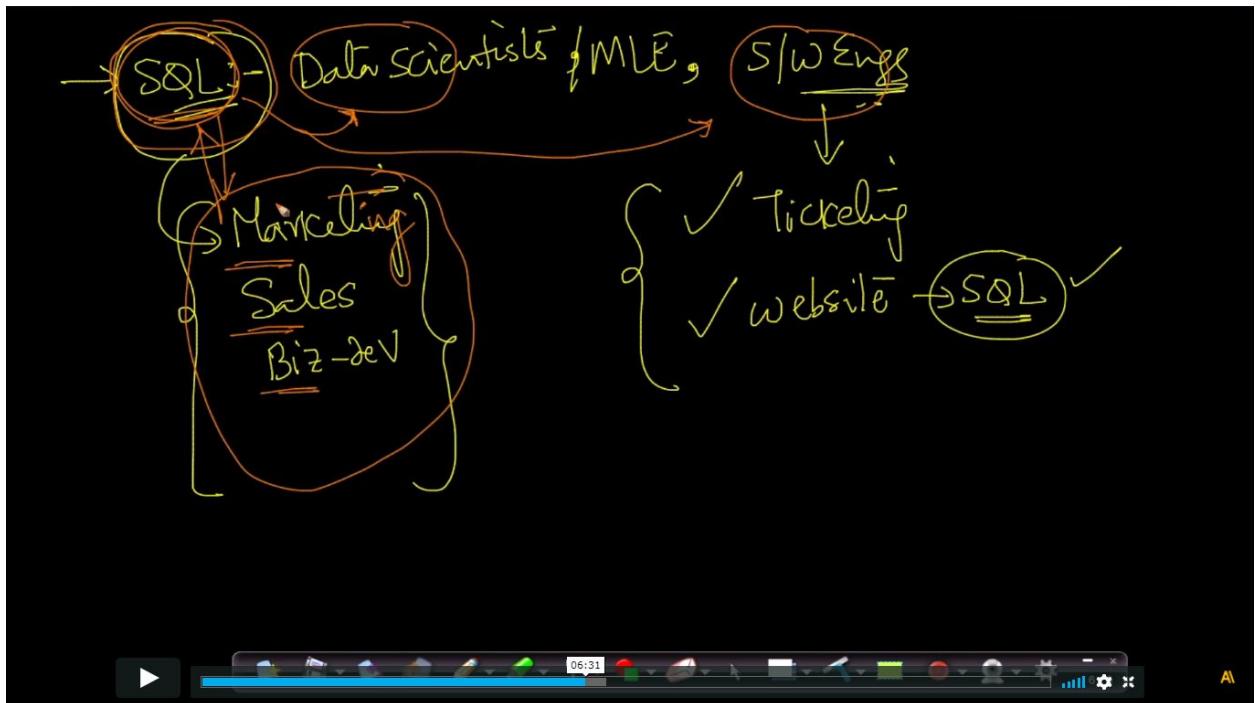
→ SQL: Structured Query language (1970's IBM)
 { Relational DB

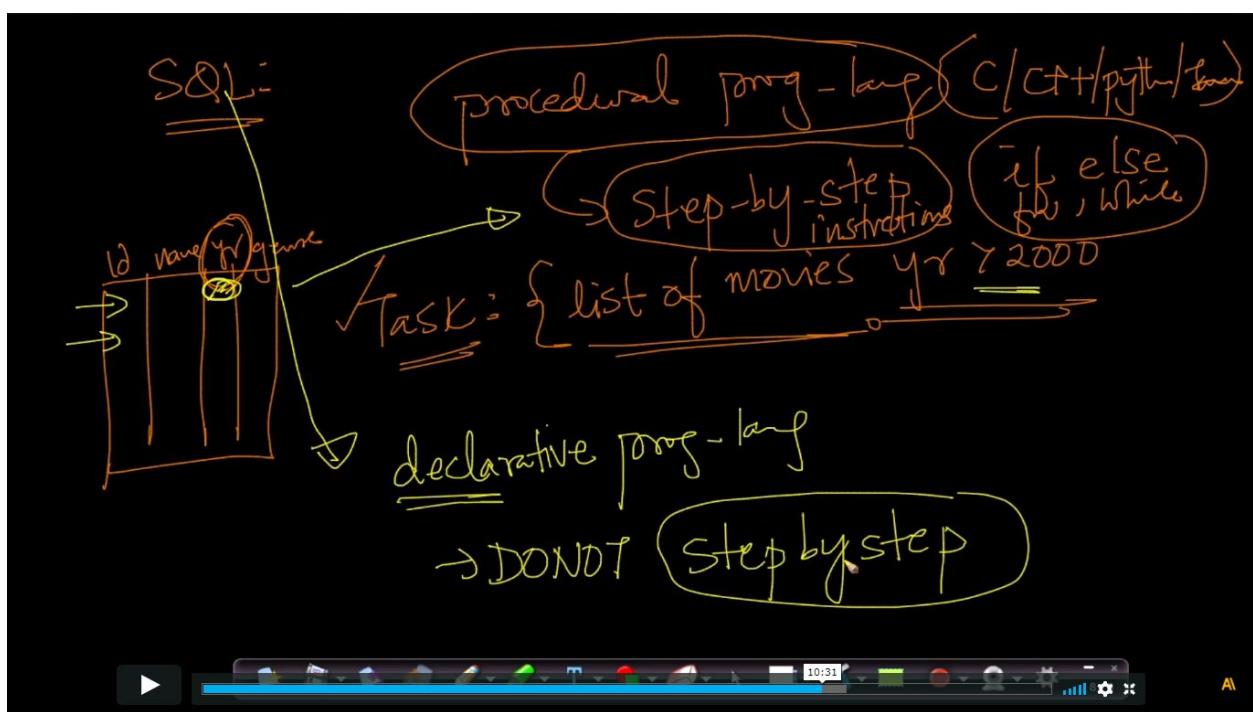
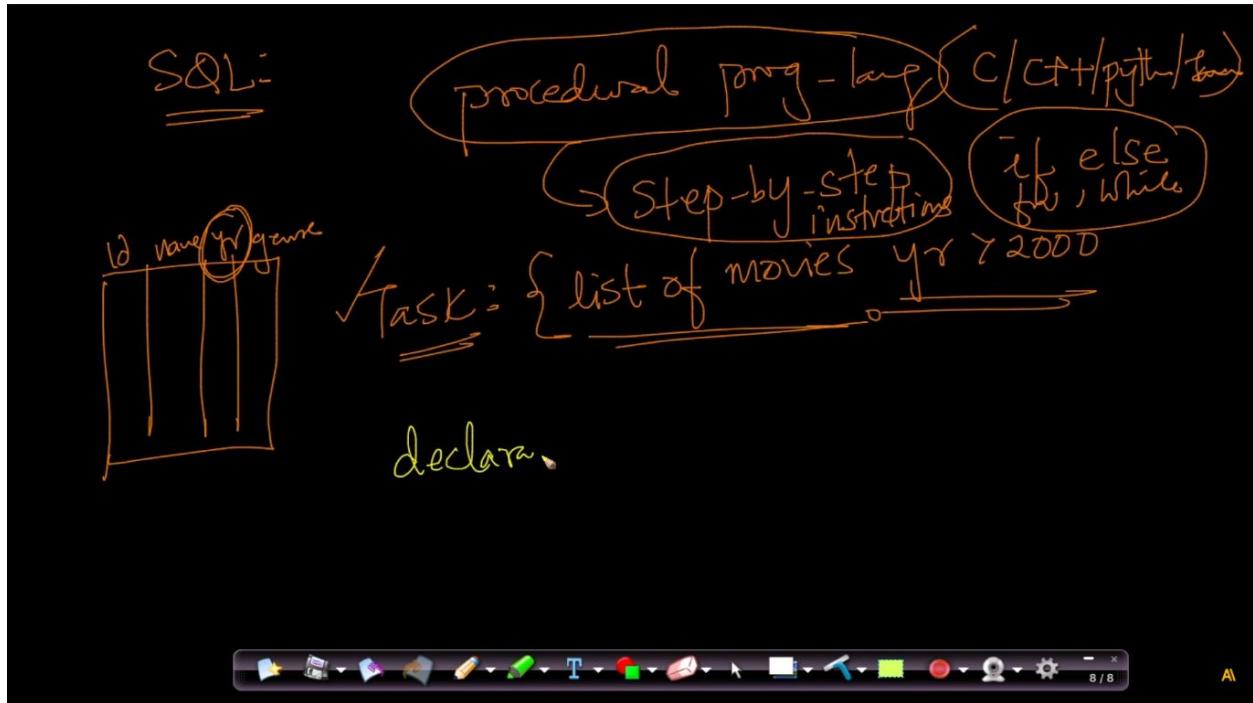
→ Standard way to query/obtain/add/delete/modify
 { list of movies
 { year > 2000

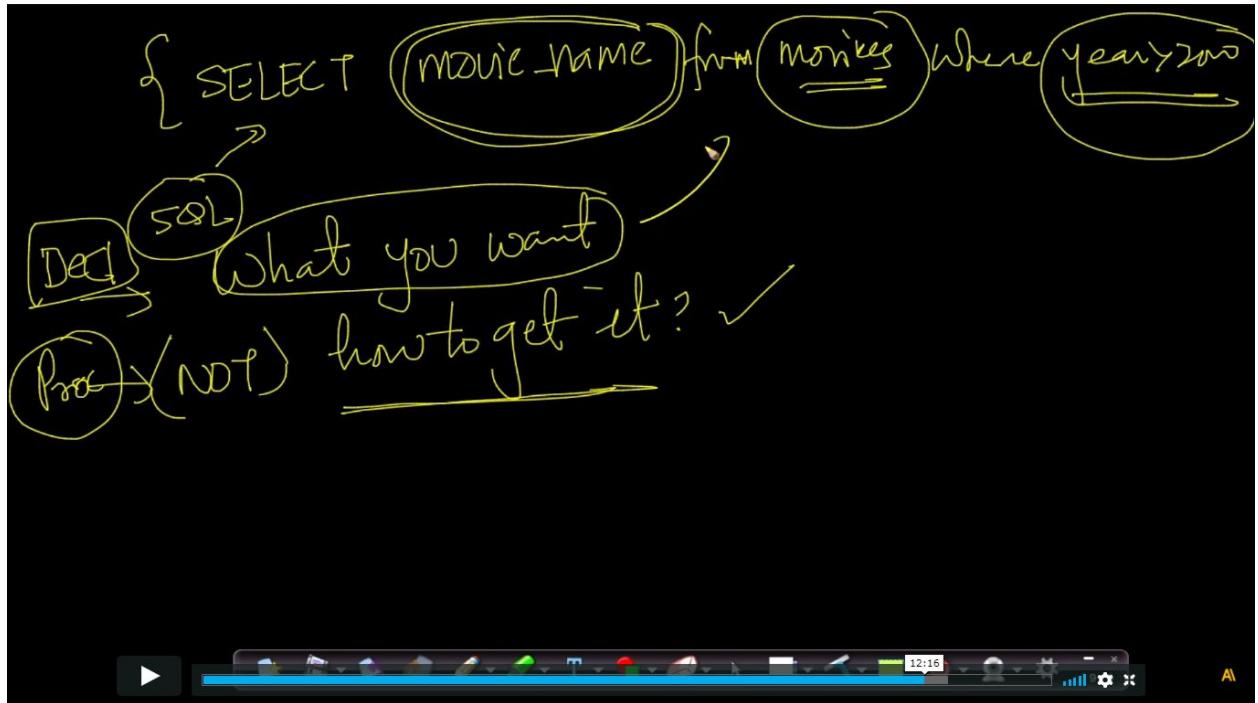
DB ↗ T₁ →
 ↗ T₂ →
 ↗ T₃ →
 ↗ T₄ →



A



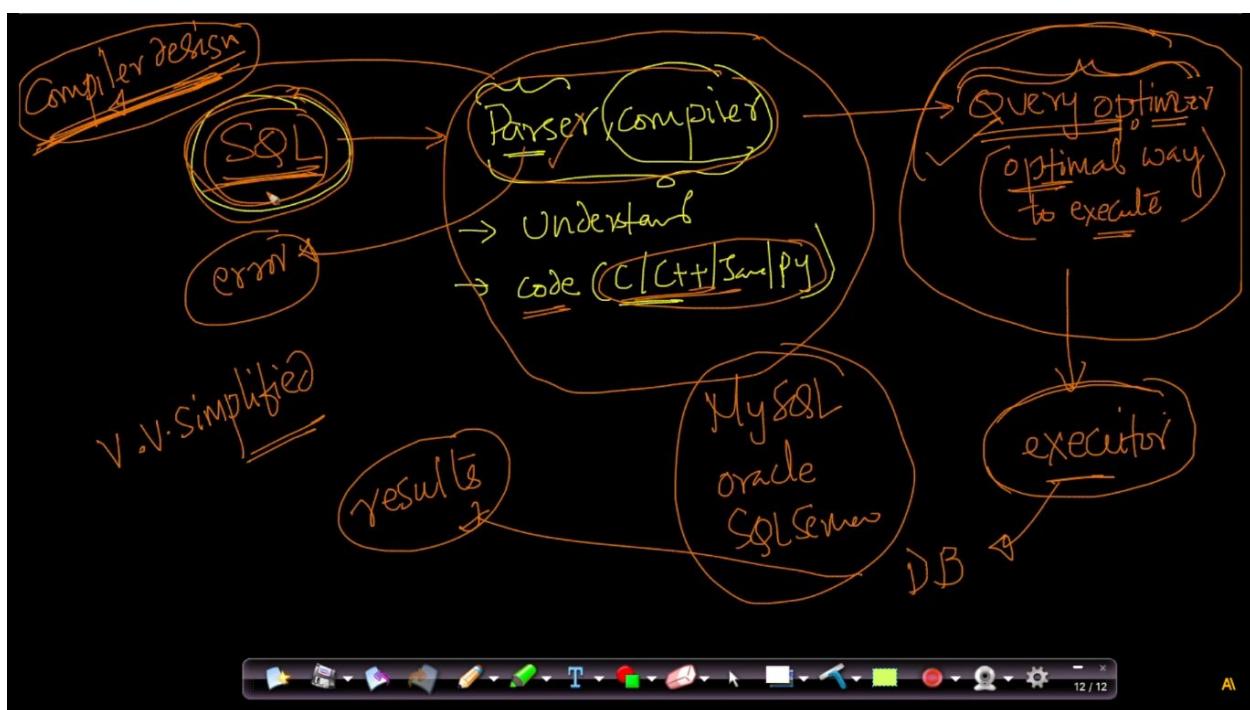
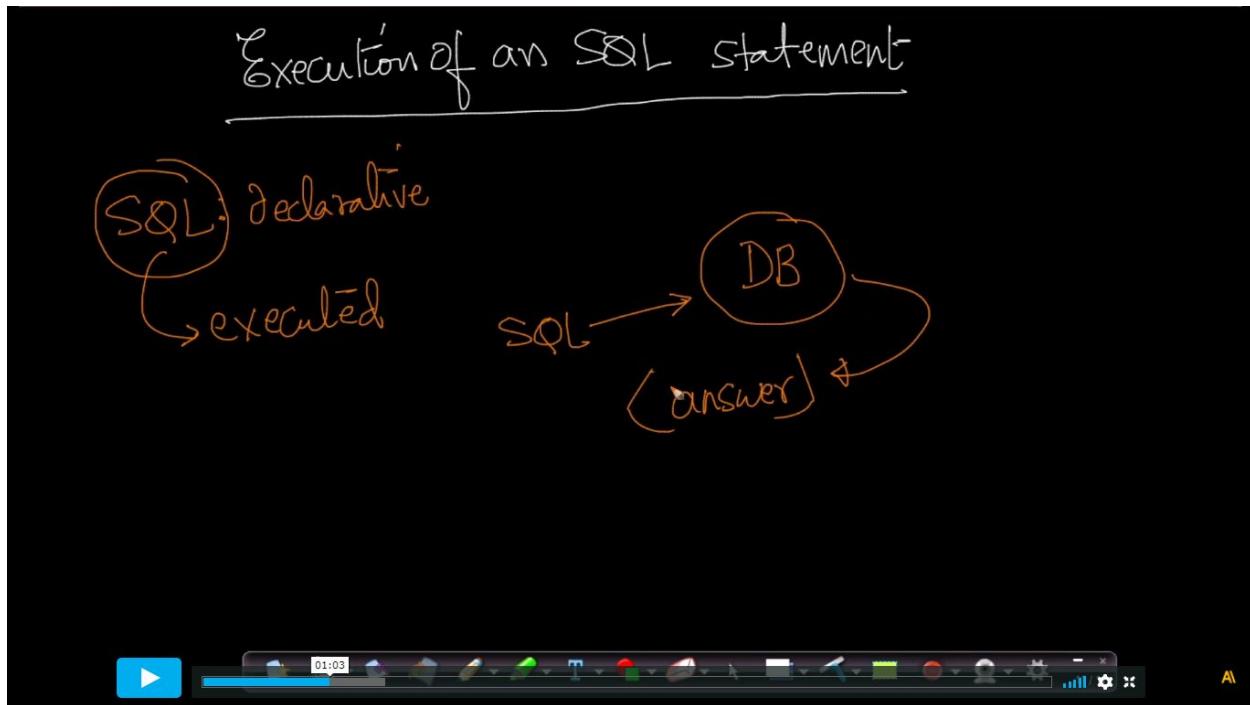




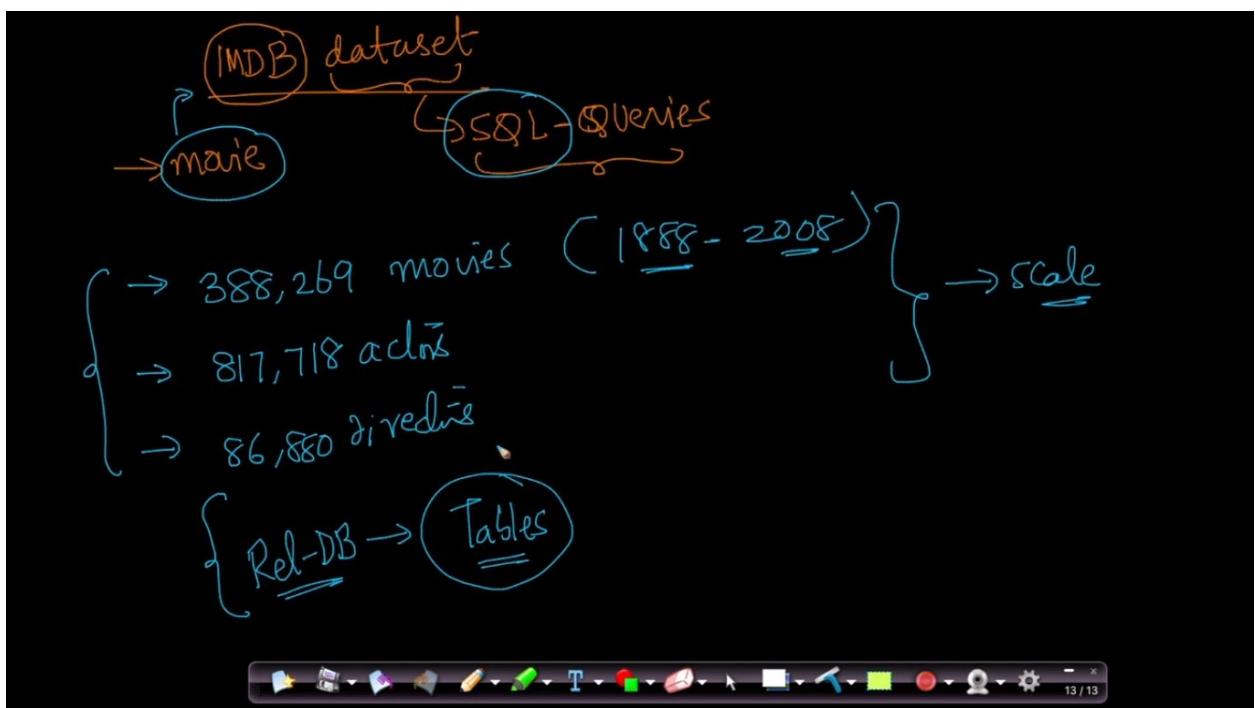
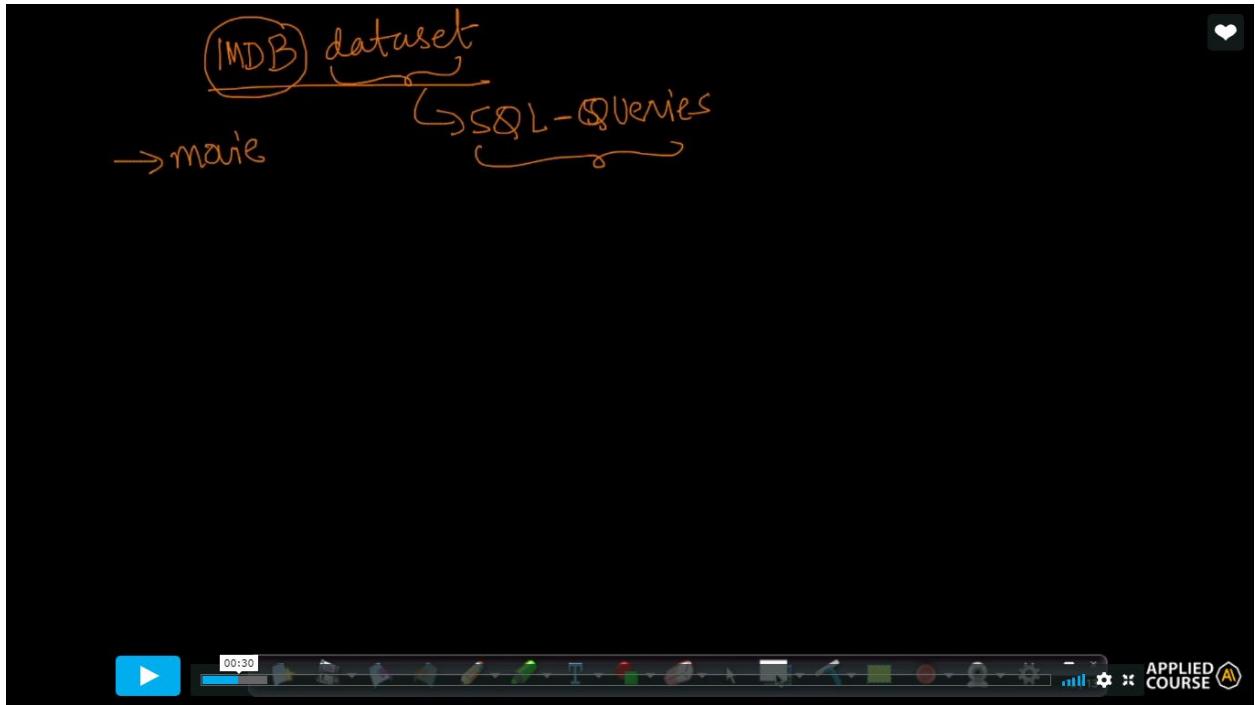
{ Super simple, easy
Very powerful

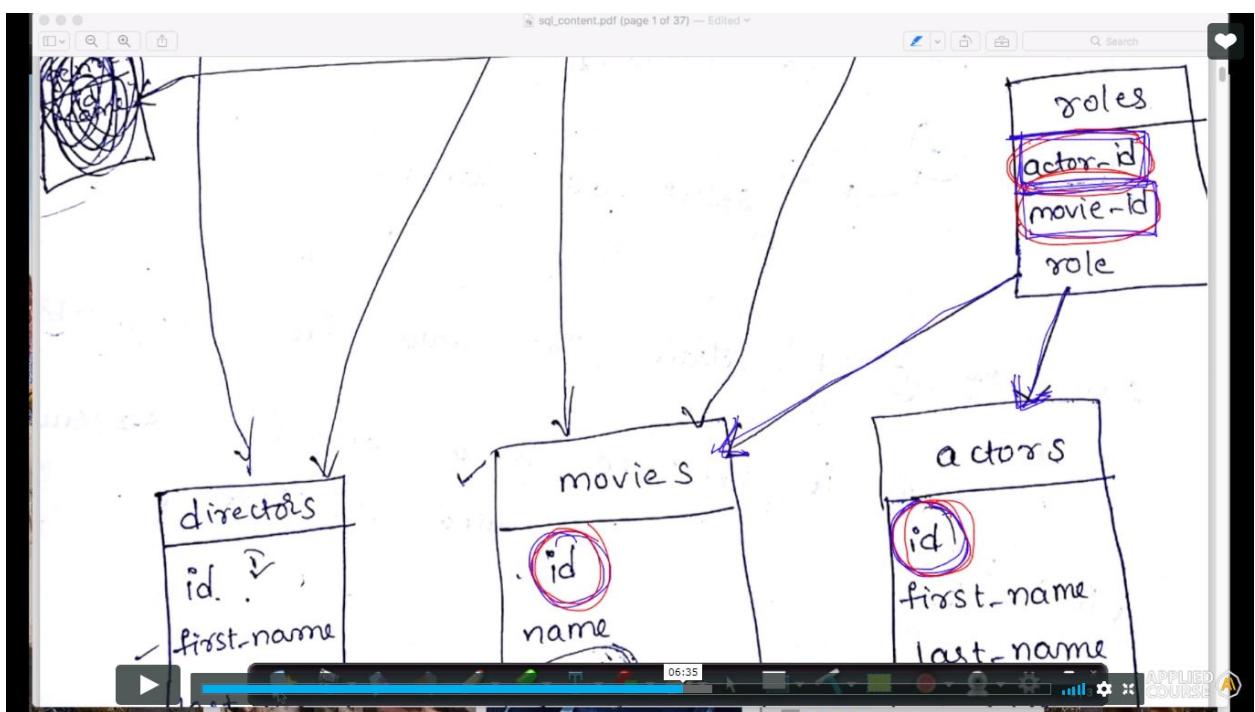
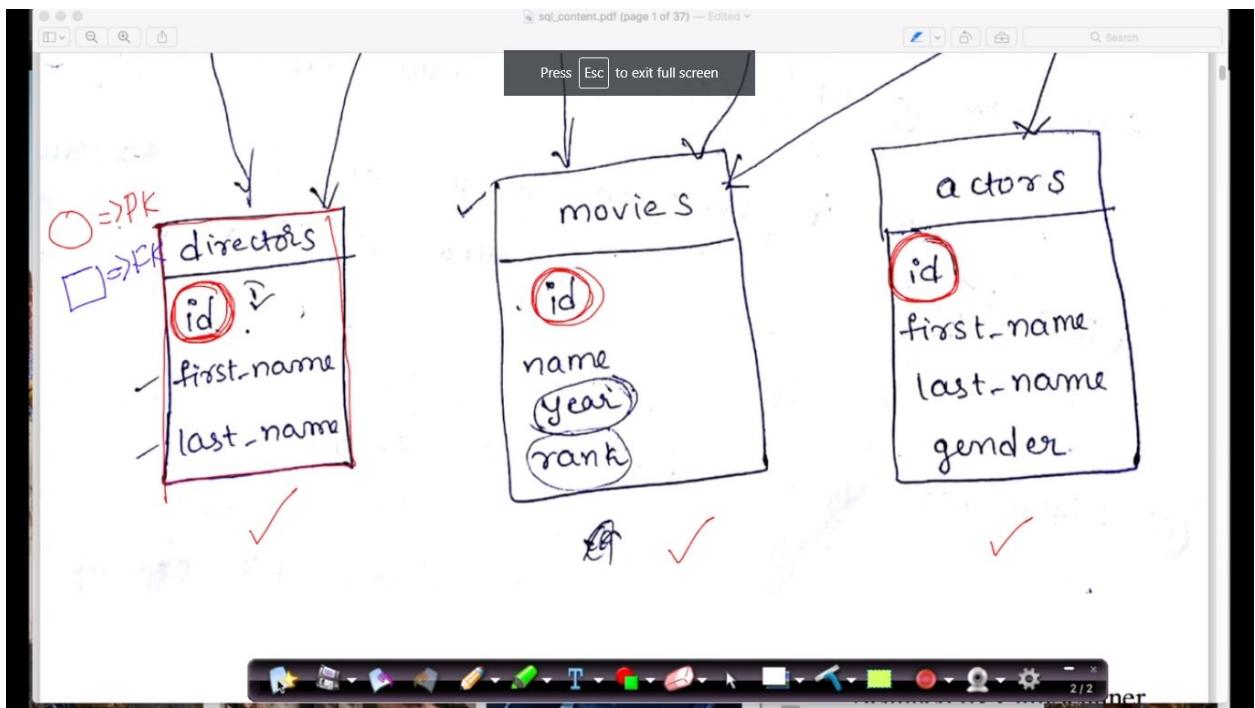


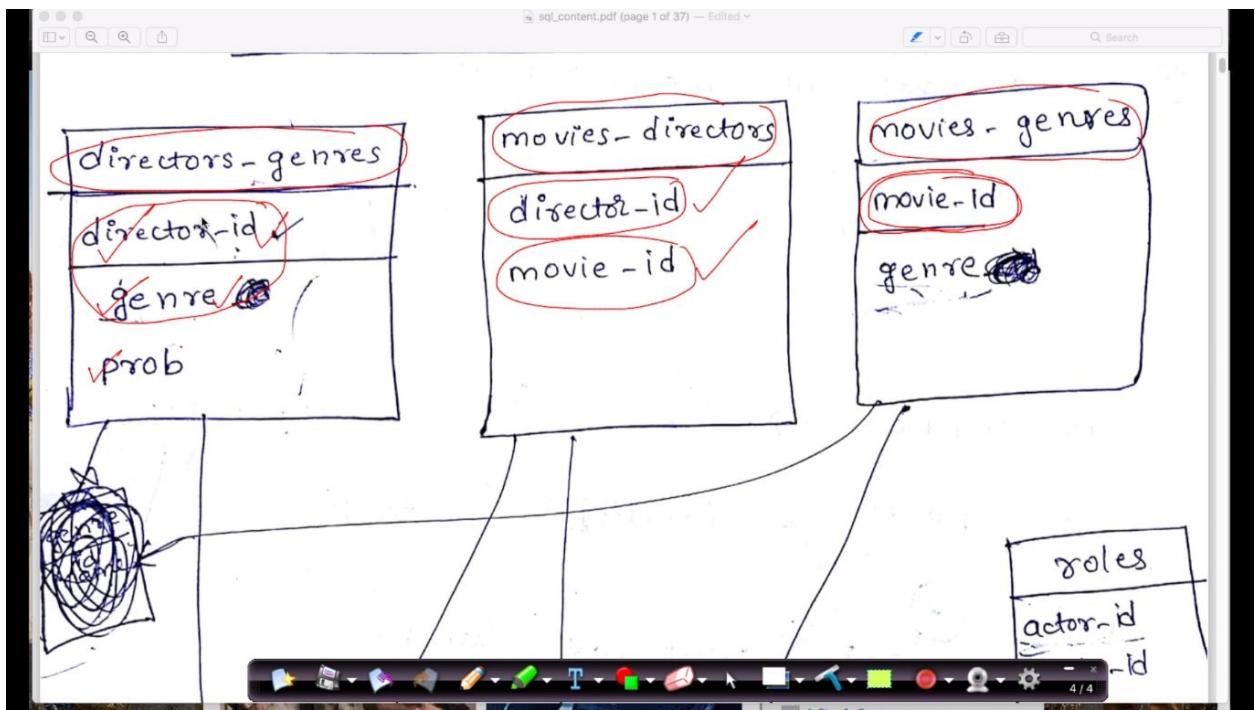
Execution of an SQL statement.

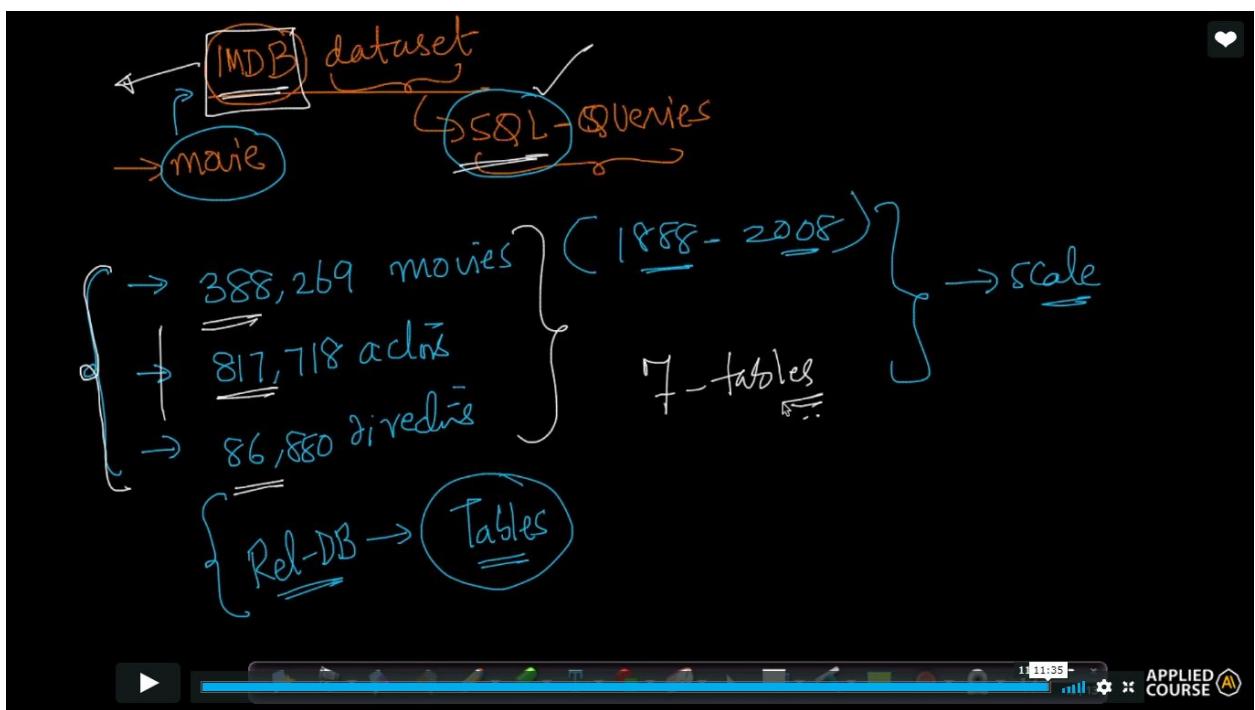
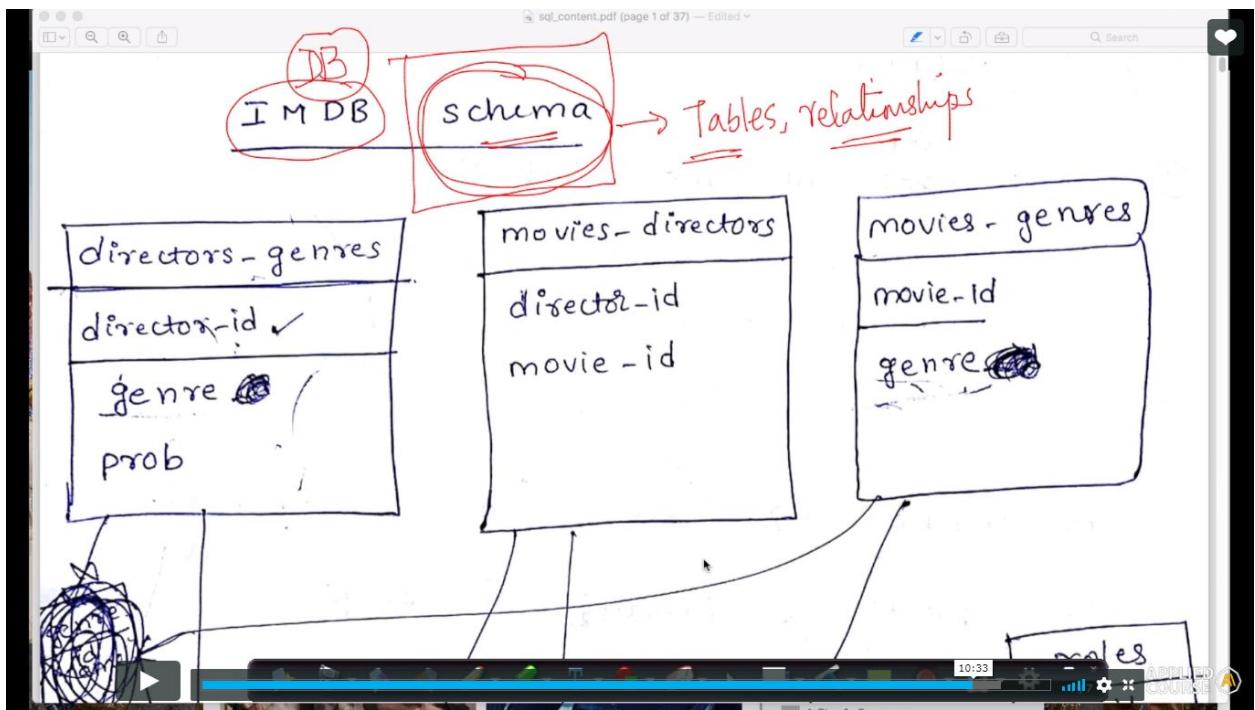


IMDB dataset









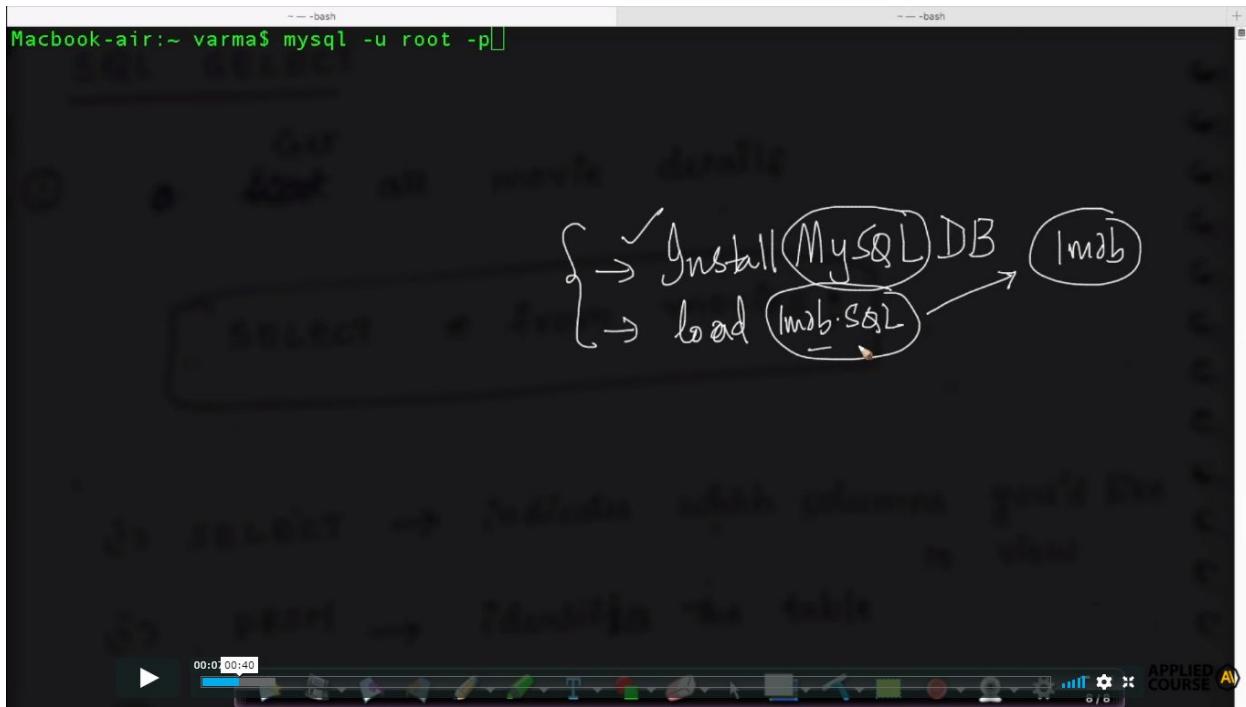
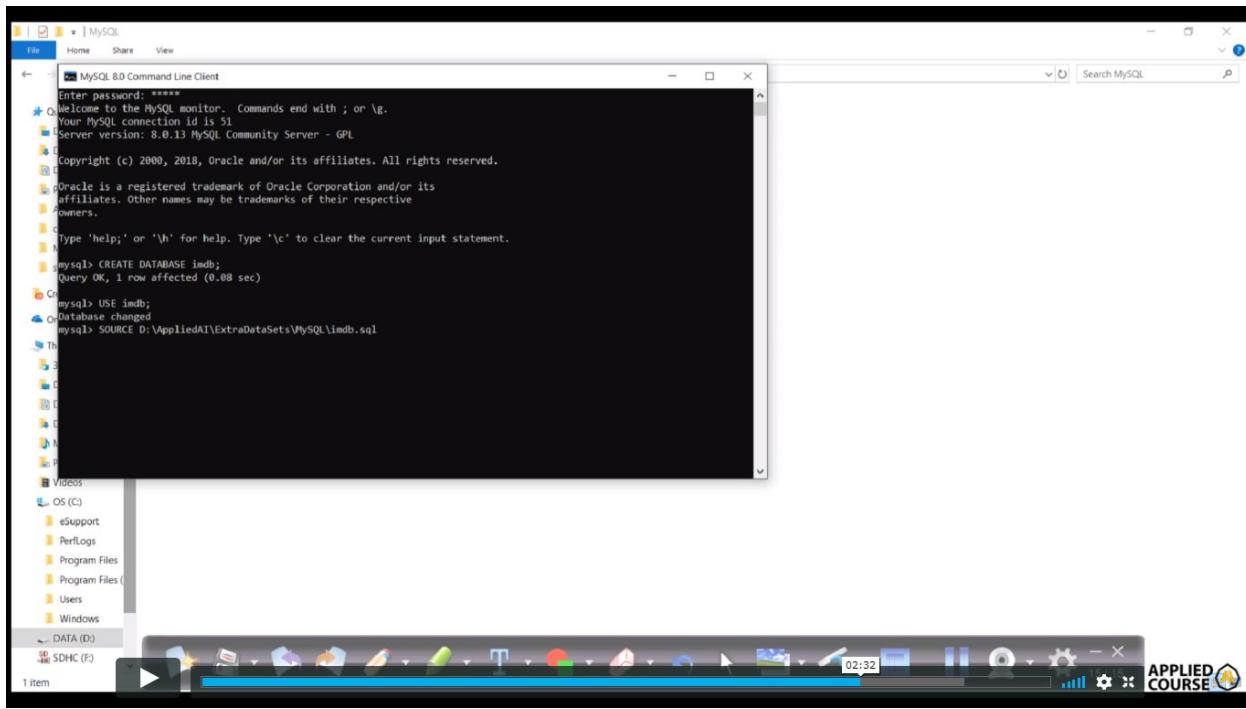
Installing MySQL

Refer:<https://dev.mysql.com/downloads/windows/installer/8.0.html>
<https://drive.google.com/open?id=1rrqZz-ddvYGvbXoP-SH767pgY06B7YHk>

Load IMDB data.

```
CREATE DATABASE imdb  
USE imdb  
SOURCE location
```

USE, DESCRIBE, SHOW TABLES



```
Macbook-air:~ varma$ mysql -u root -p
```

IMDb

{SQL → interactive
→ applied angle}

JupyterLab | mtod7/Numpy-Pandas | Applied Course | SQL IN DETAIL - Google | Test Table | LinkedIn | appliedaicourse.com/lecture/11/applied-machine-learning-online-course/3481/use-describe-show-tables/1/module-1-fundamentals-of-programming

← → C appliedaicourse.com/lecture/11/applied-machine-learning-online-course/3481/use-describe-show-tables/1/module-1-fundamentals-of-programming... | +

Mac

Windows

Linux

Terminal

Command

How to utilise Appliedaicourse

- Python for Data Science Introduction
- Python for Data Science: Data Structures
- Python for Data Science: Functions
- Python for Data Science: Numpy
- Python for Data Science: Matplotlib
- Python for Data Science: Pandas

Prev Next

imdb.sql

Type here to search

14:14 27-09-2019

```
Macbook-air:~ varma$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 19
Server version: 8.0.13 MySQL Community Server - GPL

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> 
```



```
Macbook-air:~ varma$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 19
Server version: 8.0.13 MySQL Community Server - GPL

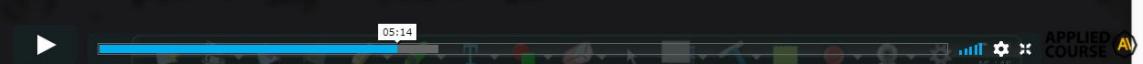
Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> USE imdb
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> 
```



```
mysql> use imdb
Database changed
mysql> 
```

USE DB-NAME

SELECT * FROM movies;



```
mysql> use imdb
Database changed
mysql> show tables
+-----+
| Tables_in_imdb |
+-----+
| actors          |
| directors       |
| directors_genres |
| movies          |
| movies_directors |
| movies_genres   |
| roles           |
+-----+
7 rows in set (0.00 sec)

mysql> 
```

SELECT * FROM movies;



```
mysql> SHOW TABLES;
+-----+
| Tables_in_imdb |
+-----+
| actors          |
| directors       |
| directors_genres|
| movies          |
| movies_directors|
| movies_genres   |
| roles           |
+-----+
7 rows in set (0.00 sec)

mysql> 
```

USE <DB-NAME>
SHOW TABLES;

```
mysql> SHOW TABLES;
+-----+
| Tables_in_imdb |
+-----+
| actors          |
| directors       |
| directors_genres|
| movies          |
| movies_directors|
| movies_genres   |
| roles           |
+-----+
7 rows in set (0.00 sec)

mysql> 
```

```
-- mysql -u root -p
mysql> SHOW TABLES;
+-----+
| Tables_in_imdb |
+-----+
| actors          |
| directors       |
| directors_genres|
| movies          |
| movies_directors|
| movies_genres   |
| roles           |
+-----+
7 rows in set (0.00 sec)

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 (0.00 sec)

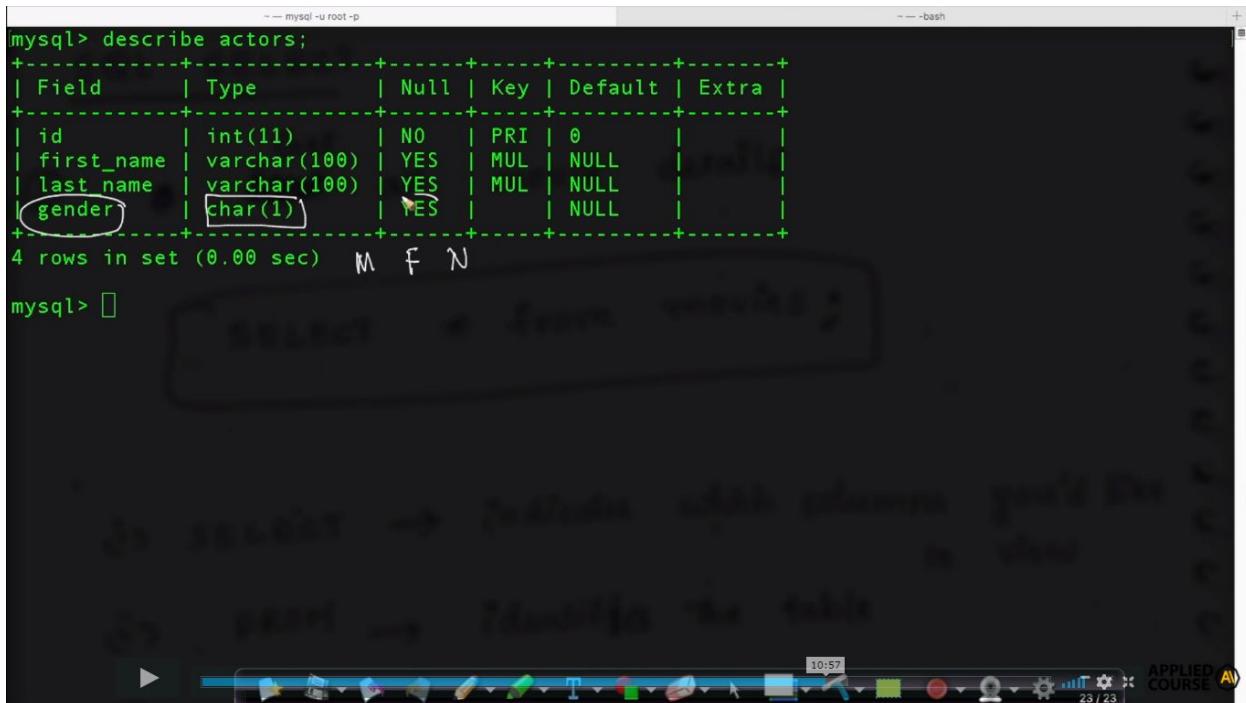
mysql> 
```

```
-- mysql -u root -p
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 (0.00 sec)

mysql> 
```

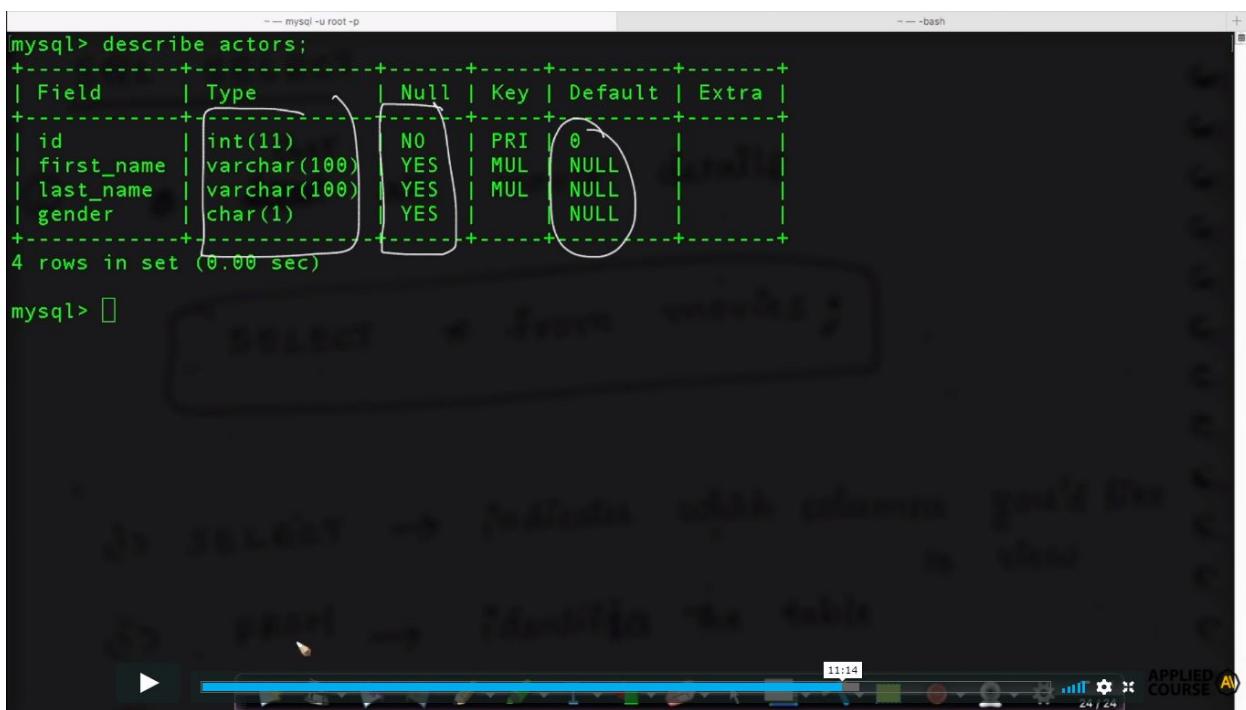
```
-- mysql -u root -p
mysql> describe actors;
+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| id   | int(11) | NO  | PRI | 0       |          |
| first_name | varchar(100) | YES | MUL | NULL    |          |
| last_name  | varchar(100) | YES | MUL | NULL    |          |
| gender     | char(1)  | YES |      | NULL    |          |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)  M F N
```

mysql> [REDACTED]



```
-- mysql -u root -p
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 (0.00 sec)
```

mysql> [REDACTED]



```
-- mysql -u root -p
mysql> show tables
[ -> ;
+-----+
| Tables_in_imdb |
+-----+
| actors
| directors
| directors_genres
| movies
| movies_directors
| movies_genres
| roles
+-----+
7 rows in set (0.01 sec)

mysql> DESCRIBE movies_directors;
+-----+-----+-----+-----+-----+
| Field      | Type     | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| director_id | int(11) | NO   | PRI | NULL    |       |
| movie_id    | int(11) | NO   | PRI | NULL    |       |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> █
```

```
-- mysql -u root -p
mysql> show tables
[ -> ;
+-----+
| Tables_in_imdb |
+-----+
| actors
| directors
| directors_genres
| movies
| movies_directors
| movies_genres
| roles
+-----+
7 rows in set (0.01 sec)

mysql> DESCRIBE movies_directors;
+-----+-----+-----+-----+-----+
| Field      | Type     | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| director_id | int(11) | NO   | PRI | NULL    |       |
| movie_id    | int(11) | NO   | PRI | NULL    |       |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> DESCIBE directors_genres
[ -> ;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
your MySQL server version for the right syntax to use near 'DESCIBE`' at line 1
```

```
-- mysql -u root -p
[ -> ;
+-----+
| Tables_in_imdb |
+-----+
| actors          |
| directors        |
| directors_genres|
| movies          |
| movies_directors|
| movies_genres   |
| roles           |
+-----+
7 rows in set (0.01 sec)

mysql> DESCRIBE movies_directors;
+-----+-----+-----+-----+-----+
| Field      | Type     | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| director_id | int(11) | NO   | PRI | NULL    |       |
| movie_id    | int(11) | NO   | PRI | NULL    |       |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> DESCRIBE directors_genres
[ -> 
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
your MySQL server
1
```

A handwritten arrow points from the word "DESCRIBE" in the last command to the word "parse" written above it.

```
-- mysql -u root -p
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)

mysql> 
```

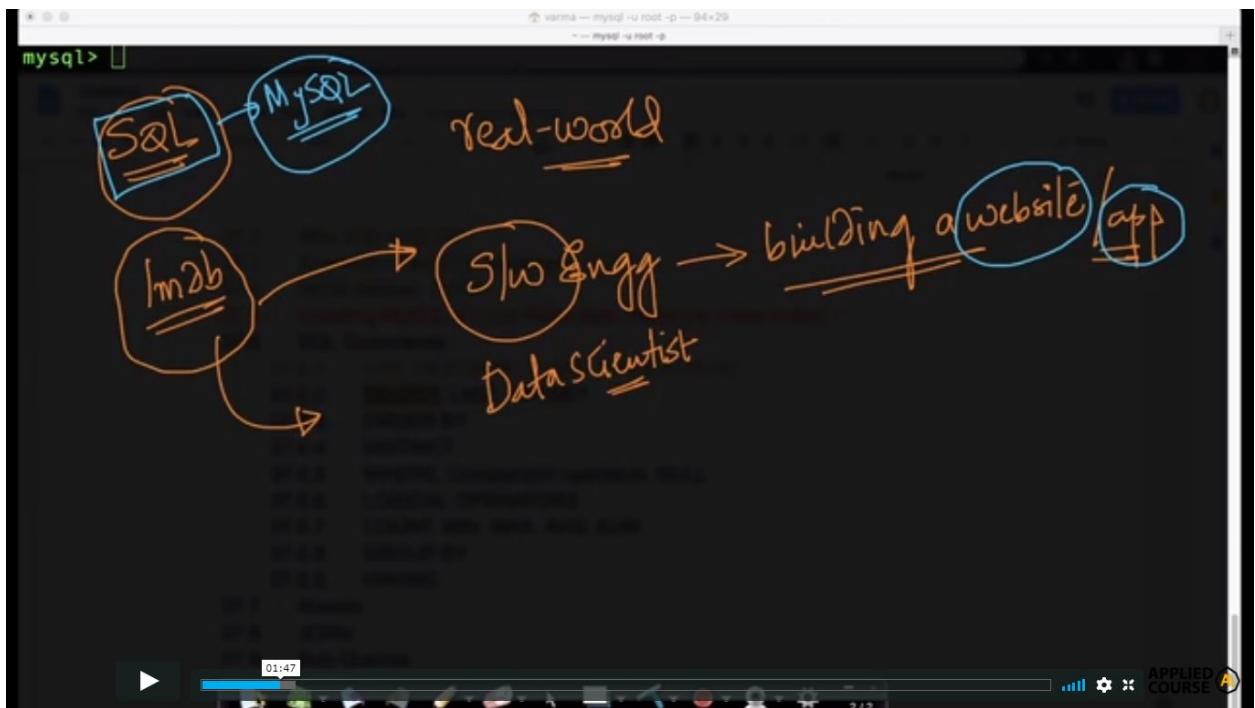
```
-- mysql -u root -p
mysql> DESCRIBE directors_genres;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| director_id | int(11) | NO   | PRI | NULL    |       |
| genre        | varchar(100)| NO  | PRI | NULL    |       |
| prob    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

mysql> 
```

Handwritten notes:

- ✓ USE <DB-name> ✓
- ✓ SHOW TABLES ✓
- ✓ DESCRIBE <table-name> ✓

SELECT:



```
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)
```

mysql> [REDACTED]

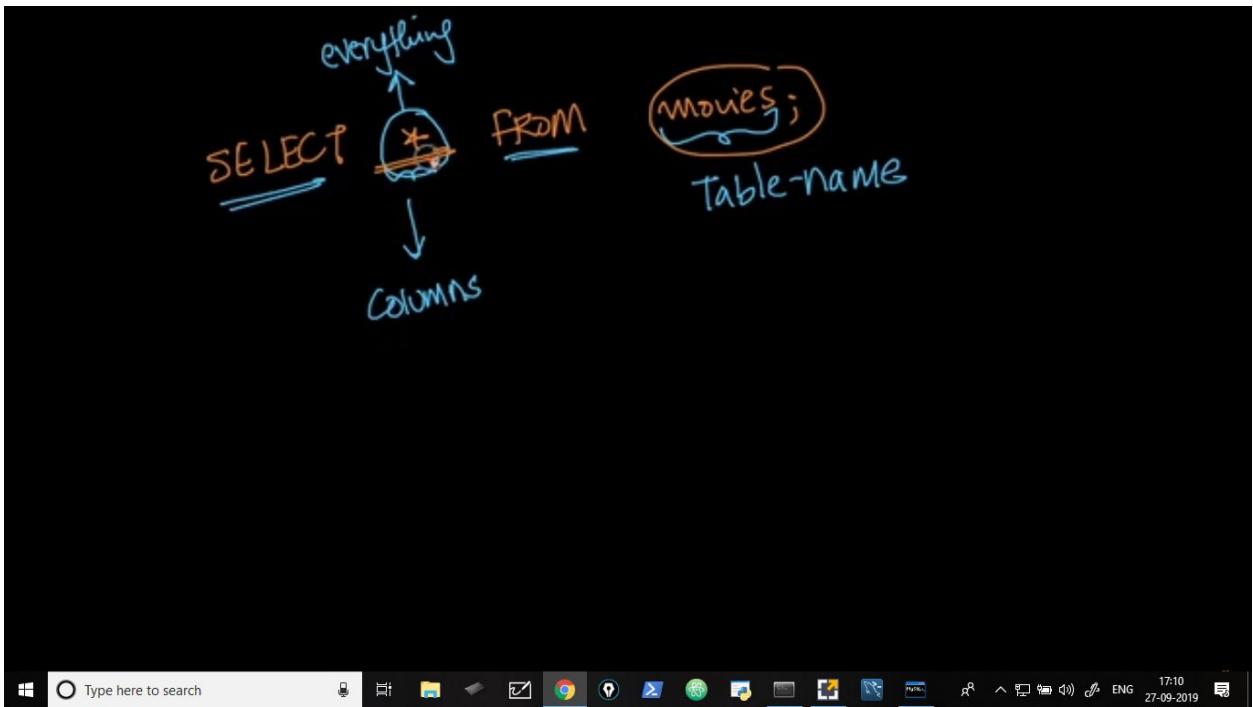
APPLIED COURSE A

```
* * *
varma — mysql -u root -p — 94x29
-- mysql -u root -p

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 (0.00 sec)

mysql> 
```

Tables → ↗

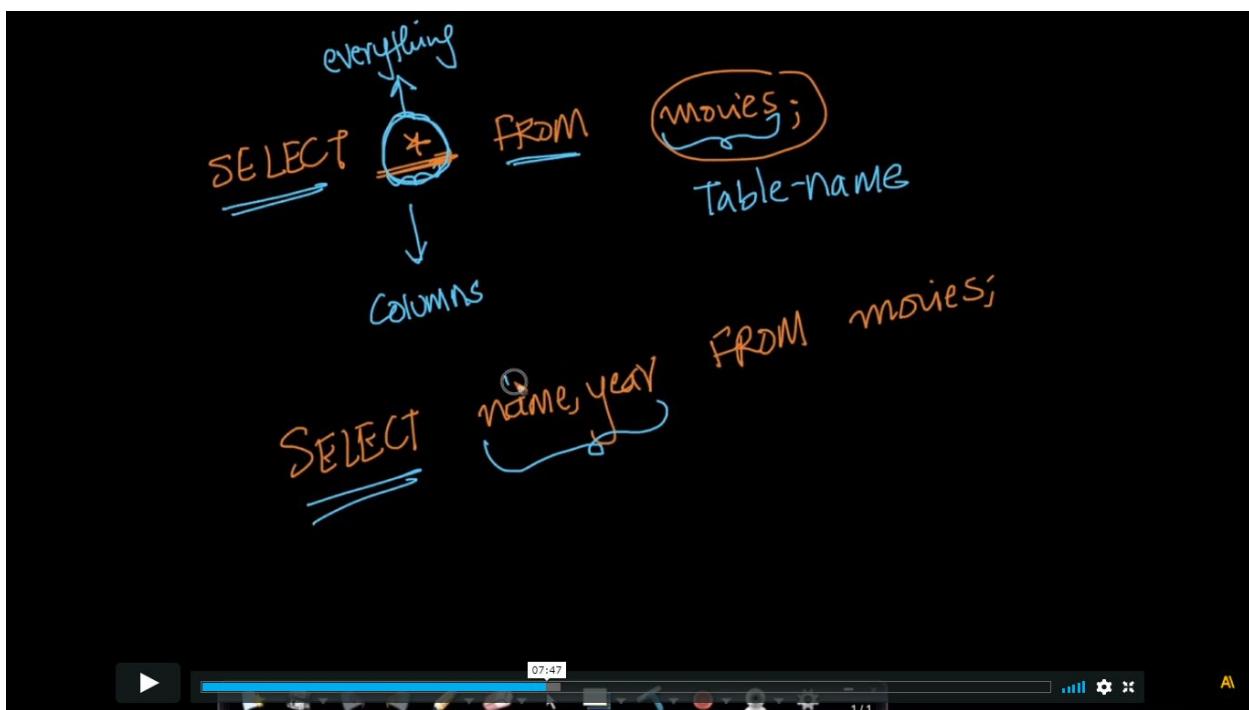


```
mysql> describe movies;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| id    | int(11) | NO   | PRI | 0       |          |
| name  | varchar(100) | YES  | MUL | NULL    |          |
| year  | int(11) | YES  |     | NULL    |          |
| rankscore | float | YES  |     | NULL    |          |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> 
```

Tables → rows of columns

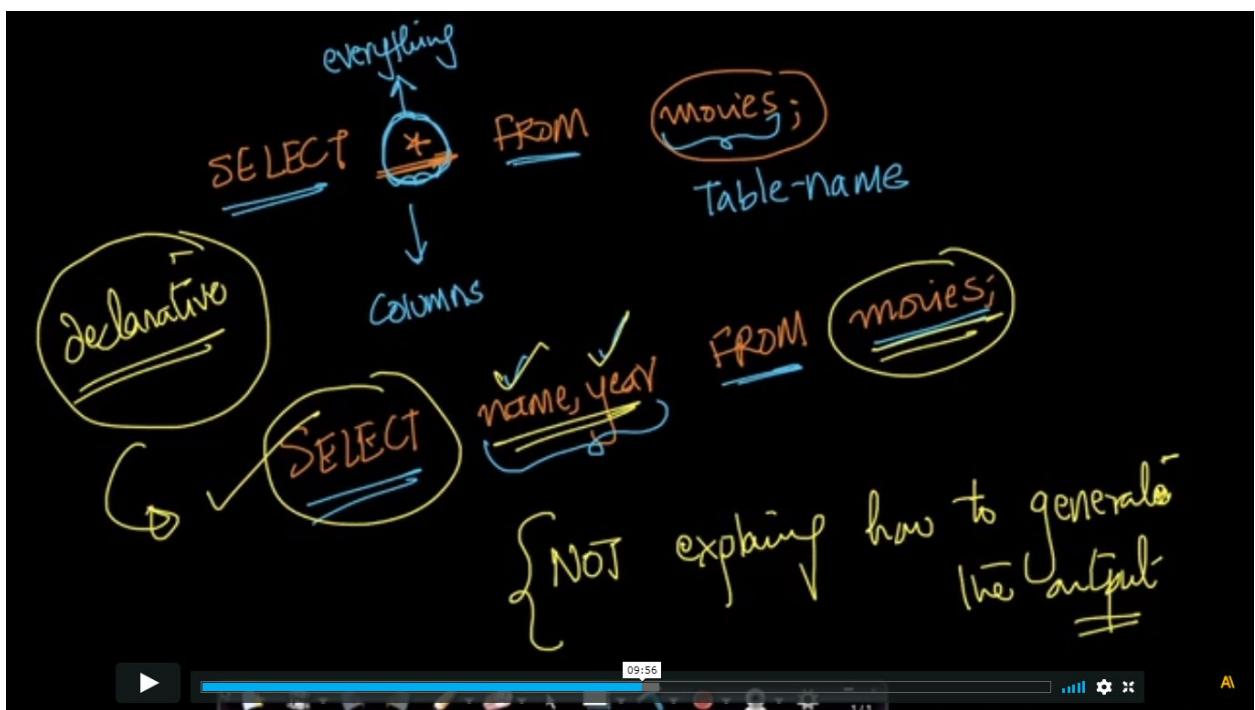
Task: list ~~name, year~~ of all movies

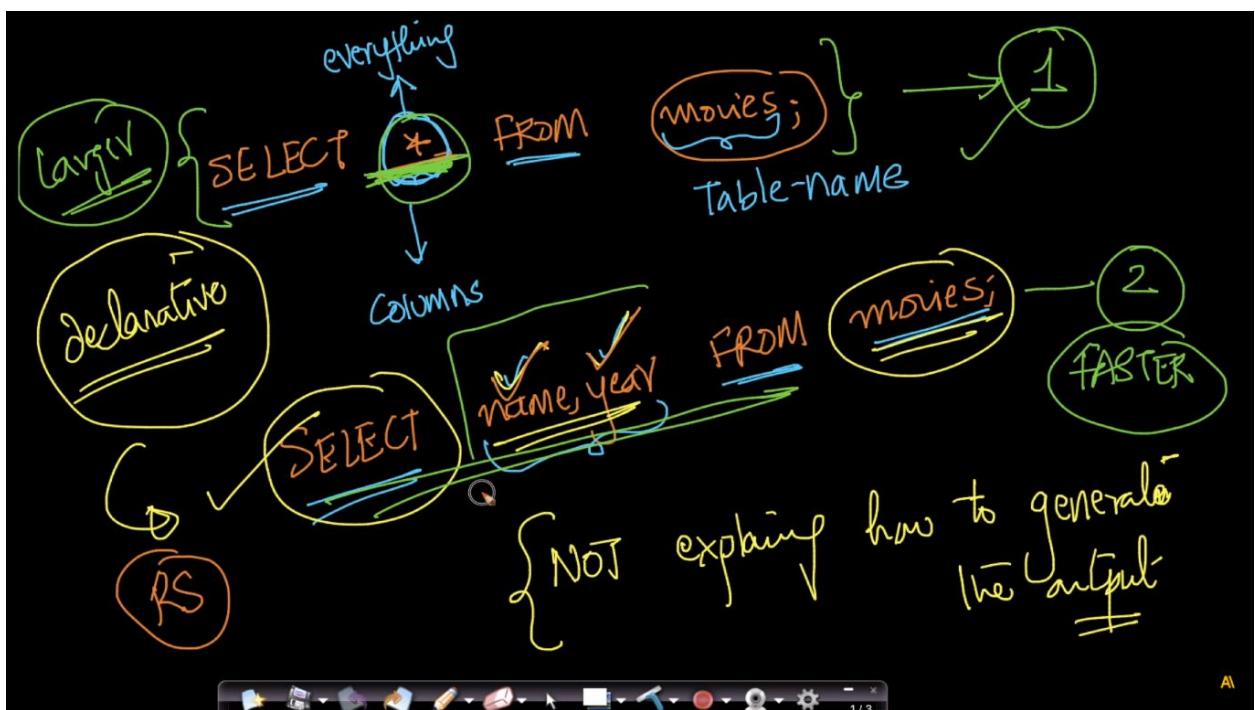
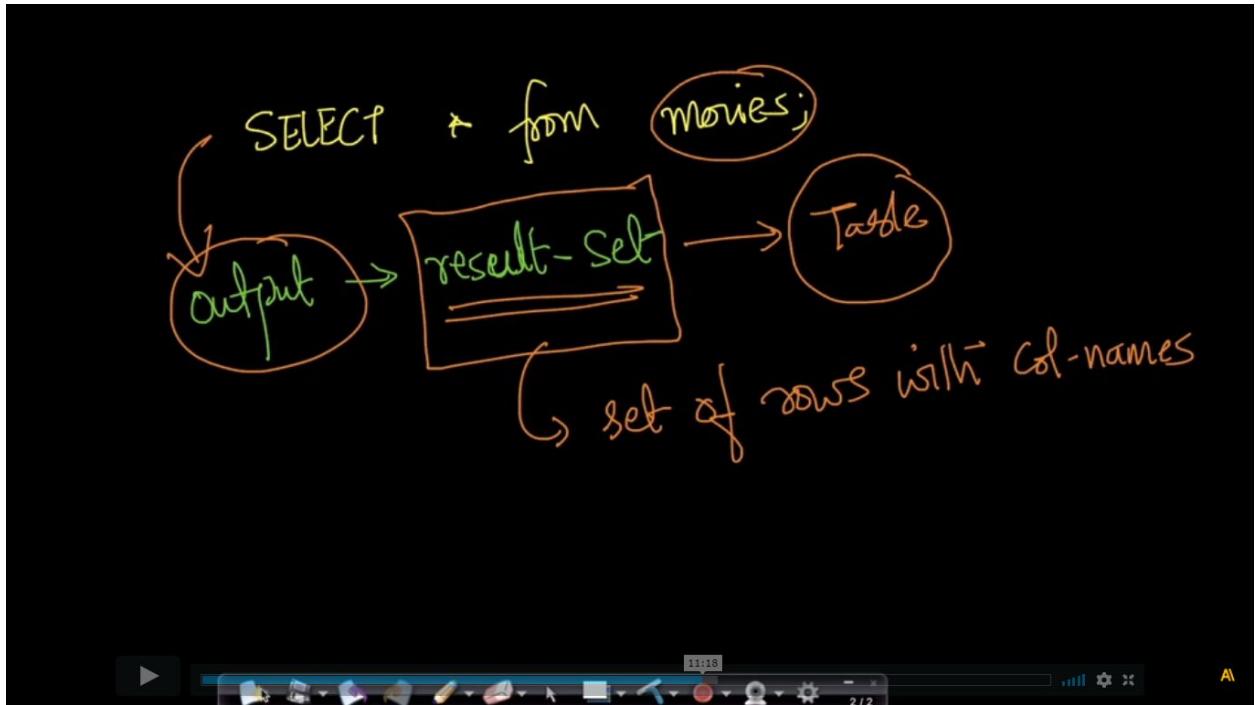


varma — mysql -u root -p — 94x29
— mysql -u root -p

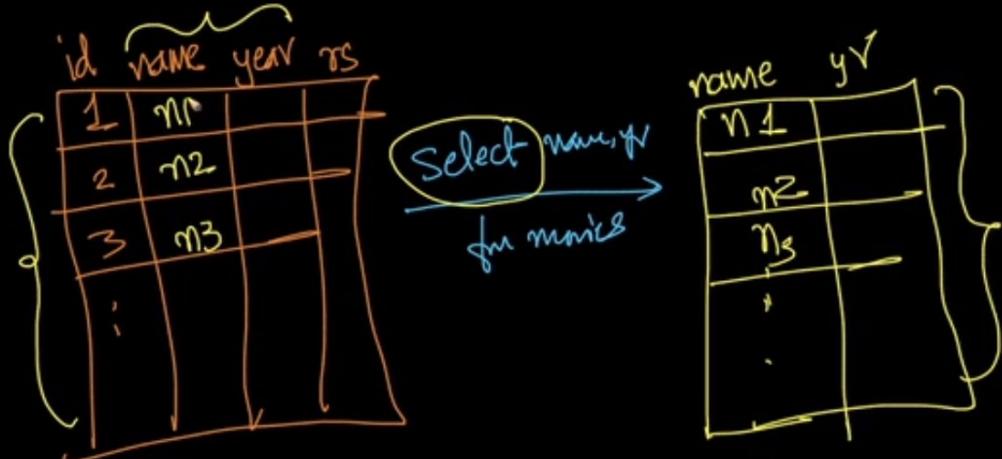
```
| "ltimo Inverno, O"
|   | 1953 |
| "ltimo Inverno, O"
|   | 1957 |
| "ltimo verano, El"
|   | 1996 |
| "nica noche, La"
|   | 1985 |
| "nica Verdade, A"
|   | 1958 |
| "nica Verdade, A"
|   | 1962 |
| "pa el nimo"
|   | 2002 |
| "zem blch krlu"
|   | 1991 |
| "rgammk"
|   | 1995 |
| "zgnm Leyla"
|   | 2002 |
| "Istanbul"
|   | 1983 |
| "sterreich"
|   | 1958 |
+
+-----+
388269 rows in set (0.19 sec)
```

mysql>





```
SQL commands UNREGISTERED
1 USE imdb;
2 SHOW TABLES;
3 DESCRIBE movies;
4 ****
5
6 ****
7
8 SELECT * FROM movies;
9 # more data transfer
10
11 #result-set: a set of rows that form the result of a query along with column-names and meta-data.
12
13 SELECT name,year FROM movies;
14
15 SELECT rankscore,name FROM movies;
16 #row order same as the one in the table
17
18 ****
19
20
21 LIMIT:
22
23 SELECT name,rankscore FROM movies LIMIT 20;
24
25 SELECT name,rankscore FROM movies LIMIT 20 OFFSET 40;
26
27 ****
28
29
30 ORDER BY:
31
32 # list ►cent... 15:37 APPLIED COURSE A
33
```



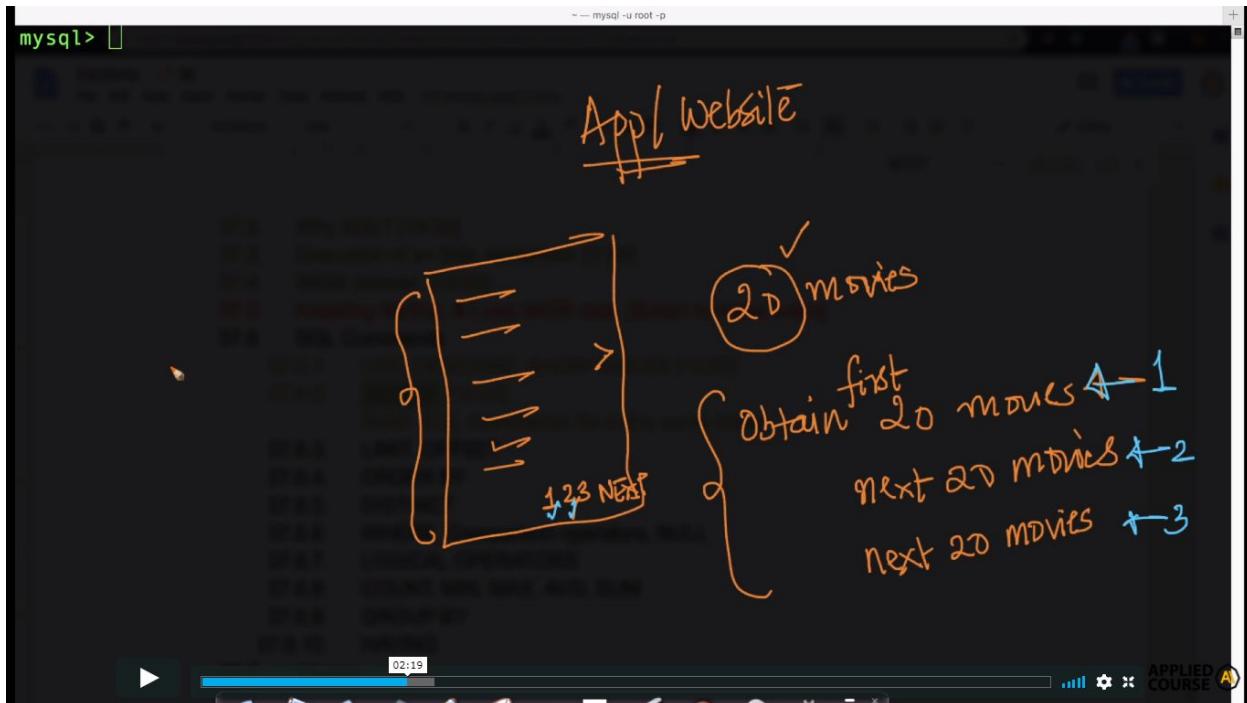
row-order preservation

A screenshot of a MySQL terminal window. The command entered is:

```
mysql> SELECT * FROM movies;
```

The terminal shows the output of the query, which includes column names and data. Handwritten annotations are present: a box around the asterisk (*) in the SELECT statement is labeled "col-names", and a circle around the "RS" in the command line is labeled "RS".

LIMIT, OFFSET



SQL commands

```

15 SELECT rankscore, name FROM movies;
16 # row order same as the one in the table
17
18 ****
19
20 LIMIT: Col-names Table # rows
21 SELECT name, rankscore FROM movies LIMIT 20; → first 20 rows
22
23 SELECT name, rankscore FROM movies LIMIT 20 OFFSET 40;
24
25 ****
26
27 ORDER BY:
28
29 # list recent movies first
30
31
32 SELECT name, rankscore, year FROM movies ORDER BY year DESC LIMIT 20;
33
34 # default: ASC
35 # the output row order amynot be same as the one in the table due to query optimzier and internal
36 # data-structres/indices.
37
38 SELECT name, rankscore FROM movies ORDER BY year DESC LIMIT 20;
39
40 ****
41
42
43 DISTINCT
44
45

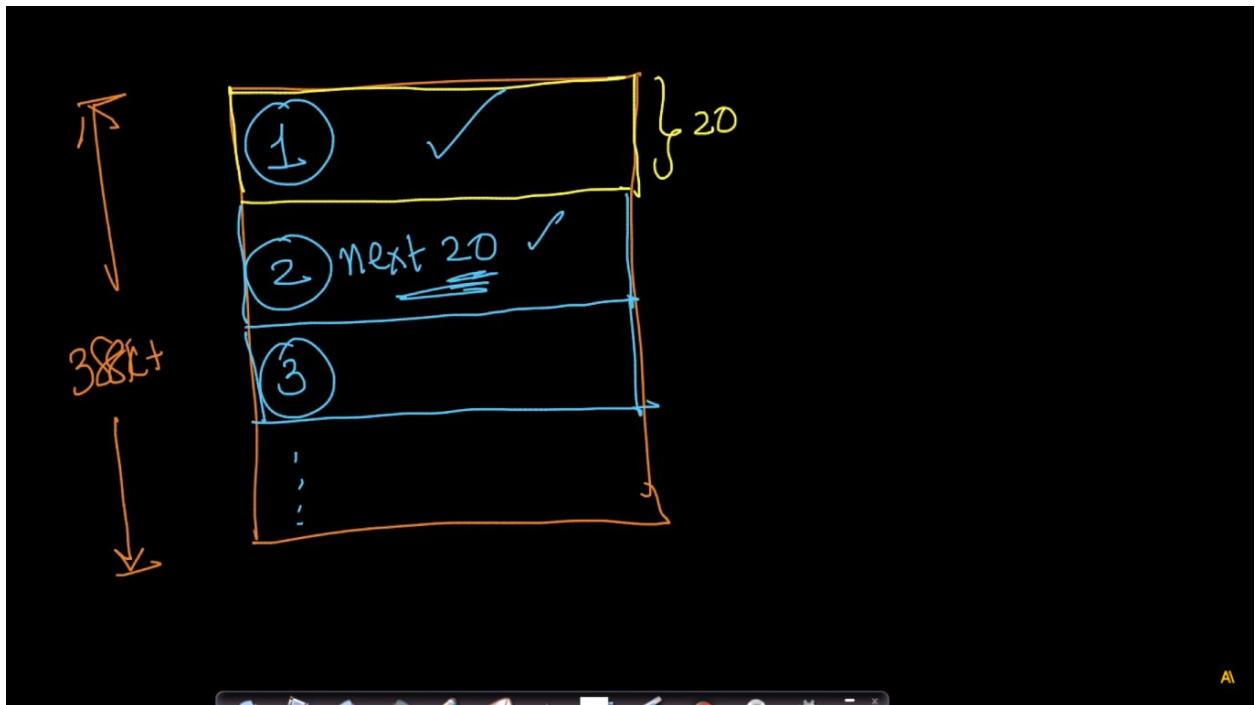
```

first 20 rows

next 20 movies

next 20 movies

388K



```
mysql> SELECT name,rankscore FROM movies LIMIT 20 OFFSET 20;
```

```
SQL commands
15 SELECT rankscore,name FROM movies;
16 #row order same as the one in the table
17 ****
18
19
20
21 LIMIT:
22
23 SELECT name,rankscore FROM movies LIMIT 20;
24
25 SELECT name,rankscore FROM movies LIMIT 20 OFFSET 40;
26 ****
27
28
29 ORDER BY:
30
31 # list recent movies first
32
33 SELECT name,rankscore,year FROM movies ORDER BY year DESC LIMIT 20;
34
35 # default:ASC
36 # the output row order amynot be same as the one in the table due to query optimzier and internal
37 data-structres/indices.
38
39 SELECT name,rankscore FROM movies ORDER BY year DESC LIMIT 20;
40 ****
41
42
43 DISTINCT
```

LIMIT:

```
SELECT name,rankscore FROM movies LIMIT 20;
```

```
SELECT name,rankscore FROM movies LIMIT 20 OFFSET 40;
```

→ This command implies to ignore first 40 rows and print next 20 rows ignornoing firsr 40 rows.

ORDER BY

```
mysql> SELECT name,rankscore FROM movies LIMIT 20 OFFSET 20;
```

```
15 SELECT rankscore,name FROM movies;
16 #row order same as the one in the table
17
18 ****
19
20
21 LIMIT: 21
22 SELECT name,rankscore FROM movies LIMIT 20;
23
24 SELECT name,rankscore FROM movies LIMIT 20 OFFSET 40;
25
26 ****
27
28
29 ORDER BY:
30
31 # list recent movies first
32
33 SELECT name,rankscore,year FROM movies ORDER BY year DESC LIMIT 20;
34
35 # default:ASC
36 # the output row order amynot be same as the one in the table due to query optimzier and internal
37 data-structres/indices.
38
39 SELECT name,rankscore FROM movies ORDER BY year DESC LIMIT 20;
40
41 ****
42
43
44 DISTINCT
```

varma — mysql -u root -p — 94x30

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

Col Table year DESC Limit 10; descending

varma — mysql -u root -p — 94x30

```
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)
```

```
mysql> 
```

```
varma — mysql -u root -p — 94x30
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)
```

```
mysql> 
```

↓ fault: ASC

APPLIED COURSE A

```

mysql> SELECT name,rankscore,year FROM movies ORDER BY year LIMIT 10;
+-----+-----+-----+
| name | rankscore | year |
+-----+-----+-----+
| Roundhay Garden Scene | NULL | 1888 |
| Traffic Crossing Leeds Bridge | NULL | 1888 |
| Monkeyshines, No. 2 | NULL | 1890 |
| Monkeyshines, No. 1 | 7.3 | 1890 |
| Monkeyshines, No. 3 | NULL | 1890 |
| Duncan Smoking | 3.6 | 1891 |
| Newark Athlete | 4.3 | 1891 |
| Duncan or Devonald with Muslin Cloud | 3.5 | 1891 |
| Monkey and Another, Boxing | 3.2 | 1891 |
| Duncan and Another, Blacksmith Shop | 3.5 | 1891 |
+-----+-----+-----+
10 rows in set (0.15 sec)

mysql> 

```

Annotations:

- Handwritten note: "first 10 rows" points to the output of the query.
- Handwritten note: "ASC" and "DESC" are circled, with arrows pointing to the "year" column header and the "ASC" and "DESC" options in the query.
- Handwritten note: "Default" points to the "ASC" option.

```

SQL commands
25 SELECT name,rankscore FROM movies LIMIT 20 OFFSET 40;
26 ****
27
28
29
30 ORDER BY: ✓
31 # list recent movies first
32
33 SELECT name,rankscore,year FROM movies ORDER BY year DESC LIMIT 10;
34
35 # default:ASC
36
37 SELECT name,rankscore,year FROM movies ORDER BY year LIMIT 10;
38
39 # the output row order maynot be same as the one in the table due to query optimzier and internal
40 # data-structres/indices.
41 ****
42
43
44 DISTINCT:
45
46
47 # list all genres of
48 SELECT DISTINCT genre FROM movies_genres;
49
50
51 # multiple-column DISTINCT
52 SELECT DISTINCT first_name, last_name FROM directors;
53
54 ****
55 WHERE:
56

```

Annotations:

- Handwritten note: "Sorting" is circled and has an arrow pointing to the "ORDER BY" section.
- Handwritten note: "ASC" is circled and has an arrow pointing to the "ASC" option in the code.
- Handwritten note: "DESC" is circled and has an arrow pointing to the "DESC" option in the code.
- Handwritten note: "Default:ASC" is circled and has an arrow pointing to the "SELECT name,rankscore,year FROM movies ORDER BY year LIMIT 10;" line.
- Handwritten note: "DISTINCT:" is circled and has an arrow pointing to the "DISTINCT" keyword.
- Handwritten note: "multiple-column DISTINCT" is circled and has an arrow pointing to the "SELECT DISTINCT first_name, last_name FROM directors;" line.

ORDER BY:

list recent movies first

CC

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

```
SQL commands UNREGISTERED
25 SELECT name,rankscore FROM movies LIMIT 20 OFFSET 40;
26 ****
27 ****
28 ****
29 ****
30 ORDER BY: ✓
31 # list recent movies first
32
33 SELECT name,rankscore,year FROM movies ORDER BY year DESC LIMIT 10;
34 # default:ASC
35
36 SELECT name,rankscore,year FROM movies ORDER BY year LIMIT 10;
37
38 # the output row order maynot be same as the one in the table due to query optimzier and internal
39 # data-structres/indices.
40 ****
41 ****
42 ****
43 ****
44 ****
45 DISTINCT:
46
47 # list all genres of
48 SELECT DISTINCT genre FROM movies_genres;
49
50
51 # multiple-column DISTINCT
52 SELECT DISTINCT first_name, last_name FROM directors;
53
54 ****
55 WHERE:
56
```

05:59 APPLIED COURSE A

```
varma — mysql -u root -p — 94x30
mysql> DESCRIBE movies_genres;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| movie_id | int(11) | NO  | PRI | NULL   |       |
| genre    | varchar(100) | NO  | PRI | NULL   |       |
+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)

mysql> □
```

HORROR
COMEDY

TASK:

DISTINCT

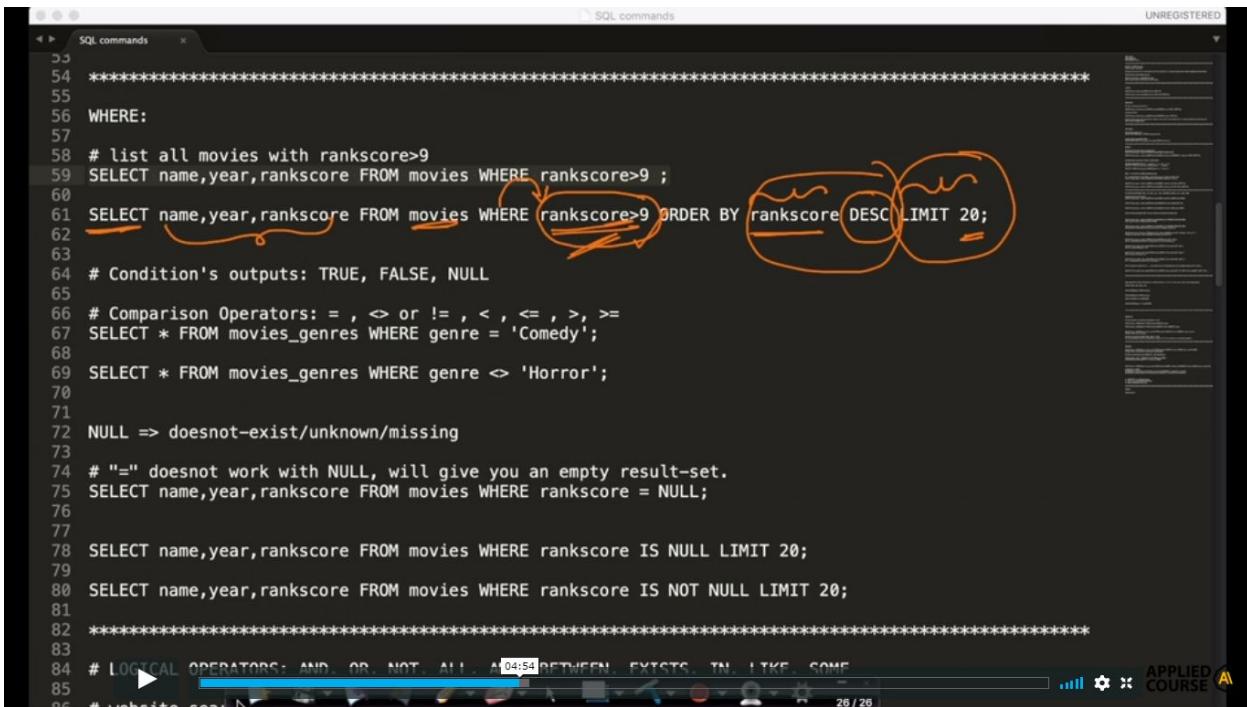
00:48 8 / 8 APPLIED COURSE A

```
varma — mysql -u root -p — 94x30
mysql> SELECT DISTINCT genre FROM movies_genres;
+-----+
| genre |
+-----+
| Documentary |
| Short |
| Comedy |
| Crime |
| Western |
| Family |
| Animation |
| Drama |
| Romance |
| Mystery |
| Thriller |
| Adult |
| Music |
| Action |
| Fantasy |
| Sci-Fi |
| Horror |
| War |
| Musical |
| Adventure |
| Film-Noir |
+-----+
21 rows in set (0.16 sec)

mysql> [REDACTED]
```

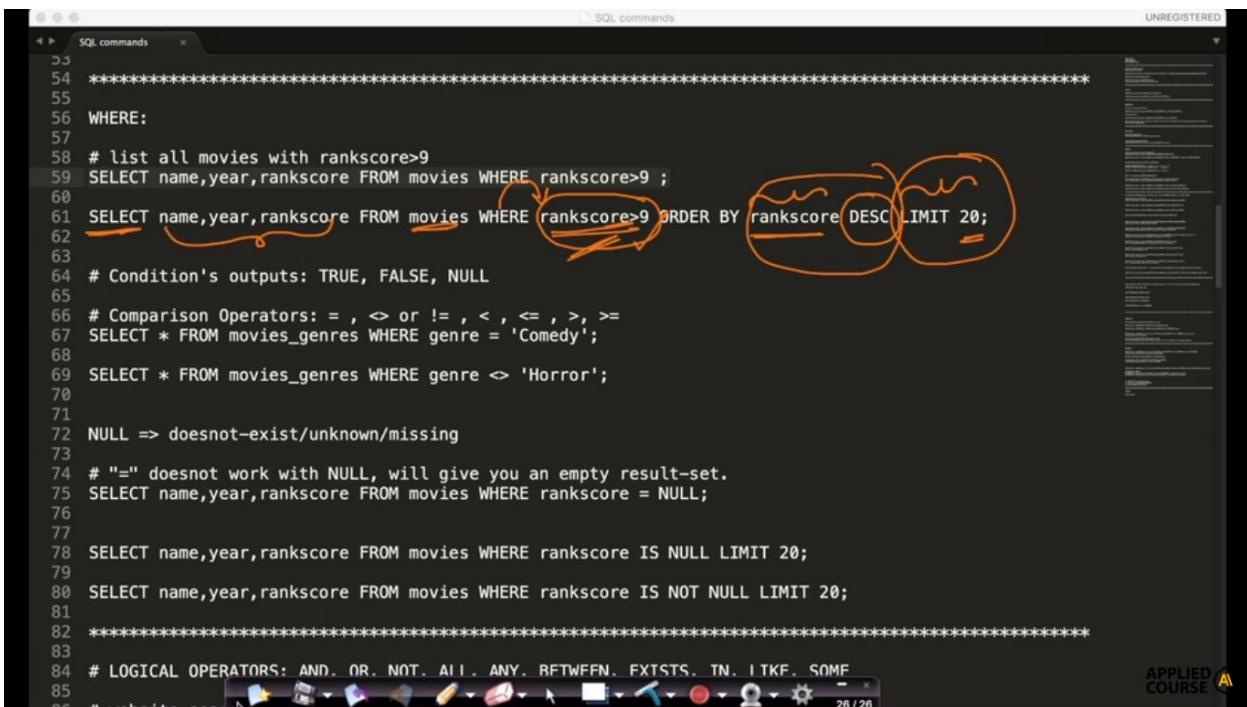
```
varma — mysql -u root -p — 94x30
mysql> SELECT DISTINCT first_name, last_name FROM directors ORDER BY first_name;[REDACTED]
```

WHERE, Comparison operators, NULL



A screenshot of a video player window titled "SQL commands". The video is at 04:54 of 26/26. The content shows a series of SQL queries. Handwritten annotations include circles around the WHERE clause and comparison operators (=, <, >), and a large circle around the entire WHERE clause of the second query. There are also some scribbles on the right side of the screen.

```
33
34 ****
35
36 WHERE:
37
38 # list all movies with rankscore>9
39 SELECT name,year,rankscore FROM movies WHERE rankscore>9 ;
40
41 SELECT name,year,rankscore FROM movies WHERE rankscore>9 ORDER BY rankscore DESC LIMIT 20;
42
43 # Condition's outputs: TRUE, FALSE, NULL
44
45 # Comparison Operators: = , <> or != , < , <= , > , >=
46 SELECT * FROM movies_genres WHERE genre = 'Comedy';
47
48 SELECT * FROM movies_genres WHERE genre <> 'Horror';
49
50
51 NULL => doesnot-exist/unknown/missing
52
53 # "=" doesnot work with NULL, will give you an empty result-set.
54 SELECT name,year,rankscore FROM movies WHERE rankscore = NULL;
55
56
57 SELECT name,year,rankscore FROM movies WHERE rankscore IS NULL LIMIT 20;
58
59 SELECT name,year,rankscore FROM movies WHERE rankscore IS NOT NULL LIMIT 20;
60
61 *****
62 # LOGICAL OPERATORS: AND. OR. NOT. ALL. ANY. BETWEEN. EXISTS. IN. LIKE. SOME
63
64 # website: www.w3schools.com/sql/
```



A screenshot of a video player window titled "SQL commands". The video is at 04:54 of 26/26. The content shows the same series of SQL queries as the previous screenshot. Handwritten annotations are identical to the first screenshot, with circles around the WHERE clause and comparison operators, and a large circle around the entire WHERE clause of the second query.

```
33
34 ****
35
36 WHERE:
37
38 # list all movies with rankscore>9
39 SELECT name,year,rankscore FROM movies WHERE rankscore>9 ;
40
41 SELECT name,year,rankscore FROM movies WHERE rankscore>9 ORDER BY rankscore DESC LIMIT 20;
42
43 # Condition's outputs: TRUE, FALSE, NULL
44
45 # Comparison Operators: = , <> or != , < , <= , > , >=
46 SELECT * FROM movies_genres WHERE genre = 'Comedy';
47
48 SELECT * FROM movies_genres WHERE genre <> 'Horror';
49
50
51 NULL => doesnot-exist/unknown/missing
52
53 # "=" doesnot work with NULL, will give you an empty result-set.
54 SELECT name,year,rankscore FROM movies WHERE rankscore = NULL;
55
56
57 SELECT name,year,rankscore FROM movies WHERE rankscore IS NULL LIMIT 20;
58
59 SELECT name,year,rankscore FROM movies WHERE rankscore IS NOT NULL LIMIT 20;
60
61 *****
62 # LOGICAL OPERATORS: AND. OR. NOT. ALL. ANY. BETWEEN. EXISTS. IN. LIKE. SOME
63
64 # website: www.w3schools.com/sql/
```

```
mysql> SELECT name,year,rankscore FROM movies WHERE rankscore = NULL;  
Empty set (0.12 sec)  
mysql>   
  
= doesn't work with NULL
```

```
10:38  
SQL commands  
*****  
53  
54 *****  
55  
56 WHERE:  
57  
58 # list all movies with rankscore>9  
59 SELECT name,year,rankscore FROM movies WHERE rankscore>9 ;  
60  
61 SELECT name,year,rankscore FROM movies WHERE rankscore>9 ORDER BY rankscore DESC LIMIT 20;  
62  
63  
64 # Condition's outputs: TRUE, FALSE, NULL  
65  
66 # Comparison Operators: = , <> or != , < , <= , > , >=  
67 SELECT * FROM movies_genres WHERE genre = 'Comedy';  
68  
69 SELECT * FROM movies_genres WHERE genre <> 'Horror';  
70  
71 NULL => doesnot-exist/unknown/missing  
72  
73 # "=" doesnot work with NULL, will give you an empty result-set.  
74 SELECT name,year,rankscore FROM movies WHERE rankscore = NULL;  
75  
76 *****  
77 SELECT name,year,rankscore FROM movies WHERE rankscore IS NULL LIMIT 20;  
78  
79 SELECT name,year,rankscore FROM movies WHERE rankscore IS NOT NULL LIMIT 20;  
80  
81 *****  
82  
83 # LOGICAL OPERATORS: AND OR NOT ALL ANY BETWEEN EXISTS IN LIKE SOME  
84 # update course  
85  
36 / 36  
APPLIED COURSE A
```

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;
```

Logical Operators

```
SQL commands UNREGISTERED
103
104 SELECT name,year,rankscore FROM movies WHERE year BETWEEN 2000 AND 1999;
105 #lowvalue <= highvalue else you will get an empty result set
106
107
108 SELECT director_id, genre FROM directors_genres WHERE genre IN ('Comedy','Horror');
109 # same as genre='Comedy' OR genre='Horror'
110
111
112 SELECT name,year,rankscore FROM movies WHERE name LIKE 'Tis%';
113 # % => wildcard character to imply zero or more characters
114
115
116 SELECT first_name, last_name FROM actors WHERE first_name LIKE '%es';
117 # first name ending in 'es'
118
119
120 SELECT first_name, last_name FROM actors WHERE first_name LIKE '%es%';
121 #first name contains 'es'
122
123
124 SELECT first_name, last_name FROM actors WHERE first_name LIKE 'Agn_s';
125 # '_' implies only atmost one character.
126
127
128 # If we want to match % or _, we should use the backslash as the escape character: \% and \_
129
130
131 SELECT first_name, last_name FROM actors WHERE first_name LIKE 'L%' AND first_name NOT LIKE 'Li%';
132
133
134 ****
135 *****
```

Annotations:

- Line 113: A handwritten note next to the '%' symbol says "nm" with an arrow pointing to it.
- Line 117: A circled 'es' is followed by a handwritten note "=> zero or more chars" with an arrow pointing to the circled text.
- Line 121: A circled 'es' is followed by handwritten notes "cs" and "zes" with arrows pointing to the circled text.

```
SQL commands UNREGISTERED
105 #lowvalue <= highvalue else you will get an empty result set
106
107
108 SELECT director_id, genre FROM directors_genres WHERE genre IN ('Comedy','Horror');
109 # same as genre='Comedy' OR genre='Horror'
110
111
112 SELECT name,year,rankscore FROM movies WHERE name LIKE 'Tis%';
113 # %> wildcard character to imply zero or more characters
114
115
116 SELECT first_name, last_name FROM actors WHERE first_name LIKE '%es';
117 # first name ending in 'es'
118
119
120 SELECT first_name, last_name FROM actors WHERE first_name LIKE '%es%';
121 #first name contains 'es'
122
123
124 SELECT first_name, last_name FROM actors WHERE first_name LIKE 'Agn_s';
125 # '_' implies exactly one character.
126
127
128 # If we want to match % or _, we should use the backslash as the escape character: \% and \_
129
130
131 SELECT first_name, last_name FROM actors WHERE first_name LIKE 'L%' AND first_name NOT LIKE 'Li%';
132
133
134 ****
135 *****
```

Annotations:

- Line 113: A circled '%' is followed by handwritten notes "=> zero or more chars" with an arrow pointing to the circled symbol.
- Line 125: A circled '_' is followed by handwritten notes "=> one char" with an arrow pointing to the circled symbol.

T

name	percentage
m1	59%
m2	63%
m3	96%

\: escape char

SQL

%. → wild card char → zero or more char

%. → symbol

$$\frac{\text{percentage}}{\text{Symbol \%}}$$

SELECT * FROM T
WHERE percentage = %.
WILD CHAR

T

	name email
m1	n1 - abc@gmail.com
m2	n2 - xyz-ab@gmail.com

LIKE , $_abc@gmail.com$

Symbol

Not as a wild card char

SELECT * FROM T WHERE email like n1 - abc@gmail.com

wild card char

T

	name percentage
m1	59%
m2	63%
m3	96%

SQL

': escape char

SELECT * FROM T
WHERE percentage like '%'

WILD CHAR

% → wild card char → zero or more char

% → symbol

Symbol %

percentage = $\frac{\text{no. of rows}}{\text{total no. of rows}}$

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;
```

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

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

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

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%';
```

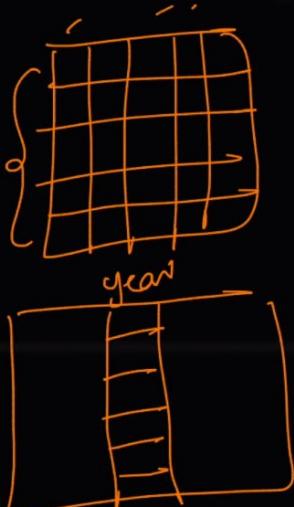
Aggregate Functions: COUNT, MIN, MAX, AVG, SUM

```
varma — mysql -u root -p — 94x30
mysql> SELECT MIN(year) FROM movies;
+-----+
| MIN(year) |
+-----+
| 1888 |
+-----+
1 row in set (0.09 sec)

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

mysql> SELECT COUNT(year) FROM movies;
+-----+
| COUNT(year) |
+-----+
| 388269 |
+-----+
1 row in set (0.07 sec)

mysql> □
```



```
varma — mysql -u root -p — 94x30
mysql> SELECT COUNT(*) FROM movies where year>2000;
+-----+
| COUNT(*) |
+-----+
| 46006 |
+-----+
1 row in set (0.08 sec)

mysql> □
```

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

Sub Queries/Nested Queries/Inner Queries

COUNT, MIN, MAX, SUM, AVG

```
SELECT MIN(year) FROM movies;
```

```
SELECT MAX(year) FROM movies;
```

```
SELECT COUNT(*) FROM movies;
```

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

```
SELECT COUNT(year) FROM movies;
```

GROUP BY

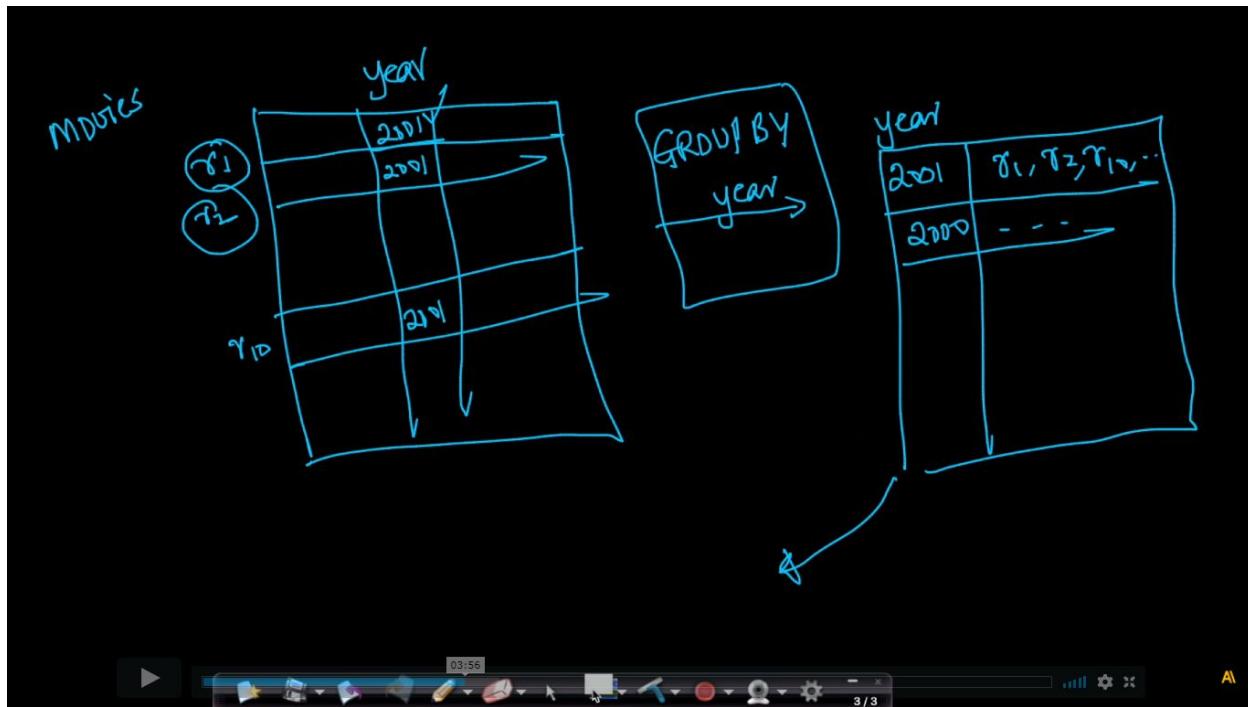
SQL commands UNREGISTERED

```

158
159
160 GROUP-BY → Data analyst, DS, SDE
161
162 # find number of movies released per year
163
164 SELECT year, COUNT(year) FROM movies GROUP BY year;
165
166 SELECT year, COUNT(year) FROM movies GROUP BY year ORDER BY year;
167
168
169 SELECT year, COUNT(year) year_count FROM movies GROUP BY year ORDER BY year_count;
170 # year_count is an alias.
171
172 # often used with COUNT, MIN, MAX or SUM.
173 # if grouping columns contain NULL values, all null values are grouped together.
174
175 ****
176
177 HAVING:
178
179 SELECT year, COUNT(year) year_count FROM movies GROUP BY year HAVING year_count>1000;
180 # specify a condition on groups using HAVING.
181
182 # often used along with GROUP BY. Not Mandatory.
183
184 SELECT name, year FROM movies HAVING year>2000;
185 # HAVING without GROUP BY is same as WHERE
186
187
188
189 SELECT year, COUNT(year) year_count FROM movies GROUP BY year HAVING year_count>20;
190

```

00:36 APPLIED COURSE A



```
varma — mysql -u root -p — 94x30
mysql>
mysql> SELECT year, COUNT(year) FROM movies GROUP BY year;
+-----+-----+
| year | COUNT(year) |
+-----+-----+
| 2002 | 12056 |
| 2000 | 11643 |
| 1971 | 4072 |
| 1913 | 3690 |
| 1915 | 3722 |
| 1923 | 1759 |
| 1920 | 2308 |
| 1921 | 2041 |
| 2001 | 11690 |
| 1939 | 1996 |
| 1941 | 1829 |
| 1912 | 3232 |
| 1999 | 10976 |
| 1996 | 8362 |
| 1918 | 2122 |
| 1914 | 3441 |
| 2004 | 8718 |
| 1980 | 4673 |
| 1989 | 5697 |
| 1975 | 4384 |
| 1924 | 1847 |
| 1911 | 2026 |
| 1986 |          |
| 1968 |          |

```

group by year as per some col

```
varma — mysql -u root -p — 94x30
+-----+-----+
| year | COUNT(year) |
+-----+-----+
| 1891 | 6 |
| 2008 | 1 |
| 1890 | 3 |
| 1888 | 2 |
+-----+-----+
120 rows in set (0.19 sec)

mysql>
mysql> SELECT year, COUNT(year) FROM movies GROUP BY year ORDER BY year;
+-----+-----+
| year | COUNT(year) |
+-----+-----+
| 1888 | 2 |
| 1890 | 3 |
| 1891 | 6 |
| 1892 | 9 |
| 1893 | 2 |
| 1894 | 59 |
| 1895 | 72 |
| 1896 | 410 |
| 1897 | 688 |
| 1898 | 1004 |
| 1899 | 845 |
| 1900 | 759 |
| 1901 | 837 |
| 1902 | 788 |
| 1903 | 831 |
| 1904 | 176 |
| 1905 |          |
| 1906 |          |

```

SQL commands UNREGISTERED

```
158
159
160 GROUP-BY
161
162 # find number of movies released per year
163 SELECT year, COUNT(year) FROM movies GROUP BY year;
164
165 SELECT year, COUNT(year) FROM movies GROUP BY year ORDER BY year;
166
167
168
169 SELECT year, COUNT(year) year_count FROM movies GROUP BY year ORDER BY year_count;
170 # year_count is an alias.
171
172 # often used with COUNT, MIN, MAX or SUM.
173 # if grouping columns contain NULL values, all null values are grouped together.
174
175 ****
176
177 HAVING:
178
179 SELECT year, COUNT(year) year_count FROM movies GROUP BY year HAVING year_count>1000;
180 # specify a condition on groups using HAVING.
181
182
183 # often used along with GROUP BY. Not Mandatory.
184
185 SELECT name, year FROM movies HAVING year>2000;
186 # HAVING without GROUP BY is same as WHERE
187
188
189
190 SELECT year, (
```

year_count

APPLIED COURSE A

```

| 2007 |      7 |
| 2008 |      1 |
+-----+-----+
120 rows in set (0.21 sec)

mysql>
mysql> SELECT year, COUNT(year) year_count FROM movies GROUP BY year ORDER BY year_count;
+-----+-----+
| year | year_count |
+-----+-----+
| 2008 |      1 |
| 1893 |      2 |
| 1888 |      2 |
| 1890 |      3 |
| 1891 |      6 |
| 2007 |      7 |
| 1892 |      9 |
| 1894 |     59 |
| 1895 |     72 |
| 1904 |    176 |
| 2006 |    195 |
| 1905 |    232 |
| 1907 |    332 |
| 1906 |    384 |
| 1896 |    410 |
| 1908 |   498 |
| 1897 |   688 |
| 1900 |   759 |
| 1902 |          |
| 1909 |          |
+-----+-----+

```

ASCI down arrow pointing to the 'year' column header in the first row of the result set.

```

158
159
160 GROUP-BY
161
162 # find number of movies released per year
163
164 SELECT year, COUNT(year) FROM movies GROUP BY year;
165
166 SELECT year, COUNT(year) FROM movies GROUP BY year ORDER BY year;
167
168
169 SELECT year, COUNT(year) year_count FROM movies GROUP BY year ORDER BY year_count;
170 # year_count is an alias.
171
172 # often used with COUNT, MIN, MAX or SUM.
173 # if grouping columns contain NULL values, all null values are grouped together.
174
175 ****
176
177 HAVING:
178
179 SELECT year, COUNT(year) year_count FROM movies GROUP BY year HAVING year_count > 1000;
180 # specify a condition on groups using HAVING.
181
182 # often used along with GROUP BY. Not Mandatory.
183
184
185 SELECT name, year FROM movies HAVING year > 2000;
186 # HAVING without GROUP BY is same as WHERE
187
188
189
190 SELECT year, (

```

Orange circle around the 'year' column header in the second row of the code editor. A curly orange arrow points from this circle to the 'year' column header in the first row of the result set above.

GROUP-BY

```
# find number of movies released per year  
  
SELECT year, COUNT(year) FROM movies GROUP BY year;  
  
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

SQL commands UNREGISTERED

```

158
159
160 GROUP-BY
161
162 # find number of movies released per year
163 SELECT year, COUNT(year) FROM movies GROUP BY year;
164
165 SELECT year, COUNT(year) FROM movies GROUP BY year ORDER BY year;
166
167
168
169 SELECT year, COUNT(year) year_count FROM movies GROUP BY year ORDER BY year_count;
170 # year_count is an alias.
171
172 # often used with COUNT, MIN, MAX or SUM.
173 # if grouping columns contain NULL values, all null values are grouped together.
174
175 ****
176
177 HAVING:
178
179 SELECT year, COUNT(year) year_count FROM movies GROUP BY year HAVING year_count>1000;
180 # specify a condition on groups using HAVING.
181
182 # often used along with GROUP BY. Not Mandatory.
183
184
185 SELECT name, year FROM movies HAVING year>2000;
186 # HAVING without GROUP BY is same as WHERE
187
188
189
190 SELECT year, (

```

APPLIED COURSE A

varma — mysql -u root -p — 94x30

```

| 2007 |      7 |
| 2008 |      1 |
+-----+
120 rows in set (0.21 sec)

mysql>
mysql> SELECT year, COUNT(year) year_count FROM movies GROUP BY year ORDER BY year_count;
+-----+-----+
| year | year_count |
+-----+-----+
| 2008 |          1 |
| 1893 |          2 |
| 1888 |          2 |
| 1890 |          3 |
| 1891 |          6 |
| 2007 |          7 |
| 1892 |          9 |
| 1894 |         59 |
| 1895 |         72 |
| 1904 |        176 |
| 2006 |        195 |
| 1905 |        232 |
| 1907 |        332 |
| 1906 |        384 |
| 1896 |        410 |
| 1908 |        498 |
| 1897 |        688 |
| 1900 |        759 |
| 1902 |    11:05   ▶
| 1909 |          1 |

```

ASC

SQL commands UNREGISTERED

```

158
159
160 GROUP-BY
161
162 # find number of movies released per year
163
164 SELECT year, COUNT(year) FROM movies GROUP BY year;
165
166 SELECT year, COUNT(year) FROM movies GROUP BY year ORDER BY year;
167
168
169 SELECT year, COUNT(year) year_count FROM movies GROUP BY year ORDER BY year_count;
170 # year_count is an alias.
171
172 # often used with COUNT, MIN, MAX or SUM.
173 # if grouping columns contain NULL values, all null values are grouped together.
174
175 ****
176
177 HAVING:
178
179 SELECT year, COUNT(year) year_count FROM movies GROUP BY year HAVING year_count>1000;
180 # specify a condition on groups using HAVING.
181
182 # often used along with GROUP BY. Not Mandatory.
183
184 SELECT name, year FROM movies HAVING year>2000;
185 # HAVING without GROUP BY is same as WHERE
186
187
188
189 SELECT year, (
190

```

12:34 APPLIED COURSE A

SQL commands

```

183 # specify a condition on groups using HAVING.
184
185 Order of execution:
186 1. GROUP BY to create groups
187 2. apply the AGGREGATE FUNCTION
188 3. Apply HAVING condition.
189
190
191 # often used along with GROUP BY. Not Mandatory.
192
193 SELECT name, year FROM movies HAVING year>2000;
194 # HAVING without GROUP BY is same as WHERE
195
196
197 SELECT year, COUNT(year) year_count FROM movies WHERE rankscore>9 GROUP BY year HAVING year_count>20;
198 # HAVING vs WHERE
199 ## WHERE is applied on individual rows while HAVING is applied on groups.
200 ## HAVING is applied after grouping while WHERE is used before grouping.
201
202
203
204
205
206
207 ****
208
209 JOINs:
210
211 #combine data in multiple tables
212
213 # For each movie, print name and the genres
214 SELECT m.name, d.genre FROM movies m JOIN movies_genres g ON m.id=d.movie_id LIMIT 20;
215

```

10:21 APPLIED COURSE A

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.
```

```
*****
```

```
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 LOCKED]
  | LOCK IN SHARE MODE]]
```

Join and Natural Join

```

183 # specify a condition on groups using HAVING.
184
185
186 Order of execution:
187 1. GROUP BY to create groups
188 2. apply the AGGREGATE FUNCTION
189 3. Apply HAVING condition.
190
191
192 # often used along with GROUP BY. Not Mandatory.
193
194 SELECT name, year FROM movies HAVING year>2000;
195 # HAVING without GROUP BY is same as WHERE
196
197
198 SELECT year, COUNT(year) year_count FROM movies WHERE rankscore>9 GROUP BY year HAVING year_count>20;
199 # HAVING vs WHERE
200 ## WHERE is applied on individual rows while HAVING is applied on groups.
201 ## HAVING is applied after grouping while WHERE is used before grouping.
202
203
204
205
206
207 ****
208
209 JOINs:
210
211 #combine data in multiple tables
212
213 # For each movie, print name and the genres
214 SELECT m.name, g.genre FROM movies m JOIN movies_genres g ON m.id=g.movie_id LIMIT 20;
215

```

SQL commands window showing code for HAVING clause. Handwritten notes include:

- 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.

```

208
209 JOINs:
210
211 #combine data in multiple tables
212
213 # For each movie, print name and the genres
214 SELECT m.name, g.genre FROM movies m JOIN movies_genres g ON m.id=g.movie_id LIMIT 20;
215
216 # table aliases: m and g
217
218 # natural join: a join where we have the same column-names across two tables.
219 #T1: C1, C2
220 #T2: C1, C3, C4
221
222 SELECT * FROM T1 JOIN T2;
223
224 SELECT * FROM T1 JOIN T2 USING (C1);
225
226 # returns C1,C2,C3,C4
227 # no need to use the keyword "ON"
228
229
230
231 # Inner join (default) vs left outer vs right outer vs full-outer join.
232
233 SELECT m.name, g.genre FROM movies m LEFT JOIN movies_genres g ON m.id=g.movie_id LIMIT 20;
234
235 #LEFT JOIN or LEFT OUTER JOIN
236 #RIGHT JOIN or RIGHT OUTER JOIN
237 #FULL JOIN or FULL OUTER JOIN
238 #JOIN or INNER JOIN
239
240

```

SQL commands window showing code for JOINs. Handwritten notes include:

- 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.
- 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

```
SQL commands
208 JOINs:
209 #combine data in multiple tables
210
211 # For each movie, print name and the genres
212 SELECT m.name, g.genre from movies m JOIN movies_genres g ON m.id=g.movie_id LIMIT 20;
213
214 # table aliases: m and g
215
216 # natural join a join where we have the same column-names across two tables.
217 #T1: C1, C2
218 #T2: C1, C3, C4
219
220 SELECT * FROM T1 JOIN T2;
221
222 SELECT * FROM T1 JOIN T2 USING (C1);
223
224 # returns C1,C2,C3,C4
225 # no need to use the keyword "ON"
226
227
228
229
230 # Inner join (default) vs left outer vs right outer vs full-outer join.
231
232 SELECT m.name, g.genre from movies m LEFT JOIN movies_genres g ON m.id=g.movie_id LIMIT 20;
233
234 #LEFT JOIN or LEFT OUTER JOIN
235 #RIGHT JOIN or RIGHT OUTER JOIN
236 #FULL JOIN or FULL OUTER JOIN
237 #JOIN or INNER JOIN
```

```
SQL commands
208 JOINs:
209 #combine data in multiple tables
210
211 # For each movie, print name and the genres
212 SELECT m.name, g.genre from movies m JOIN movies_genres g ON m.id=g.movie_id LIMIT 20;
213
214 # table aliases: m and g
215
216 # natural join: a join where we have the same column-names across two tables.
217 #T1: C1, C2
218 #T2: C1, C3, C4
219
220 SELECT * FROM T1 JOIN T2;
221
222 SELECT * FROM T1 JOIN T2 USING (C1);
223
224 # returns C1,C2,C3,C4
225 # no need to use the keyword "ON"
226
227
228
229
230 # Inner join (default) vs left outer vs right outer vs full-outer join.
231
232 SELECT m.name, g.genre from movies m LEFT JOIN movies_genres g ON m.id=g.movie_id LIMIT 20;
233
234 #LEFT JOIN or LEFT OUTER JOIN
235 #RIGHT JOIN or RIGHT OUTER JOIN
236 #FULL JOIN or FULL OUTER JOIN
237 #JOIN or INNER JOIN
```

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

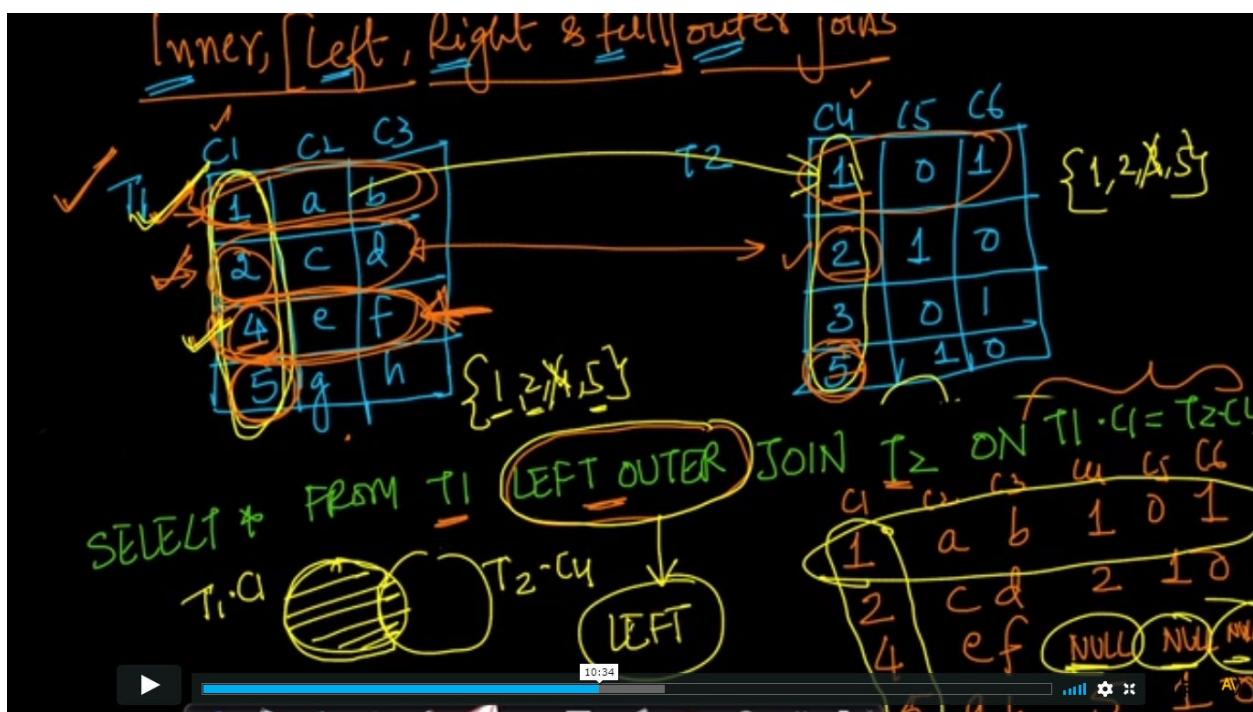
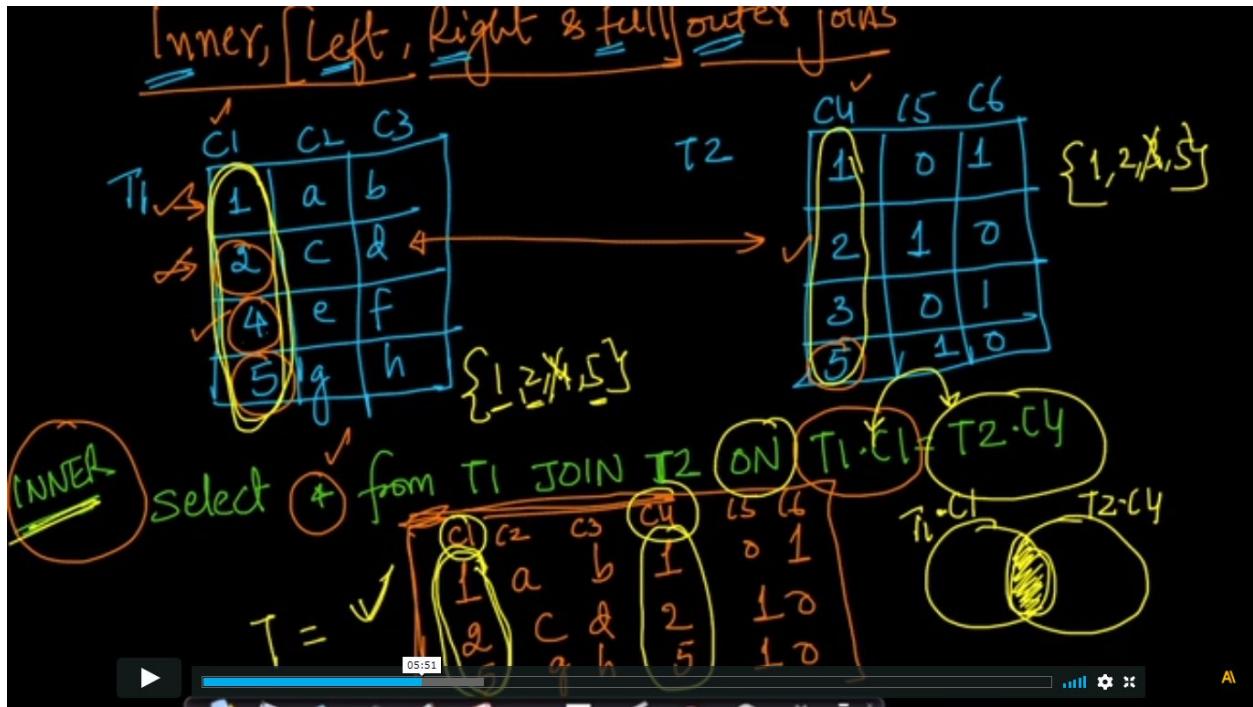
SELECT * FROM T1 JOIN T2;

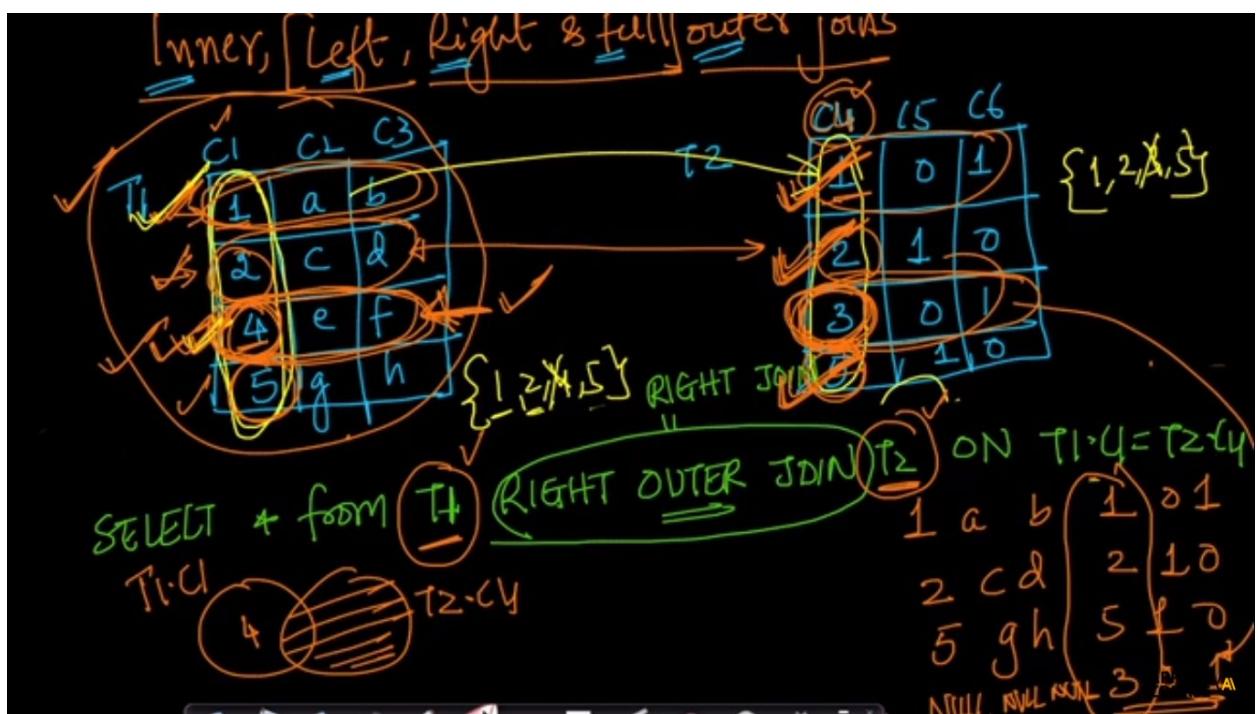
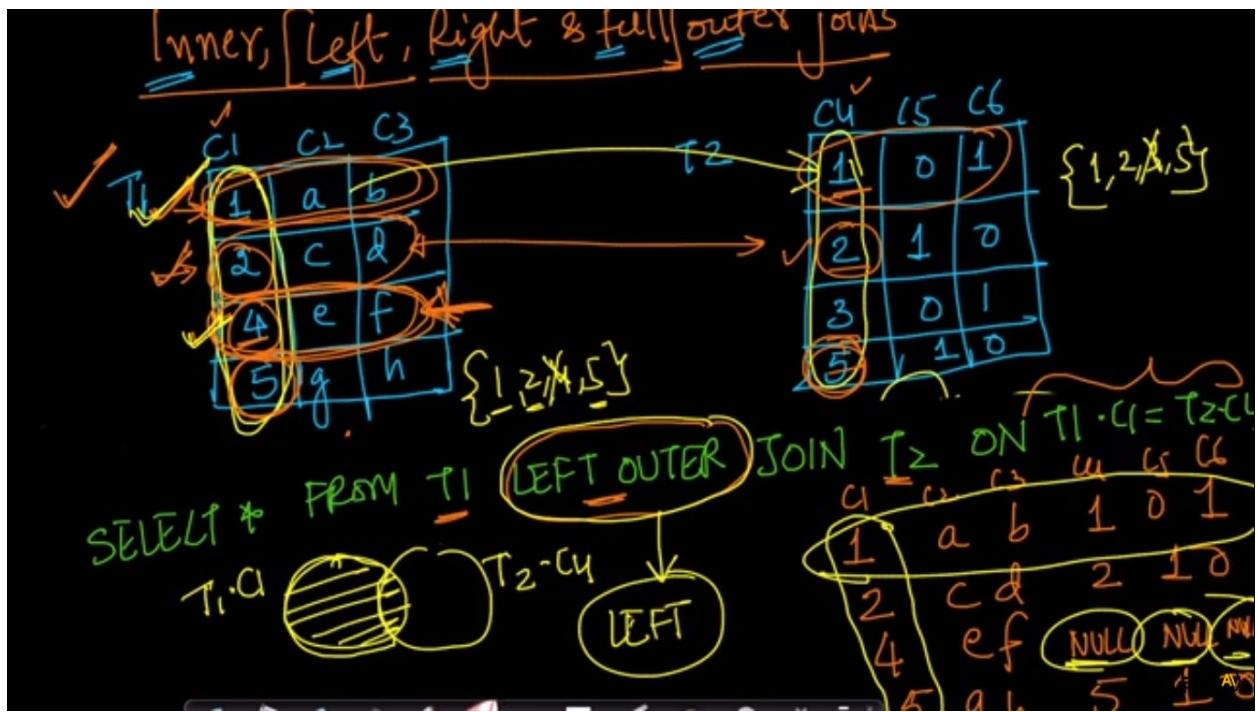
SELECT * FROM T1 JOIN T2 USING (C1);

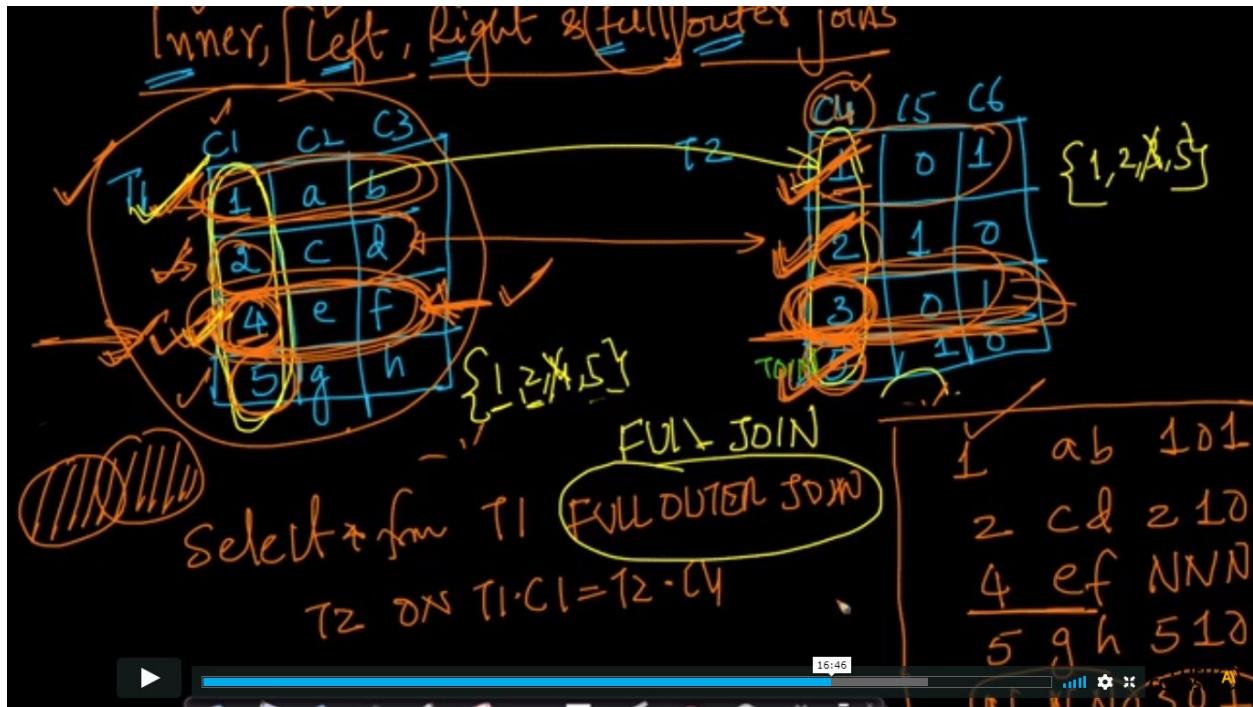
# returns C1,C2,C3,C4
# no need to use the keyword "ON"

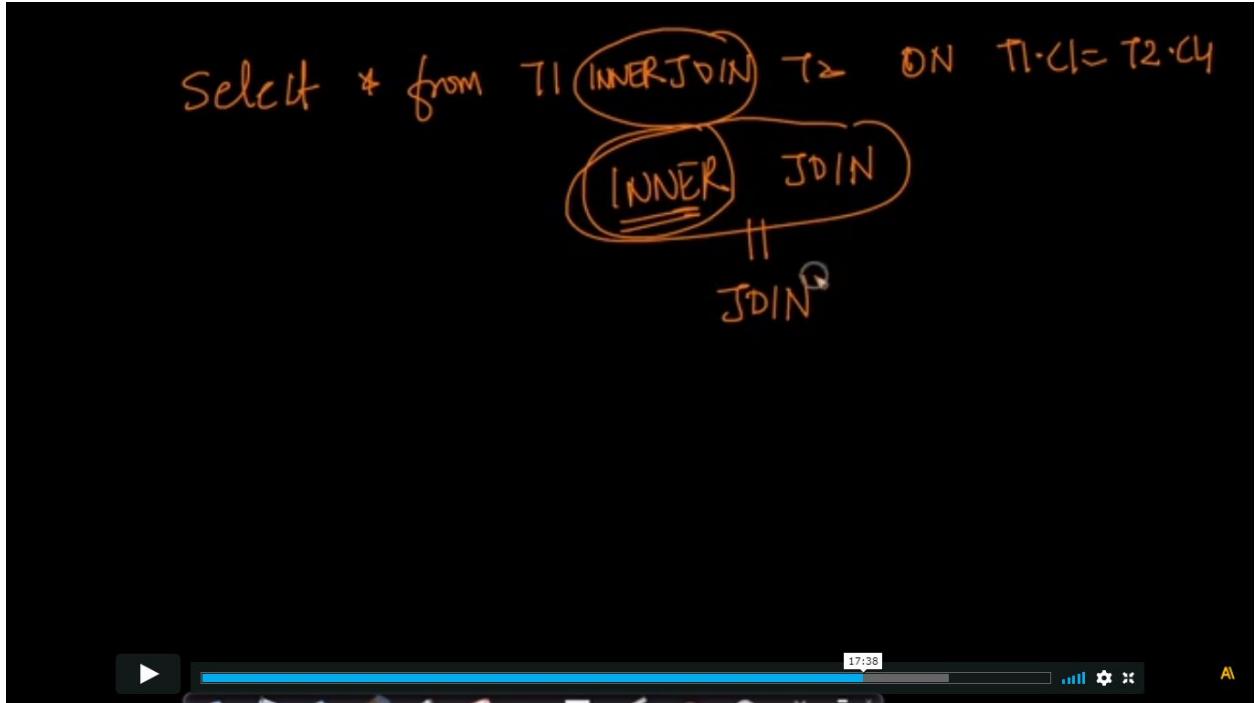
# Inner join (default) vs left outer vs right outer vs full-outer join.
```

Inner, Left, Right and Outer joins.









SQL commands

```

235
236
237 SELECT m.name, g.genre from movies m  LEFT JOIN movies_genres g ON m.id=g.movie_id LIMIT 20
238 #LEFT JOIN or LEFT OUTER JOIN
239 #RIGHT JOIN or RIGHT OUTER JOIN
240 #FULL JOIN or FULL OUTER JOIN
241 #JOIN or INNER JOIN
242
243
244 # NULL for missing counterpart rows.
245
246 # 3-way joins and k-way joins
247 SELECT a.first_name, a.last_name FROM actors a JOIN roles r ON a.id=r.actor_id JOIN movies m on
248 m.id=r.movie_id AND m.name='Officer 444';
249 #Practical note about joins: Joins can be expensive computationally when we have large tables.
250
251 ****
252
253 Sub-Queries: Nested Queries or Inner Queries
254
255 # List all actors in the movie Schindler's List
256 SELECT first_name, last_name from actors WHERE id IN
257     ( SELECT actor_id from roles WHERE movie_id IN
258         (SELECT id FROM movies where name='Schindler's List')
259     );
260 #https://www.imdb.com/title/tt0108052/fullcredits/?ref_=tt_ov_st_sm
261
262 # first the inner query is exuted and then the outer query is executed using the values in the inner query
263
264 # Applied Course

```

Diagram illustrating joins:

- A left join (LJ) is shown as two tables where every row from the left table is matched with all rows from the right table.
- An inner join (IJ) is shown as two tables where only rows from both tables that have matching keys are selected.
- A right join (RJ) is shown as two tables where every row from the right table is matched with all rows from the left table.
- A full outer join (FOJ) is shown as two tables where all rows from both tables are selected, including unmatched rows from both sides.

Notes on joins:

- Joins can be expensive computationally when we have large tables.
- Sub-queries: Nested Queries or Inner Queries
- # Applied Course

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.
```

```
*****
```

Sub Queries/Nested Queries/Inner Queries

SQL commands UNREGISTERED

```

253 ****
254
255 Sub-Queries or Nested Queries or Inner Queries
256 # List all actors in the movie Schindler's List https://www.imdb.com/title/tt0108052/fullcredits/?ref_=tt_ov_st_sm
257
258
259
260
261 SELECT first_name, last_name from actors WHERE id IN
262   ( SELECT actor_id from roles WHERE movie_id IN
263     (SELECT id FROM movies where name='Schindler's List')
264   );
265
266
267
268 # Syntax:
269 SELECT column_name [, column_name ]
270 FROM table1 [, table2 ]
271 WHERE column_name OPERATOR
272   (SELECT column_name [, column_name ]
273    FROM table1 [, table2 ]
274    [WHERE])
275
276 # first the inner query is executed and then the outer query is executed using the output values in
277 # the inner query
278
279 # IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL, Comparison operators
280
281 #EXISTS returns true if the subquery returns one or more records or NULL
282 # ANY operator means TRUE if any of the subquery values meet the condition.
283 # ALL operator means TRUE if all of the subquery values meet the condition.

```

SQL commands
UNREGISTERED

varma — mysql -u root -p — 94x29 jupyter-notebook + Python /Users/varma/Videos — bash

```

mysql> describe actors
-> ;
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id: 16
Current database: imdb

+-----+-----+-----+-----+-----+
| 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> :|

```

```
varma — mysql -u root -p — 94x29
varma — jupyter-notebook - Python ...
varma — /Users/varma/Videos — bash

mysql> describe actors
->;
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id: 16
Current database: imdb

+-----+-----+-----+-----+-----+
| 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> d cri
```

```
varma — mysql -u root -p — 94x29
varma — jupyter-notebook - Python ...
varma — /Users/varma/Videos — bash

+-----+-----+-----+-----+-----+
| 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)

mysql>
```

Handwritten annotations:

- circled "id" in the first row of the "actors" table.
- circled "first_name" in the second row of the "actors" table.
- circled "last_name" in the third row of the "actors" table.
- circled "gender" in the fourth row of the "actors" table.
- circled "actor_id" in the first row of the "roles" table.
- circled "movie_id" in the second row of the "roles" table.
- circled "role" in the third row of the "roles" table.
- circled "id" in the first row of the "movies" table.
- circled "name" in the second row of the "movies" table.
- circled "year" in the third row of the "movies" table.
- circled "rankscore" in the fourth row of the "movies" table.

Labels written on the right side of the screen:

- "adrs" near the top of the "roles" table.
- "roles" near the bottom of the "roles" table.
- "movies" near the bottom of the "movies" table.

```
SQL commands SQL commands UNREGISTERED
253 ****
254
255 Sub-Queries or Nested Queries or Inner Queries
256
257 # List all actors in the movie Schindler's List
258 #https://www.imdb.com/title/tt0108052/fullcredits/?ref\_=tt\_ov\_st\_sm
259
260
261 SELECT first_name, last_name from actors WHERE id IN
262 [ ( SELECT actor_id from roles WHERE movie_id IN
263 [ (SELECT id FROM movies where name='Schindler's List)
264 );
265
266
267
268 # Syntax:
269 SELECT column_name [, column_name ]
270 FROM table1 [, table2 ]
271 WHERE column_name OPERATOR
272 (SELECT column_name [, column_name ]
273 FROM table1 [, table2 ]
274 [WHERE])
275
276 # first the inner query is executed and then the outer query is executed using the output values in
277 # the inner query
278
279 # IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL, Comparison operators
280
281 #EXISTS returns true if the subquery returns one or more records or NULL
282 # ANY operator returns TRUE if any of the subquery values meet the condition.
283 # ALL operator returns TRUE if all of the subquery values meet the condition.
284
```

Sets

(IN) {1,2,3,4,5}

2 E {1,2,3,4,5}

```
SQL commands SQL commands UNREGISTERED
253 ****
254
255 Sub-Queries or Nested Queries or Inner Queries
256
257 # List all actors in the movie Schindler's List
258 #https://www.imdb.com/title/tt0108052/fullcredits/?ref\_=tt\_ov\_st\_sm
259
260
261 SELECT first_name, last_name from actors WHERE id IN
262 [ ( SELECT actor_id from roles WHERE movie_id IN
263 [ (SELECT id FROM movies where name='Schindler's List)
264 );
265
266
267
268 # Syntax:
269 SELECT column_name [, column_name ]
270 FROM table1 [, table2 ]
271 WHERE column_name OPERATOR
272 (SELECT column_name [, column_name ]
273 FROM table1 [, table2 ]
274 [WHERE])
275
276 # first the inner query is executed and then the outer query is executed using the output values in
277 # the inner query
278
279 # IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL, Comparison operators
280
281 #EXISTS returns true if the subquery returns one or more records or NULL
282 # ANY operator returns TRUE if any of the subquery values meet the condition.
283 # ALL operator returns TRUE if all of the subquery values meet the condition.
284
```

for ()

{ fn() }

}

```
SQL commands UNREGISTERED
253 ****
254
255 Sub-Queries or Nested Queries or Inner Queries
256
257 # List all actors in the movie Schindler's List
258 #https://www.imdb.com/title/tt0108052/fullcredits/?ref\_=tt\_ov\_st\_sm
259
260 SELECT first_name, last_name from actors WHERE id IN
261 ( SELECT actor_id from roles WHERE movie_id IN
262 ( SELECT id FROM movies where name='Schindler's List')
263 );
264
265
266
267
268 # Syntax:
269 SELECT column_name [, column_name ]
270 FROM table1 [, table2 ]
271 WHERE column_name OPERATOR
272 (SELECT column_name [, column_name ]
273 FROM table1 [, table2 ]
274 [WHERE])
275
276 # first the innner query is executed and then the outer query is executed using the output values in
277 # the inner query
278
279 # IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL, Comparison operators
280
281 #EXISTS returns true if the subquery returns one or more records or NULL
282 # ANY operator returns TRUE if any of the query values meet the condition.
283 # ALL operator returns TRUE if all query values meet the condition.
```

SQL commands UNREGISTERED

253 ****

254

255 Sub-Queries or Nested Queries or Inner Queries

256

257 # List all actors in the movie Schindler's List

258 #https://www.imdb.com/title/tt0108052/fullcredits/?ref_=tt_ov_st_sm

259

260 SELECT first_name, last_name from actors WHERE id IN

261 (SELECT actor_id from roles WHERE movie_id IN

262 (SELECT id FROM movies where name='Schindler's List')

263);

264

265

266

267

268 # Syntax:

269 SELECT column_name [, column_name]

270 FROM table1 [, table2]

271 WHERE column_name OPERATOR

272 (SELECT column_name [, column_name]

273 FROM table1 [, table2]

274 [WHERE])

275

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

277 # the inner query

278

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

280

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

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

283 # ALL operator

```
SQL commands UNREGISTERED
252
253 ****
254
255 Sub-Queries or Nested Queries or Inner Queries
256
257 # List all actors in the movie Schindler's List
258 #https://www.imdb.com/title/tt0108052/fullcredits/?ref\_=tt\_ov\_st\_sm
259
260 SELECT first_name, last_name from actors WHERE id IN
261 ( SELECT actor_id from roles WHERE movie_id IN
262 ( SELECT id FROM movies where name='Schindler's List')
263 );
264
265
266
267
268 # Syntax:
269 SELECT column_name [, column_name ]
270 FROM table1 [, table2 ]
271 WHERE column_name OPERATOR
272 (SELECT column_name [, column_name ]
273 FROM table1 [, table2 ]
274 [WHERE])
275
276 # first the innner query is executed and then the outer query is executed using the output values in
277 # the inner query
278
279 # IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL, Comparison operators
280
281 #EXISTS returns true if the subquery returns one or more records or NULL
282 # ANY operator returns TRUE if any of the query values meet the condition.
283 # ALL operator
```

SQL commands UNREGISTERED

252

253 ****

254

255 Sub-Queries or Nested Queries or Inner Queries

256

257 # List all actors in the movie Schindler's List

258 #https://www.imdb.com/title/tt0108052/fullcredits/?ref_=tt_ov_st_sm

259

260 SELECT first_name, last_name from actors WHERE id IN

261 (SELECT actor_id from roles WHERE movie_id IN

262 (SELECT id FROM movies where name='Schindler's List')
263);
264

265

266

267

268 # Syntax:

269 SELECT column_name [, column_name]

270 FROM table1 [, table2]

271 WHERE column_name OPERATOR

272 (SELECT column_name [, column_name]

273 FROM table1 [, table2]

274 [WHERE])

275

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

277 # the inner query

278

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

280

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

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

283 # ALL operator

```
SQL commands UNREGISTERED
264 );
265
266
267
268 # Syntax:
269 SELECT column_name [, column_name ]
270 FROM table1 [, table2 ]
271 WHERE column_name OPERATOR
272   (SELECT column_name [, column_name ]
273    FROM table1 [, table2 ]
274    [WHERE])
275
276 # first the inner query is executed and then the outer query is executed using the output values in
277 # the inner query
278
279 # IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL, Comparison operators
280
281 #EXISTS returns true if the subquery returns one or more records or NULL
282 # ANY operator returns TRUE if any of the subquery values meet the condition.
283 # ALL operator returns TRUE if all of the subquery values meet the condition.
284
285
286
287 SELECT * FROM movies where rankscore >= ALL (SELECT MAX(rankscore) from movies);
288 # get all movies whose rankscore is same as the maximum rankscore.
289
290 # e.g: rankscore <= ALL....
291
292 # https://en.wikipedia.org/wiki/Correlated_subquery
293
294 ****
295
296
297
298
299
```

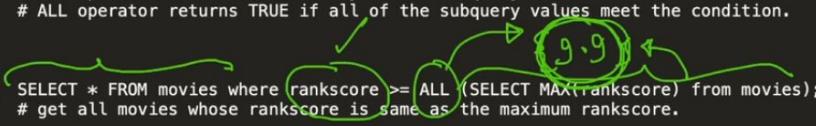
EXISTS
Σ {1,2,3}
NULL

```
SQL commands UNREGISTERED
268 # Syntax:
269 SELECT column_name [, column_name ]
270 FROM table1 [, table2 ]
271 WHERE column_name OPERATOR
272   (SELECT column_name [, column_name ]
273    FROM table1 [, table2 ]
274    [WHERE])
275
276 # first the inner query is executed and then the outer query is executed using the output values in
277 # the inner query
278
279 # IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL, Comparison operators
280
281 #EXISTS returns true if the subquery returns one or more records or NULL
282 # ANY operator returns TRUE if any of the subquery values meet the condition.
283 # ALL operator returns TRUE if all of the subquery values meet the condition.
284
285
286
287 SELECT * FROM movies where rankscore >= ALL (SELECT MAX(rankscore) from movies);
288 # get all movies whose rankscore is same as the maximum rankscore.
289
290 # e.g: rankscore <= ALL....
291
292 # https://en.wikipedia.org/wiki/Correlated_subquery
293
294 ****
295
296 Data Manipulation Language:
297
298 INSERT INTO
299
```

```
SQL commands UNREGISTERED
268 # Syntax:
269 SELECT column_name [, column_name ]
270 FROM table1 [, table2 ]
271 WHERE column_name OPERATOR
272   (SELECT column_name [, column_name ]
273    FROM table1 [, table2 ]
274   [WHERE])
275
276 # first the inner query is executed and then the outer query is executed using the output values in
277 # the inner query
278
279 # IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL, Comparison operators
280
281 #EXISTS returns true if the subquery returns one or more records or NULL
282 # ANY operator returns TRUE if any of the subquery values meet the condition.
283 # ALL operator returns TRUE if all of the subquery values meet the condition.
284
285
286
287 SELECT * FROM movies where rankscore >= ALL (SELECT MAX(rankscore) from movies);
288 # get all movies whose rankscore is same as the maximum rankscore.
289
290 # e.g: rankscore <> ALL(...)
291
292 # https://en.wikipedia.org/wiki/Correlated_subquery
293
294 ****
295
296 Data Manipulation Language:
297
298 INSERT INTO
299
300
```

```
SQL commands UNREGISTERED
268 # Syntax:
269 SELECT column_name [, column_name ]
270 FROM table1 [, table2 ]
271 WHERE column_name OPERATOR
272   (SELECT column_name [, column_name ]
273    FROM table1 [, table2 ]
274   [WHERE])
275
276 # first the inner query is executed and then the outer query is executed using the output values in
277 # the inner query
278
279 # IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL, Comparison operators
280
281 #EXISTS returns true if the subquery returns one or more records or NULL
282 # ANY operator returns TRUE if any of the subquery values meet the condition.
283 # ALL operator returns TRUE if all of the subquery values meet the condition.
284
285
286
287 SELECT * FROM movies where rankscore >= ALL (SELECT MAX(rankscore) from movies);
288 # get all movies whose rankscore is same as the maximum rankscore.
289
290 # e.g: rankscore <> ALL(...)
291
292 # https://en.wikipedia.org/wiki/Correlated_subquery
293
294 ****
295
296 Data Manipulation Language:
297
298 INSERT INTO
299
300
```

```
SQL commands UNREGISTERED
268 # Syntax:
269 SELECT column_name [, column_name ]
270 FROM table1 [, table2 ]
271 WHERE column_name OPERATOR
272   (SELECT column_name [, column_name ]
273    FROM table1 [, table2 ]
274   [WHERE])
275
276 # first the inner query is executed and then the outer query is executed using the output values in
277 # the inner query
278
279 # IN, NOT IN, EXISTS, NOT EXISTS, ANY, ALL, Comparison operators
280
281 #EXISTS returns true if the subquery returns one or more records or NULL
282 # ANY operator returns TRUE if any of the subquery values meet the condition.
283 # ALL operator returns TRUE if all of the subquery values meet the condition.
284
285
286
287 SELECT * FROM movies where rankscore >= ALL (SELECT MAX(rankscore) from movies);
288 # get all movies whose rankscore is same as the maximum rankscore.
289
290 # e.g: rankscore <= ALL(...)
291
292 # https://en.wikipedia.org/wiki/Correlated_subquery
293
294 ****
295
296 Data Manipulation Language:
297
298 INSERT INTO
299
300
```



APPLIED COURSE A

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

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

alias: emp
inner query
per dept
emp-sal > avg(dept)

In the above query the outer query is

```
SELECT employee_number, name
FROM employees emp
WHERE salary > ...
```

and the inner query (the correlated subquery) is

```
SELECT AVG(salary)
FROM employees
WHERE department = emp.department
```

19:24 | APPLIED COURSE

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

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

emp 1
1 e1 d1 b1
2 e2 d2 b2

In the above query the outer query is

```
SELECT employee_number, name
FROM employees emp
WHERE salary > ...
```

and the inner query (the correlated subquery) is

```
SELECT AVG(salary)
FROM employees
WHERE department = emp.department
```

Query-optimizer

22:25 | APPLIED COURSE

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 inner 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
```

```
DML:INSERT
```

```
*****
```

Data Manupulation 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 nnested sub query:

[https://en.wikipedia.org/wiki/Insert_\(SQL\)#Copying_rows_from_other_tables](https://en.wikipedia.org/wiki/Insert_(SQL)#Copying_rows_from_other_tables)

DML:UPDATE , DELETE

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.

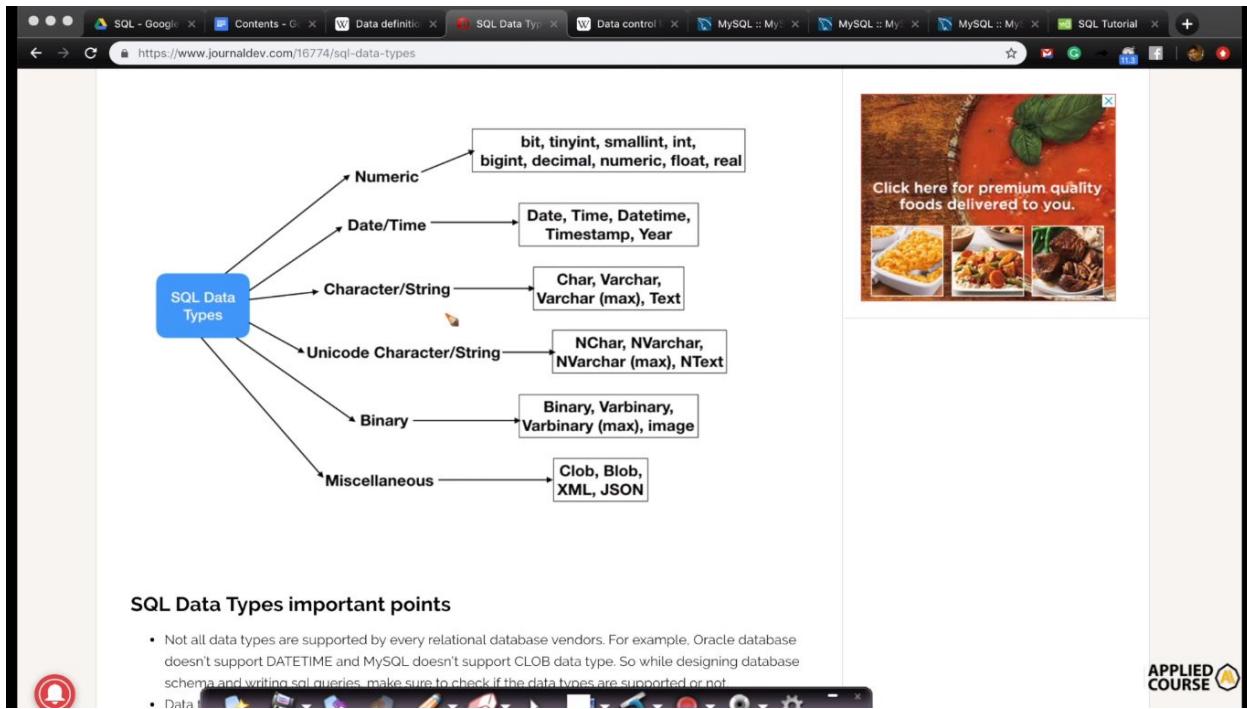
#DELETE

```
DELETE FROM movies WHERE id=412321;
```

Remove all rows: TRUNCATE TABLE TableName;

Same as selete without a WHERE Clause.

DDL:CREATE TABLE



Data Definition Language

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

DDL:ALTER: ADD, MODIFY, DROP

ALTER: ADD, MODIFY, DROP

ALTER TABLE language ADD country VARCHAR(50);

ALTER TABLE language MODIFY country VARCHAR(60);

ALTER TABLE langauge DROP country;

*****D

DDL:DROP TABLE, TRUNCATE, DELETE

Removes both the table and all of the data permanently.

DROP TABLE Tablename;

DROP TABLE TableName IF 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: GRANT, REVOKE

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>

Learning resources

<https://www.w3schools.com/sql/>

<https://dev.mysql.com/doc/refman/8.0/en/>
