Milestone 2 Report
CSE 560 - Data Models and Query Language
Team Name – Triangulation

Teammates :

Shivani Reddy Katta
UB Person Number - 50412911 | skatta@buffalo.edu
Aditya Nongmeikapam
UB Person Number - 50418825 | adityano@buffalo.edu
Jayesh Kishor Suryavanshi
UB Person Number - 50417114 | jayeshki@buffalo.edu

1 Problem Statement

When it comes to users, they can decide which movie/series they wish to watch based on the parameters like rating, genres, actor/actress, director and much more. Lack of a database will result in the user search to be really difficult. As movies and series often contain a lot of parts/episodes, this complexity increases even more. It will be a challenge to really get the desired movie or a popular TV show if the proper organized database structure is absent. With movies, TV shows come various other attributes like crew, directors, writers, the title and runtime of that show. There may be even more number of attributes associated with the above data. If the user wants to retrieve information about the data he/she wants, it should be really accessible.

2 Proposed Solution

The proposed solution involves storing all the available data of movies/TV shows of a particular time frame in an organised and structured way. This solution aims to solve the problems faced by users when they want to search for the information that they want. Once this data is stored, we can retrieve the required tuple/tuples from the tables with certain conditions like top comedy TV show, top movies of all times, lowest rated action movies, crew with most successful movie ratings, best actor of all time in drama genre and much more.

3 Why not conventional approach?

As the amount of data grows, even powerful tools like excel don’t deliver the expected results. On the other hand, Structured Query Language is much faster as the amount of data grows exponentially. Granular access control is not possible in conventional tools like Excel. This is one of the reasons why SQL is preferred in Business Intelligence tools in large organisations which process huge amounts of data on a day to day basis. Even Big Data tools like Spark, Impala use SQL. Few advantages of SQL are Faster query processing, standardized syntax, portable, interactive and offers multiple data views. Also, Modification/Manipulation of data and database table such as insertion, deletion and updation is really convenient. It is difficult to store large data in Excel sheet and SQL is much faster than Excel when data gets larger. Can’t use join feature in the Excel which is most frequently used while querying. Granular access control is not possible in Excel. Updating the schema will be difficult in Excel when compared to database system. Sharing of Excel file and maintaining a consistent state is difficult in Excel when compared to Database systems.

4 Target Users

A User is someone who watches a Movie/TV Show, even companies who want to know the statistics of movies and TV shows. Someone who researches (like if they want know a season with highly rated episodes, Director how directed more films).

5 User Privileges, Administrators

The Users have the view access i.e. they cannot perform CRUD operations on the data that they wish to retrieve. However, an administrator is someone who can perform create, read, update, delete operations of the database. The administrator is generally the company/group which owns the database.

6 How to solve real life scenario

Suppose a user who wants to stream data which can be in any form like movie, TV show. game streams etc. The video streaming sites on which the user is consuming content can query in our database based on the inputs given by the user like a particular genre, language, director, actor, most rated, most viewed etc. Users can add reviews/ratings after they watch the content and the database gets updated.

7 Entity-Relationship (E/R) Diagram

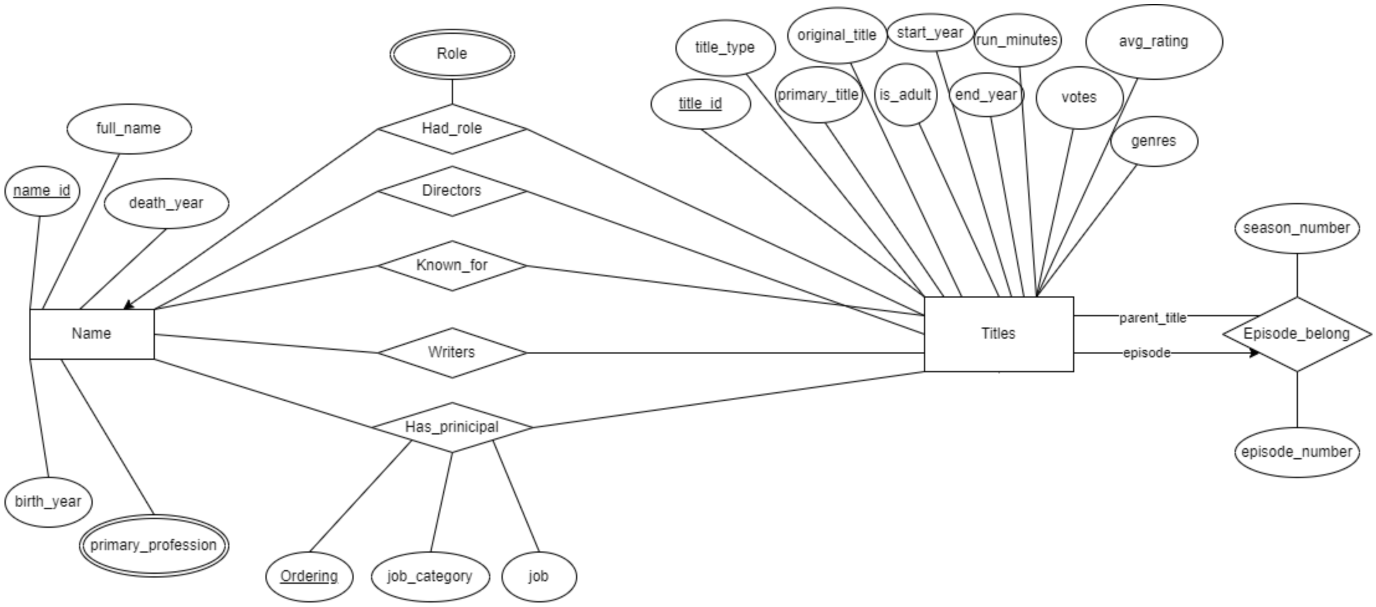


Fig1 . ER Diagram

8 Logical Schema Diagram

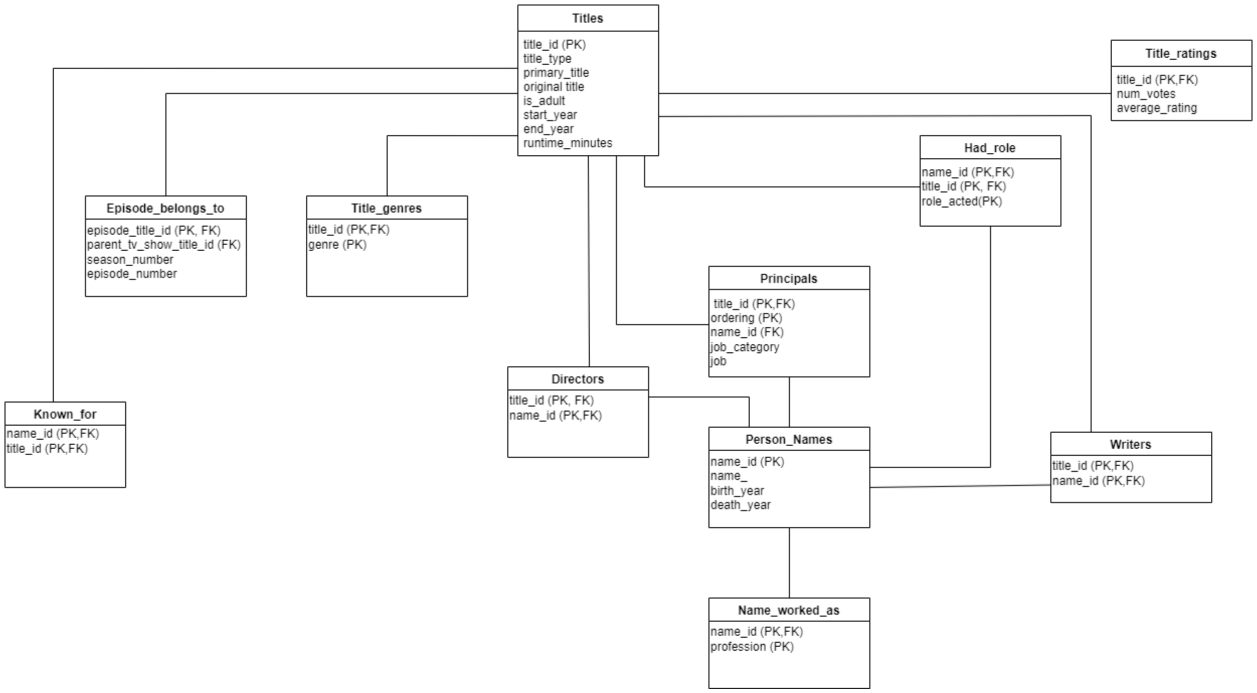
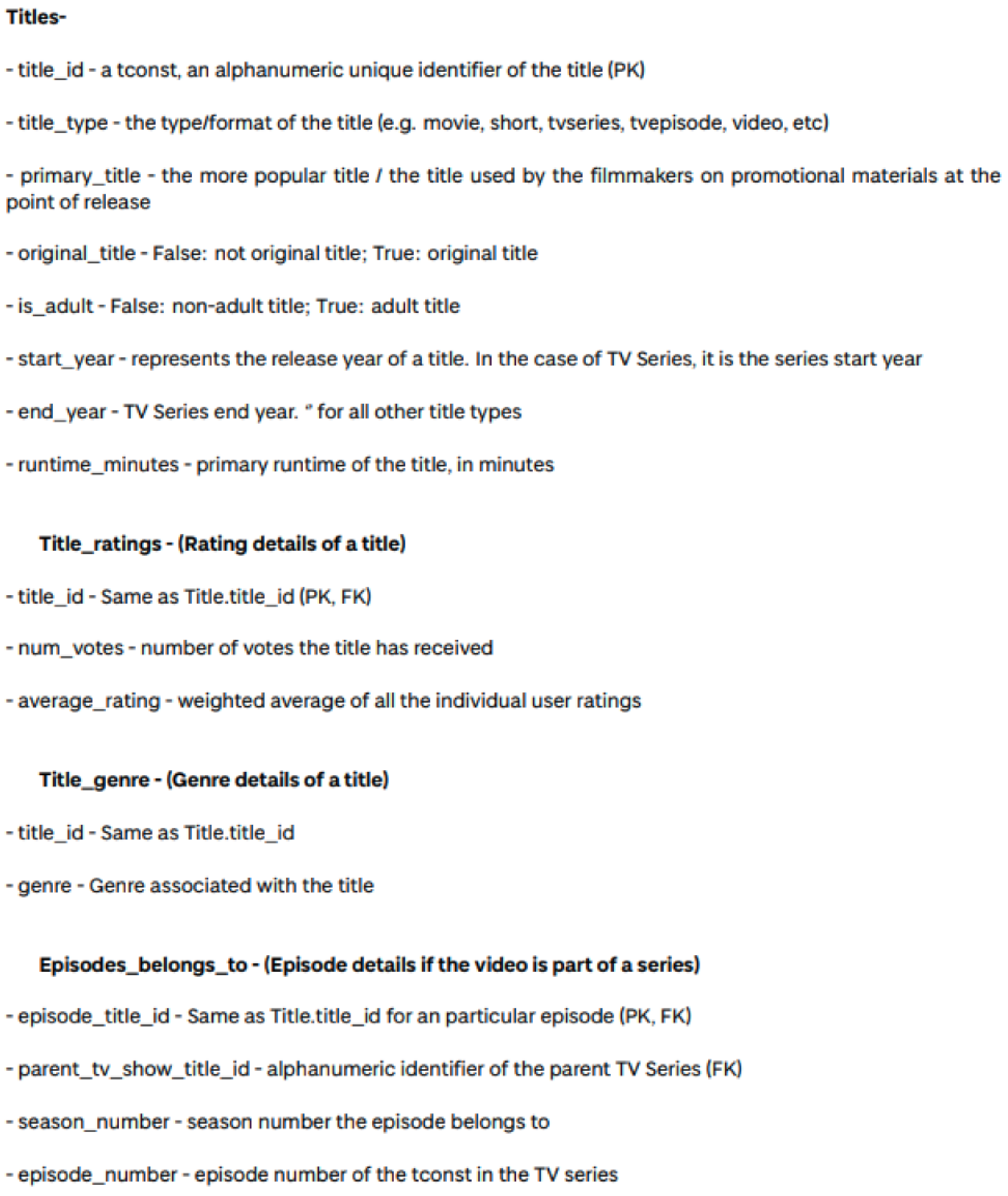


Figure 2: IMDB Database Logical Schema Diagram

9 Attribute Definitions



Titles-

- title_id - a tconst, an alphanumeric unique identifier of the title (PK)

- title_type - the type/format of the title (e.g. movie, short, tvseries, tvepisode, video, etc)

- primary_title - the more popular title / the title used by the filmmakers on promotional materials at the point of release

- original_title - False: not original title; True: original title

- is_adult - False: non-adult title; True: adult title

- start_year - represents the release year of a title. In the case of TV Series, it is the series start year

- end_year - TV Series end year. ‘’ for all other title types

- runtime_minutes - primary runtime of the title, in minutes

Title_ratings - (Rating details of a title)

- title_id - Same as Title.title_id (PK, FK)

- num_votes - number of votes the title has received

- average_rating - weighted average of all the individual user ratings

Title_genre - (Genre details of a title)

- title_id - Same as Title.title_id

- genre - Genre associated with the title

Episodes_belongs_to - (Episode details if the video is part of a series)

- episode_title_id - Same as Title.title_id for an particular episode (PK, FK)

- parent_tv_show_title_id - alphanumeric identifier of the parent TV Series (FK)

- season_number - season number the episode belongs to

- episode_number - episode number of the tconst in the TV series

Principals - (Contains principal cast/crew details)

- title_id - Same as Title.title_id (PK,FK)
- ordering - a number to uniquely identify rows for a given titleId (PK)
- name_id - Name_id of the actor (FK)
- job_category - the category of job that person was in
- job - the specific job title if applicable else NULL

Directors - (Details of the director of the title)

- title_id - Same as Title.title_id for the person has directed (PK,FK)
- name_id - Name_id of the director (PK,FK) (Movie can have multiple directors)

Writers - (Details of the writers)

- title_id - Same as Title.title_id for the person has writen the script (PK,FK)
- name_id - Name_id of the writer (PK,FK) [A movie can have multiple directors]

Person_names - (Details of a person)

- name_id - alphanumeric unique identifier of the name/person (PK)
- full_name - name by which the person is most often credited
- birth_year - in YYYY format
- death_year - in YYYY format if applicable, else NULL

Name_worked_as - (Profession of an person)

- name_id - Name_id of the person (PK,FK)
- profession - Name of the profession (PK) (Pesron can have multiple profession)

Known_for (List of title for which a person is known for)

- name_id - Name_id of the person (PK,FK)
- title_id - Same as Title.title_id, One of the title for which the person is known for. (PK,FK)

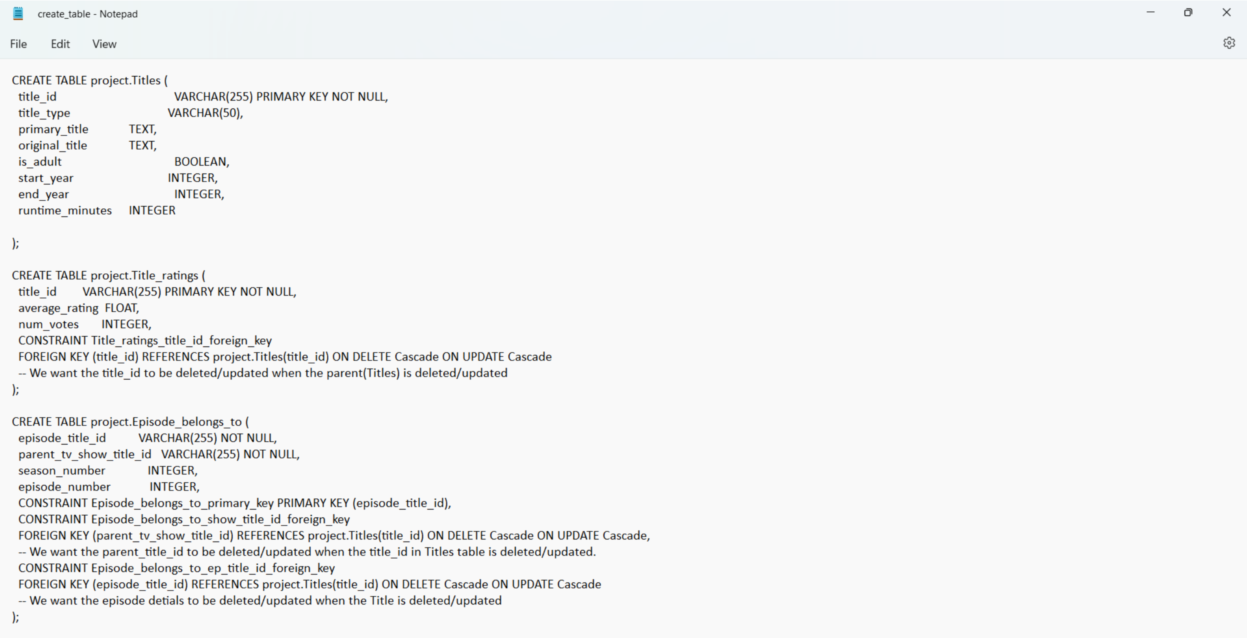
Had_role - (Role details of an actor)

- name_id - Name_id of the actor (PK,FK)

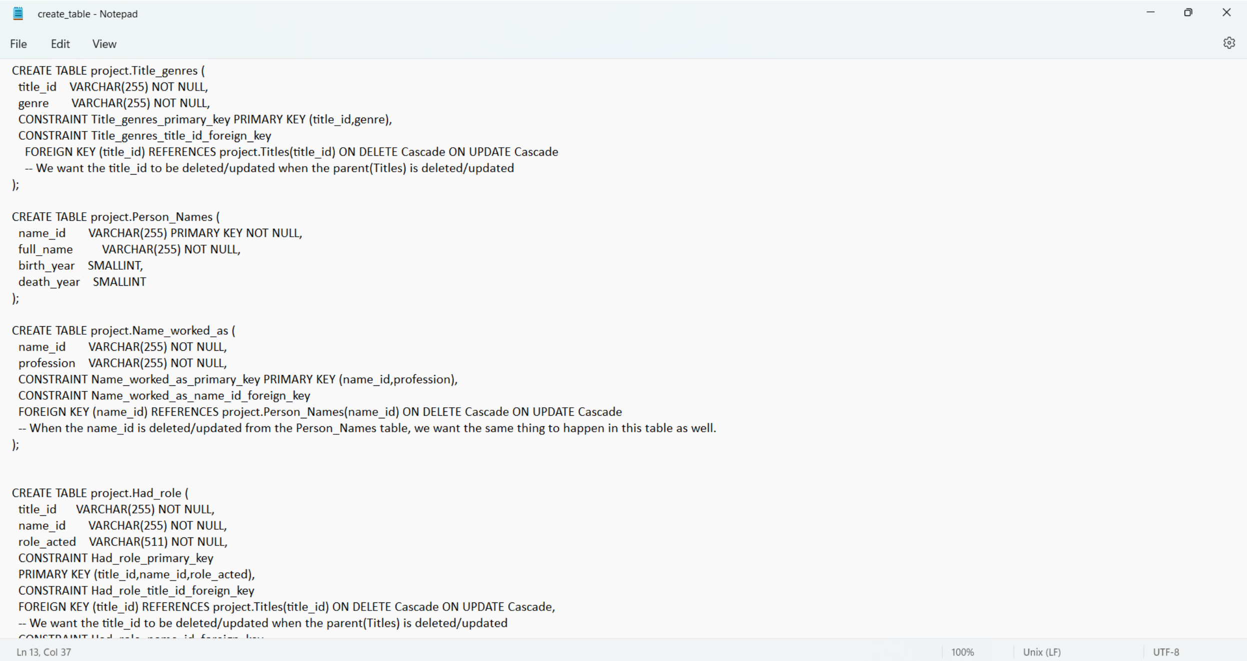
- title_id - Title id of the movie/video/series etc (PK,FK)

- role_acted - name of the role (PK) [Because a person can have multiple roles]

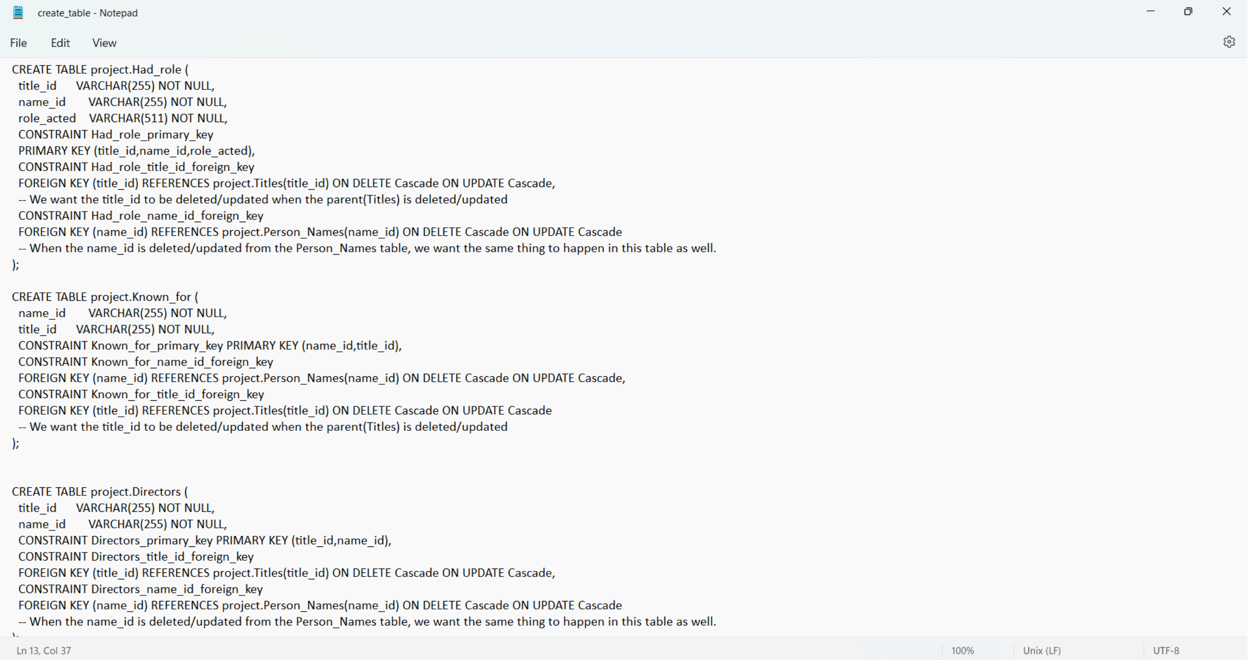
10. Constraints



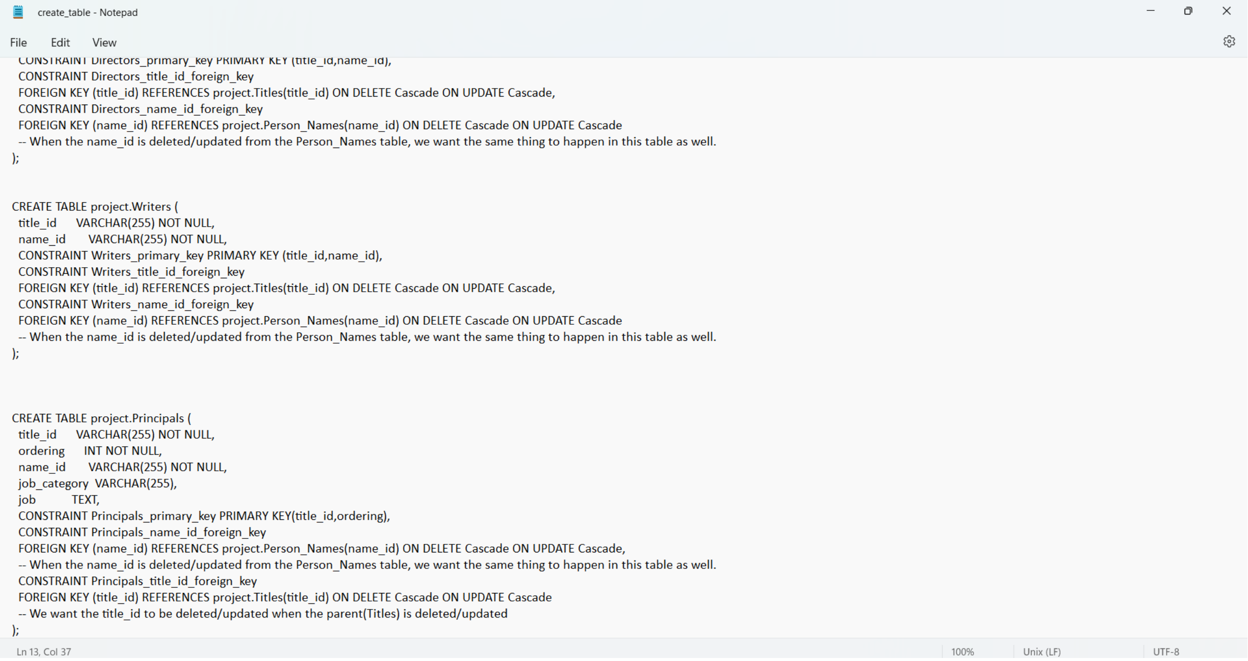
```
CREATE TABLE project.Titles (
  title_id                        VARCHAR(255) PRIMARY KEY NOT NULL,
  title_type                      VARCHAR(50),
  primary_title         TEXT,
  original_title        TEXT,
  is_adult                        BOOLEAN,
  start_year                      INTEGER,
  end_year                        INTEGER,
  runtime_minutes      INTEGER

);

CREATE TABLE project.Title_ratings (
  title_id       VARCHAR(255) PRIMARY KEY NOT NULL,
  average_rating  FLOAT,
  num_votes      INTEGER,
  CONSTRAINT Title_ratings_title_id_foreign_key
  FOREIGN KEY (title_id) REFERENCES project.Titles(title_id) ON DELETE Cascade ON UPDATE Cascade
  -- We want the title_id to be deleted/updated when the parent(Titles) is deleted/updated
);

CREATE TABLE project.Episode_belongs_to (
  episode_title_id          VARCHAR(255) NOT NULL,
  parent_tv_show_title_id   VARCHAR(255) NOT NULL,
  season_number             INTEGER,
  episode_number            INTEGER,
  CONSTRAINT Episode_belongs_to_primary_key PRIMARY KEY (episode_title_id),
  CONSTRAINT Episode_belongs_to_show_title_id_foreign_key
  FOREIGN KEY (parent_tv_show_title_id) REFERENCES project.Titles(title_id) ON DELETE Cascade ON UPDATE Cascade,
  -- We want the parent_title_id to be deleted/updated when the title_id in Titles table is deleted/updated.
  CONSTRAINT Episode_belongs_to_ep_title_id_foreign_key
  FOREIGN KEY (episode_title_id) REFERENCES project.Titles(title_id) ON DELETE Cascade ON UPDATE Cascade
  -- We want the episode detials to be deleted/updated when the Title is deleted/updated
);
```



```
CREATE TABLE project.Title_genres (
  title_id    VARCHAR(255) NOT NULL,
  genre       VARCHAR(255) NOT NULL,
  CONSTRAINT Title_genres_primary_key PRIMARY KEY (title_id,genre),
  CONSTRAINT Title_genres_title_id_foreign_key
    FOREIGN KEY (title_id) REFERENCES project.Titles(title_id) ON DELETE Cascade ON UPDATE Cascade
    -- We want the title_id to be deleted/updated when the parent(Titles) is deleted/updated
);

CREATE TABLE project.Person_Names (
  name_id       VARCHAR(255) PRIMARY KEY NOT NULL,
  full_name       VARCHAR(255) NOT NULL,
  birth_year    SMALLINT,
  death_year    SMALLINT
);

CREATE TABLE project.Name_worked_as (
  name_id       VARCHAR(255) NOT NULL,
  profession    VARCHAR(255) NOT NULL,
  CONSTRAINT Name_worked_as_primary_key PRIMARY KEY (name_id,profession),
  CONSTRAINT Name_worked_as_name_id_foreign_key
  FOREIGN KEY (name_id) REFERENCES project.Person_Names(name_id) ON DELETE Cascade ON UPDATE Cascade
  -- When the name_id is deleted/updated from the Person_Names table, we want the same thing to happen in this table as well.
);


CREATE TABLE project.Had_role (
  title_id      VARCHAR(255) NOT NULL,
  name_id       VARCHAR(255) NOT NULL,
  role_acted    VARCHAR(511) NOT NULL,
  CONSTRAINT Had_role_primary_key
  PRIMARY KEY (title_id,name_id,role_acted),
  CONSTRAINT Had_role_title_id_foreign_key
  FOREIGN KEY (title_id) REFERENCES project.Titles(title_id) ON DELETE Cascade ON UPDATE Cascade,
  -- We want the title_id to be deleted/updated when the parent(Titles) is deleted/updated
  CONSTRAINT Had_role_name_id_foreign_key
```



```
CREATE TABLE project.Had_role (
 title_id      VARCHAR(255) NOT NULL,
 name_id       VARCHAR(255) NOT NULL,
 role_acted    VARCHAR(511) NOT NULL,
 CONSTRAINT Had_role_primary_key
 PRIMARY KEY (title_id,name_id,role_acted),
 CONSTRAINT Had_role_title_id_foreign_key
 FOREIGN KEY (title_id) REFERENCES project.Titles(title_id) ON DELETE Cascade ON UPDATE Cascade,
 -- We want the title_id to be deleted/updated when the parent(Titles) is deleted/updated
 CONSTRAINT Had_role_name_id_foreign_key
 FOREIGN KEY (name_id) REFERENCES project.Person_Names(name_id) ON DELETE Cascade ON UPDATE Cascade
 -- When the name_id is deleted/updated from the Person_Names table, we want the same thing to happen in this table as well.
);

CREATE TABLE project.Known_for (
 name_id       VARCHAR(255) NOT NULL,
 title_id      VARCHAR(255) NOT NULL,
 CONSTRAINT Known_for_primary_key PRIMARY KEY (name_id,title_id),
 CONSTRAINT Known_for_name_id_foreign_key
 FOREIGN KEY (name_id) REFERENCES project.Person_Names(name_id) ON DELETE Cascade ON UPDATE Cascade,
 CONSTRAINT Known_for_title_id_foreign_key
 FOREIGN KEY (title_id) REFERENCES project.Titles(title_id) ON DELETE Cascade ON UPDATE Cascade
 -- We want the title_id to be deleted/updated when the parent(Titles) is deleted/updated
);


CREATE TABLE project.Directors (
 title_id      VARCHAR(255) NOT NULL,
 name_id       VARCHAR(255) NOT NULL,
 CONSTRAINT Directors_primary_key PRIMARY KEY (title_id,name_id),
 CONSTRAINT Directors_title_id_foreign_key
 FOREIGN KEY (title_id) REFERENCES project.Titles(title_id) ON DELETE Cascade ON UPDATE Cascade,
 CONSTRAINT Directors_name_id_foreign_key
 FOREIGN KEY (name_id) REFERENCES project.Person_Names(name_id) ON DELETE Cascade ON UPDATE Cascade
 -- When the name_id is deleted/updated from the Person_Names table, we want the same thing to happen in this table as well.
);
```



```
 CONSTRAINT Directors_primary_key PRIMARY KEY (title_id,name_id),
 CONSTRAINT Directors_title_id_foreign_key
 FOREIGN KEY (title_id) REFERENCES project.Titles(title_id) ON DELETE Cascade ON UPDATE Cascade,
 CONSTRAINT Directors_name_id_foreign_key
 FOREIGN KEY (name_id) REFERENCES project.Person_Names(name_id) ON DELETE Cascade ON UPDATE Cascade
 -- When the name_id is deleted/updated from the Person_Names table, we want the same thing to happen in this table as well.
);


CREATE TABLE project.Writers (
 title_id      VARCHAR(255) NOT NULL,
 name_id       VARCHAR(255) NOT NULL,
 CONSTRAINT Writers_primary_key PRIMARY KEY (title_id,name_id),
 CONSTRAINT Writers_title_id_foreign_key
 FOREIGN KEY (title_id) REFERENCES project.Titles(title_id) ON DELETE Cascade ON UPDATE Cascade,
 CONSTRAINT Writers_name_id_foreign_key
 FOREIGN KEY (name_id) REFERENCES project.Person_Names(name_id) ON DELETE Cascade ON UPDATE Cascade
 -- When the name_id is deleted/updated from the Person_Names table, we want the same thing to happen in this table as well.
);



CREATE TABLE project.Principals (
 title_id      VARCHAR(255) NOT NULL,
 ordering      INT NOT NULL,
 name_id       VARCHAR(255) NOT NULL,
 job_category  VARCHAR(255),
 job           TEXT,
 CONSTRAINT Principals_primary_key PRIMARY KEY(title_id,ordering),
 CONSTRAINT Principals_name_id_foreign_key
 FOREIGN KEY (name_id) REFERENCES project.Person_Names(name_id) ON DELETE Cascade ON UPDATE Cascade,
 -- When the name_id is deleted/updated from the Person_Names table, we want the same thing to happen in this table as well.
 CONSTRAINT Principals_title_id_foreign_key
 FOREIGN KEY (title_id) REFERENCES project.Titles(title_id) ON DELETE Cascade ON UPDATE Cascade
 -- We want the title_id to be deleted/updated when the parent(Titles) is deleted/updated
);
```

11. BCNF Provement :

Titles:

title_id (PK)-> title_type, primary_title, original_title, is_adult, start_year, end_year, runtime_minutes

Title_ratings:

title_id (PK, FK) -> number_of_votes, average_rating

Episode_belongs_to:

episode_title_id (PK, FK) -> parent_tv_show_title_id (FK), season_number, episode_number

Title_genre:

title_id (PK, FK), genre_id (PK) -> title_id, genre_id

Had_role:

name_id (PK), title_id (PK), role (PK) -> name_id, title_id, role

Principals:

title_id (PK, FK), ordering (PK) -> name_id (FK), job_category, job

Directors:

title_id (PK, FK), name_id (PK, FK) -> title_id, name_id

Person_names:

name_id (PK) -> full_name, birth_year, death_year

Writers:

title_id (PK, FK), name_id (PK, FK) -> title_id, name_id

Name_worked_as:

Name_id (PK, FK), profession (PK) -> name_id, profession

1NF

As all the attributes are single valued attributes, the schema is in first normal form.

2NF

All non prime attributes fully depend on the candidate key. No subset of candidate key can determine any non prime attribute. Hence it is in second normal form.

3NF

All non prime attributes are only determined by the prime attributes or candidate key. Hence the schema satisfies the third normal form.

BCNF (Boyce-Codd normal form)

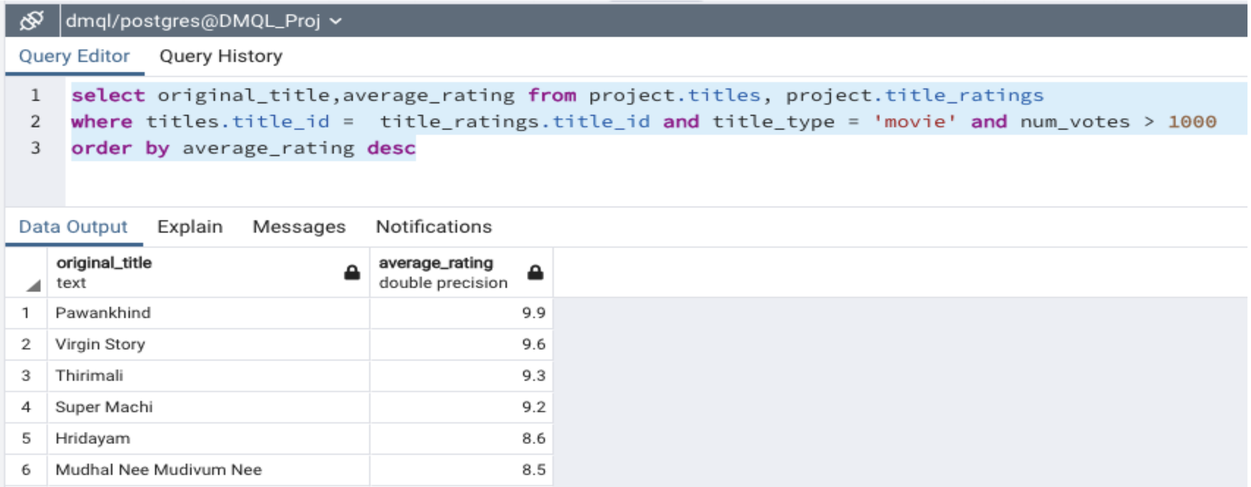
For every above functional dependency X->Y, X is always a superkey. Hence it is proved that the schema is in BCNF.

12. SQL Execution :

UseCase1: Find best movies for year 2022 based on ratings given by users.

Query : select original_title,average_rating from project.titles, project.title_ratings where titles.title_id = title_ratings.title_id and title_type = 'movie' and num_votes > 1000

QueryResults:



```
select original_title,average_rating from project.titles, project.title_ratings
where titles.title_id =  title_ratings.title_id and title_type = 'movie' and num_votes > 1000
order by average_rating desc
```

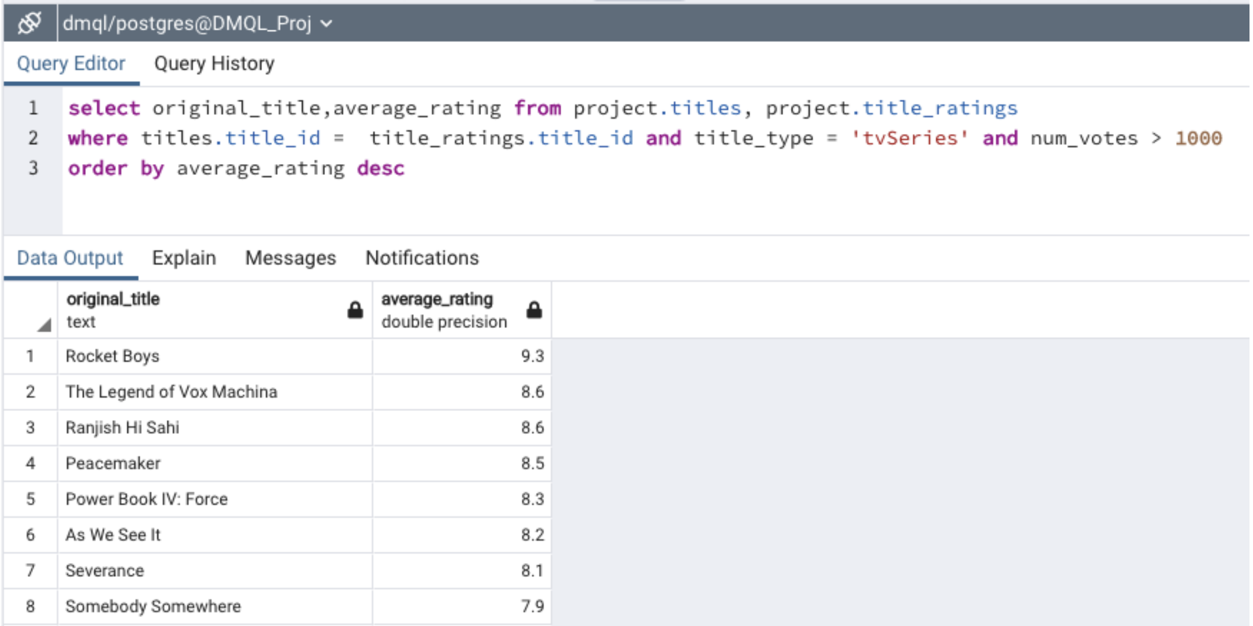
	original_title text	average_rating double precision
1	Pawankhind	9.9
2	Virgin Story	9.6
3	Thirimali	9.3
4	Super Machi	9.2
5	Hridayam	8.6
6	Mudhal Nee Mudivum Nee	8.5

UseCase 2 : Find best TV Series rankings of all time

Query : select original_title,average_rating from project.titles, project.title_ratings

where titles.title_id = title_ratings.title_id and title_type = 'tvSeries' and num_votes > 1000
order by average_rating desc

QueryResult :



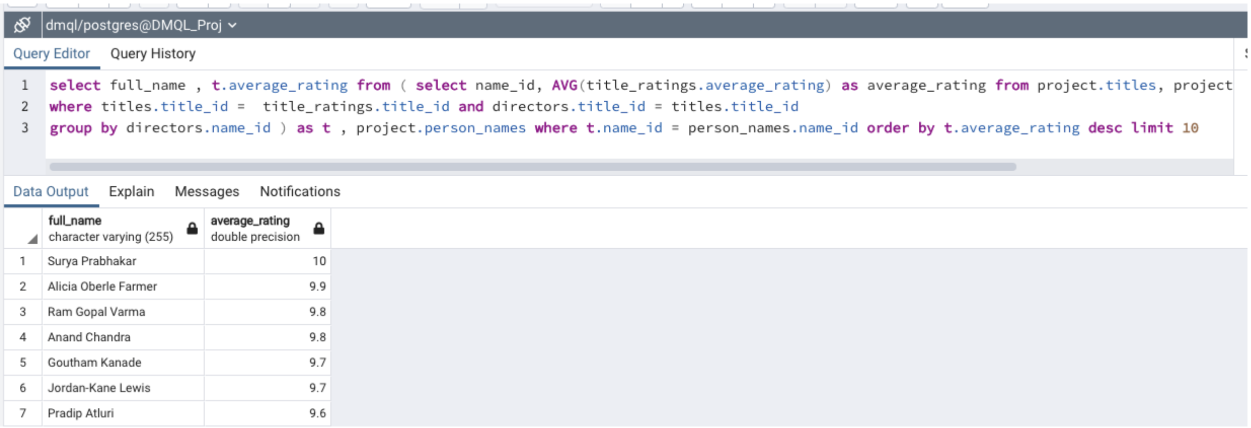
```sql
select original_title,average_rating from project.titles, project.title_ratings
where titles.title_id =  title_ratings.title_id and title_type = 'tvSeries' and num_votes > 1000
order by average_rating desc
```

	original_title text	average_rating double precision
1	Rocket Boys	9.3
2	The Legend of Vox Machina	8.6
3	Ranjish Hi Sahi	8.6
4	Peacemaker	8.5
5	Power Book IV: Force	8.3
6	As We See It	8.2
7	Severance	8.1
8	Somebody Somewhere	7.9

UseCase 3 : Find the highest rated top 10 directors of all time

Query : select full_name , t.average_rating from (select name_id, AVG(title_ratings.average_rating) as average_rating from project.titles, project.title_ratings,project.directors
where titles.title_id = title_ratings.title_id and directors.title_id = titles.title_id
group by directors.name_id) as t , project.person_names where t.name_id = person_names.name_id order by t.average_rating desc limit 10

Query Result :



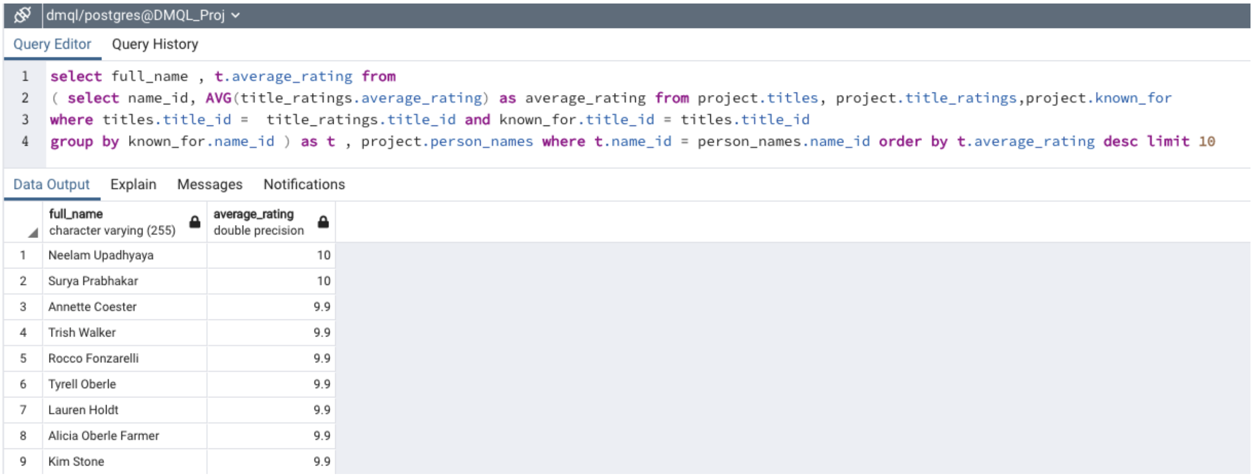
```sql
select full_name , t.average_rating from ( select name_id, AVG(title_ratings.average_rating) as average_rating from project.titles, project
where titles.title_id =  title_ratings.title_id and directors.title_id = titles.title_id
group by directors.name_id ) as t , project.person_names where t.name_id = person_names.name_id order by t.average_rating desc limit 10
```

	full_name character varying (255)	average_rating double precision
1	Surya Prabhakar	10
2	Alicia Oberle Farmer	9.9
3	Ram Gopal Varma	9.8
4	Anand Chandra	9.8
5	Goutham Kanade	9.7
6	Jordan-Kane Lewis	9.7
7	Pradip Atluri	9.6

UseCase 4 : Name of the actor, acted in the movie with highest rating in all regions/particular region.

Query : select full_name , t.average_rating from
(select name_id, AVG(title_ratings.average_rating) as average_rating from project.titles, project.title_ratings,project.known_for
where titles.title_id = title_ratings.title_id and known_for.title_id = titles.title_id
group by known_for.name_id) as t , project.person_names where t.name_id = person_names.name_id order by t.average_rating desc limit 10

QueryResult :



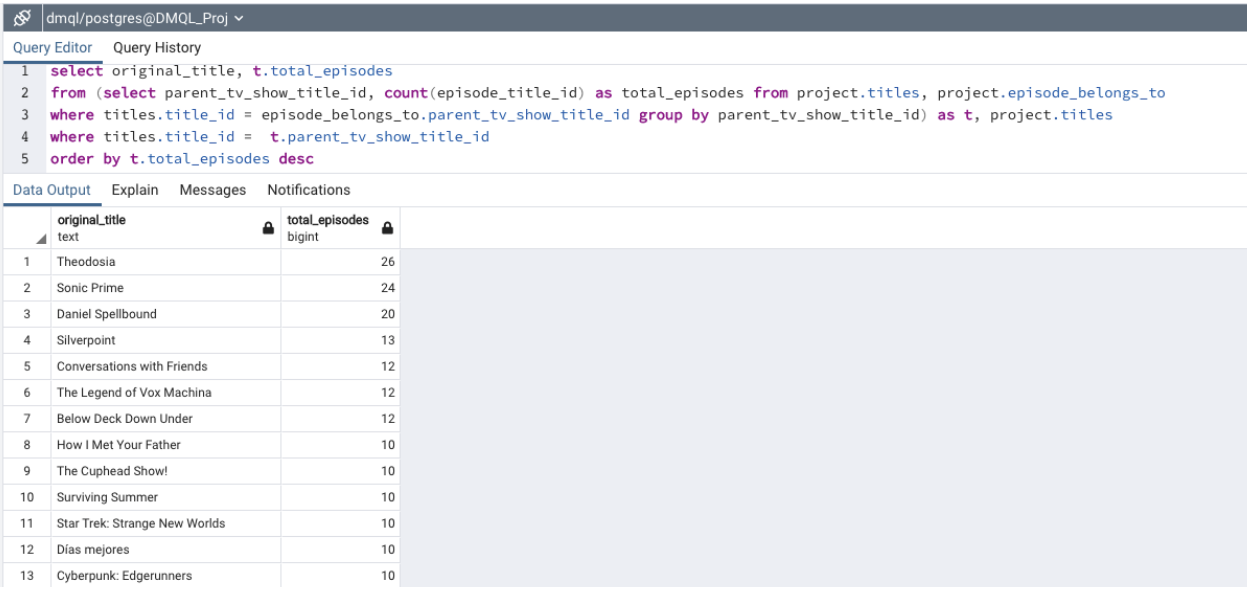
```
select full_name , t.average_rating from
( select name_id, AVG(title_ratings.average_rating) as average_rating from project.titles, project.title_ratings,project.known_for
where titles.title_id =  title_ratings.title_id and known_for.title_id = titles.title_id
group by known_for.name_id ) as t , project.person_names where t.name_id = person_names.name_id order by t.average_rating desc limit 10
```

	full_name character varying (255)	average_rating double precision
1	Neelam Upadhyaya	10
2	Surya Prabhakar	10
3	Annette Coester	9.9
4	Trish Walker	9.9
5	Rocco Fonzarelli	9.9
6	Tyrell Oberle	9.9
7	Lauren Holdt	9.9
8	Alicia Oberle Farmer	9.9
9	Kim Stone	9.9

UseCase 5 : Series with most no.of episodes (no.of episodes -in each season * no.of seasons)   .

Query : select original_title, t.total_episodes
from (select parent_tv_show_title_id, count(episode_title_id) as total_episodes from project.titles, project.episode_belongs_to
where titles.title_id = episode_belongs_to.parent_tv_show_title_id group by parent_tv_show_title_id) as t, project.titles
where titles.title_id = t.parent_tv_show_title_id
order by t.total_episodes desc

Query result :



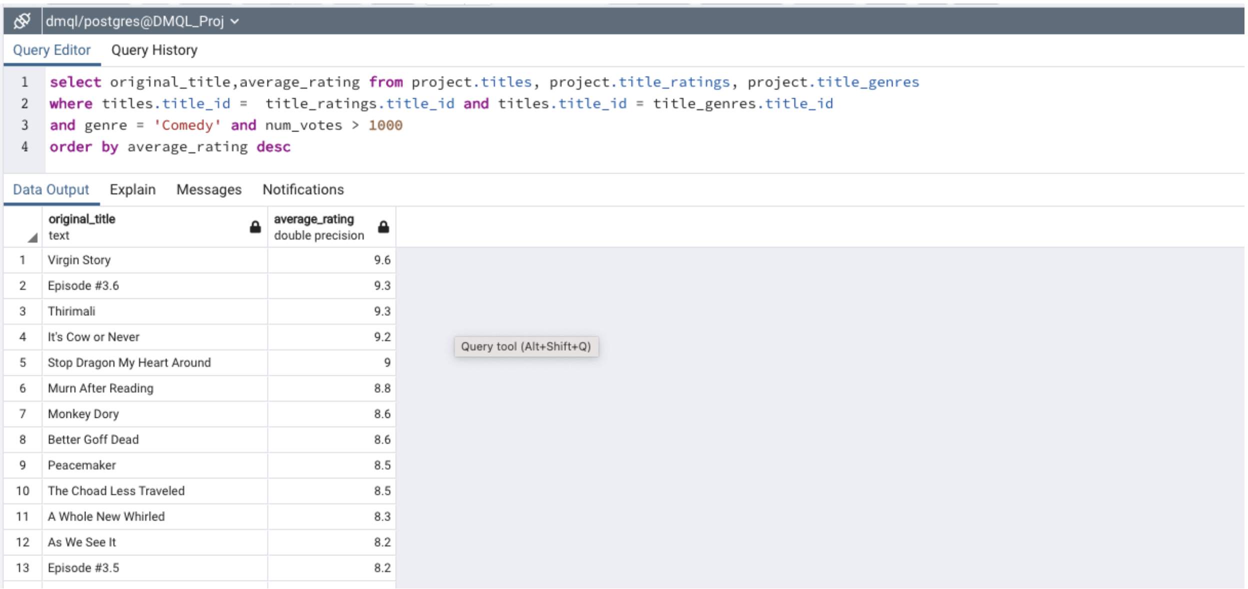
```sql
select original_title, t.total_episodes
from (select parent_tv_show_title_id, count(episode_title_id) as total_episodes from project.titles, project.episode_belongs_to
where titles.title_id = episode_belongs_to.parent_tv_show_title_id group by parent_tv_show_title_id) as t, project.titles
where titles.title_id =  t.parent_tv_show_title_id
order by t.total_episodes desc
```

	original_title text	total_episodes bigint
1	Theodosia	26
2	Sonic Prime	24
3	Daniel Spellbound	20
4	Silverpoint	13
5	Conversations with Friends	12
6	The Legend of Vox Machina	12
7	Below Deck Down Under	12
8	How I Met Your Father	10
9	The Cuphead Show!	10
10	Surviving Summer	10
11	Star Trek: Strange New Worlds	10
12	Días mejores	10
13	Cyberpunk: Edgerunners	10

UseCase 6 : Best Comedy movies with votes > 1000.

Query : select original_title,average_rating from project.titles, project.title_ratings, project.title_genres
where titles.title_id = title_ratings.title_id and titles.title_id = title_genres.title_id
and genre = 'Comedy' and num_votes > 1000
order by average_rating desc

Query Result:



```sql
select original_title,average_rating from project.titles, project.title_ratings, project.title_genres
where titles.title_id =  title_ratings.title_id and titles.title_id = title_genres.title_id
and genre = 'Comedy' and num_votes > 1000
order by average_rating desc
```

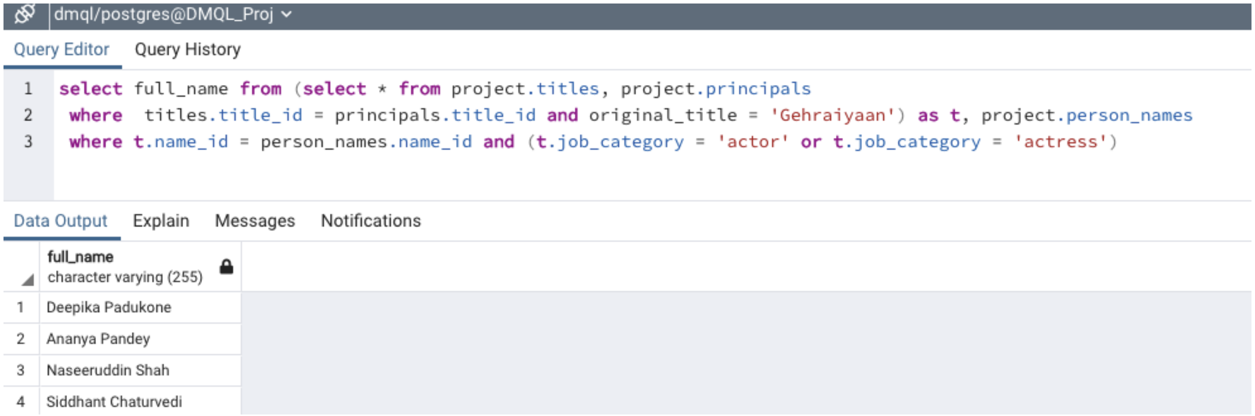
	original_title text	average_rating double precision
1	Virgin Story	9.6
2	Episode #3.6	9.3
3	Thirimali	9.3
4	It's Cow or Never	9.2
5	Stop Dragon My Heart Around	9
6	Murn After Reading	8.8
7	Monkey Dory	8.6
8	Better Goff Dead	8.6
9	Peacemaker	8.5
10	The Choad Less Traveled	8.5
11	A Whole New Whirled	8.3
12	As We See It	8.2
13	Episode #3.5	8.2

UseCase 7 : Get all cast of a single movie

Query : select full_name from (select * from project.titles, project.principals

where titles.title_id = principals.title_id and original_title = 'Gehraiyaan') as t,
project.person_names
where t.name_id = person_names.name_id and (t.job_category = 'actor' or
t.job_category = 'actress')

Query Result :



```
select full_name from (select * from project.titles, project.principals
 where  titles.title_id = principals.title_id and original_title = 'Gehraiyaan') as t, project.person_names
 where t.name_id = person_names.name_id and (t.job_category = 'actor' or t.job_category = 'actress')
```

	full_name character varying (255)
1	Deepika Padukone
2	Ananya Pandey
3	Naseeruddin Shah
4	Siddhant Chaturvedi

UseCase 8 : Trigger for logging new insertion data in Titles table (Logs are stored in a new table “history”.

Query : CREATE TABLE project.history(title_id varchar NOT NULL, entry_date VARCHAR(100) NOT NULL);

CREATE or REPLACE FUNCTION historyFunc() RETURNS TRIGGER AS
$examp_table$
BEGIN
INSERT INTO project.history(title_id, entry_date) VALUES (new.title_id,
current_timestamp);
RETURN new;
END;
$examp_table$ LANGUAGE plpgsql;

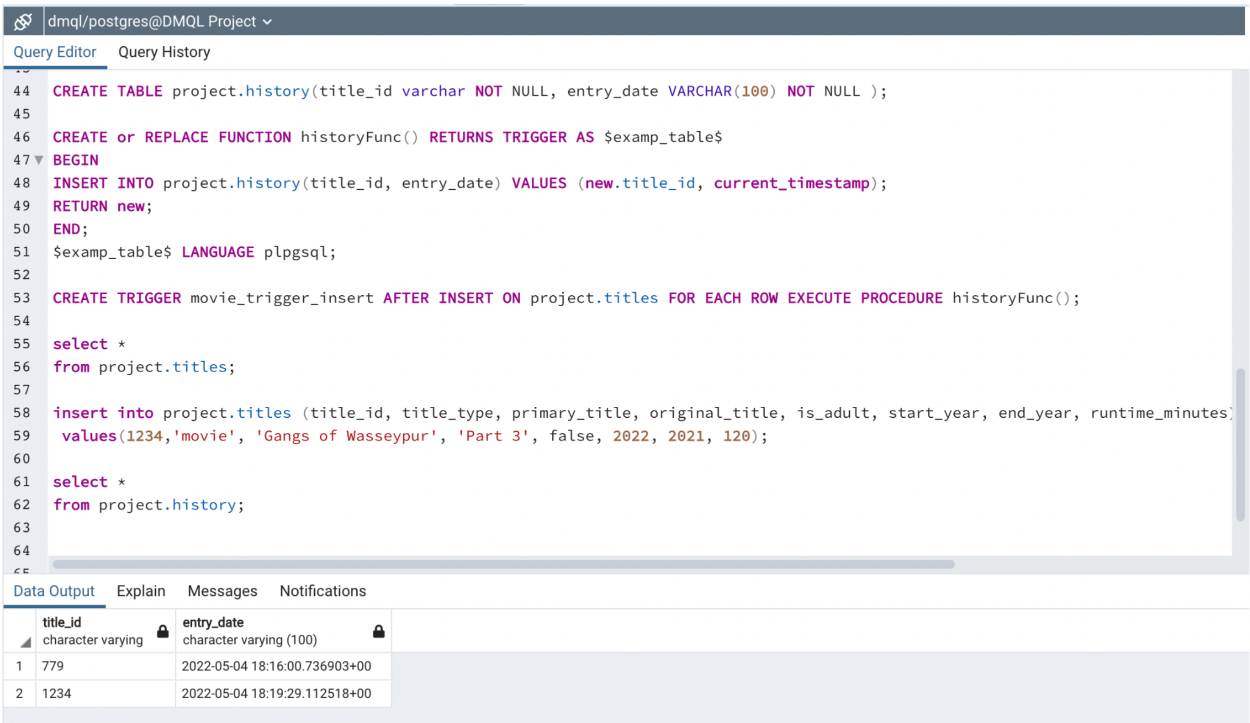
CREATE TRIGGER movie_trigger_insert AFTER INSERT ON project.titles FOR EACH ROW EXECUTE PROCEDURE historyFunc();

select *
from project.titles;

insert into project.titles (title_id, title_type, primary_title, original_title, is_adult,
start_year, end_year, runtime_minutes)
values(1234,'movie', 'Gangs of Wasseypur', 'Part 3', false, 2022, 2021, 120);

select *
from project.history;

Query Result :



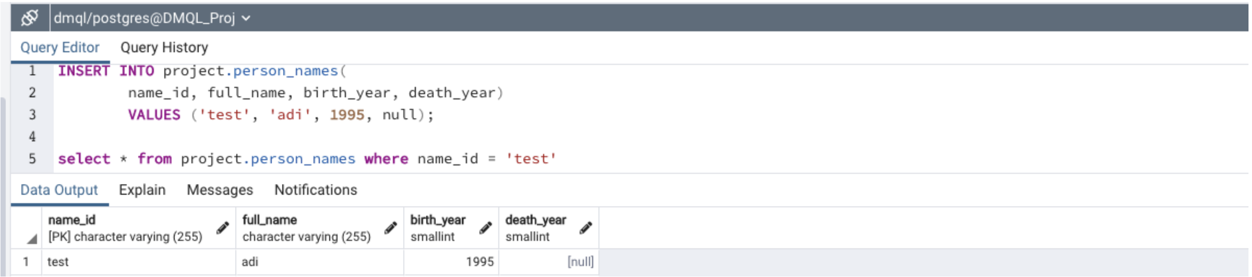
```
CREATE TABLE project.history(title_id varchar NOT NULL, entry_date VARCHAR(100) NOT NULL );

CREATE or REPLACE FUNCTION historyFunc() RETURNS TRIGGER AS $examp_table$
BEGIN
INSERT INTO project.history(title_id, entry_date) VALUES (new.title_id, current_timestamp);
RETURN new;
END;
$examp_table$ LANGUAGE plpgsql;

CREATE TRIGGER movie_trigger_insert AFTER INSERT ON project.titles FOR EACH ROW EXECUTE PROCEDURE historyFunc();

select *
from project.titles;

insert into project.titles (title_id, title_type, primary_title, original_title, is_adult, start_year, end_year, runtime_minutes)
 values(1234,'movie', 'Gangs of Wasseypur', 'Part 3', false, 2022, 2021, 120);

select *
from project.history;
```

	title_id character varying	entry_date character varying (100)
1	779	2022-05-04 18:16:00.736903+00
2	1234	2022-05-04 18:19:29.112518+00

UseCase 9 : Inserting a new person into the person_names table.

Query : INSERT INTO project.person_names(
name_id, full_name, birth_year, death_year)
VALUES ('test', 'adi', 1995, null);

Query Result :



```
INSERT INTO project.person_names(
        name_id, full_name, birth_year, death_year)
        VALUES ('test', 'adi', 1995, null);

select * from project.person_names where name_id = 'test'
```

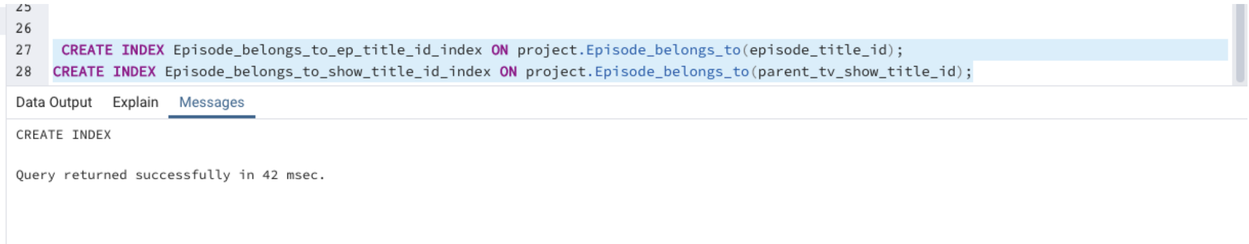
	name_id [PK] character varying (255)	full_name character varying (255)	birth_year smallint	death_year smallint
1	test	adi	1995	[null]

UseCase 10 : Indexing.

Query :
CREATE INDEX Episode_belongs_to_ep_title_id_index ON project.Episode_belongs_to(episode_title_id);

CREATE INDEX Episode_belongs_to_show_title_id_index ON project.Episode_belongs_to(parent_tv_show_title_id);

Query Result :



```
 CREATE INDEX Episode_belongs_to_ep_title_id_index ON project.Episode_belongs_to(episode_title_id);
CREATE INDEX Episode_belongs_to_show_title_id_index ON project.Episode_belongs_to(parent_tv_show_title_id);
```

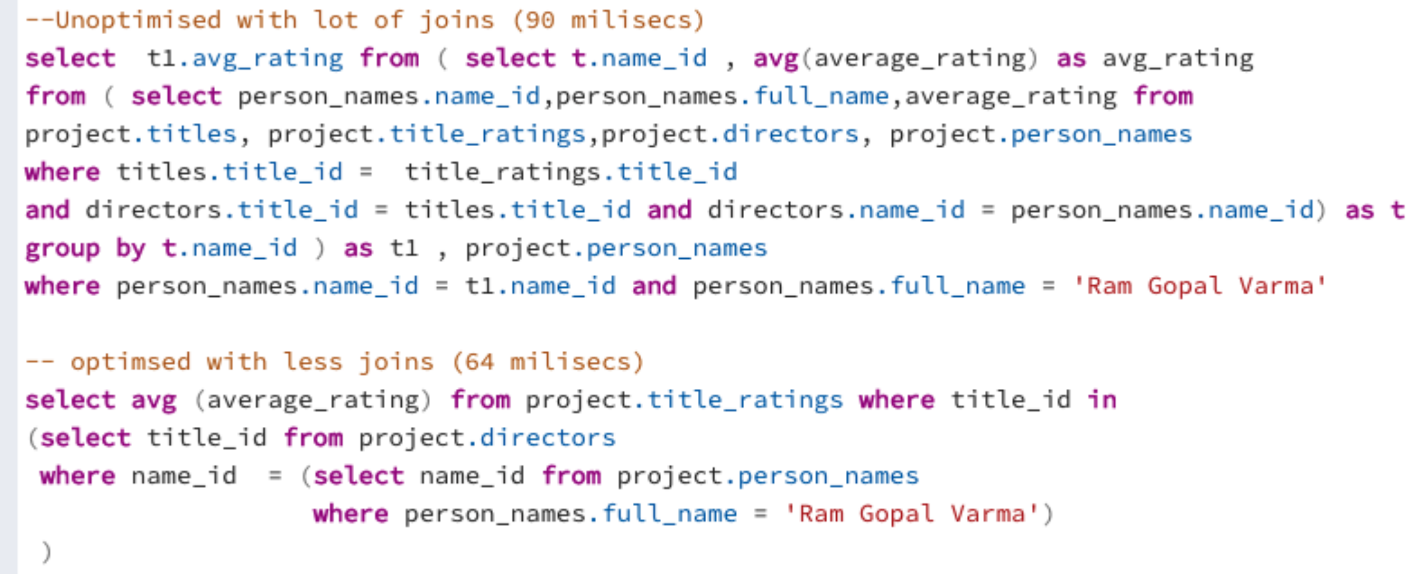
```
CREATE INDEX

Query returned successfully in 42 msec.
```

UseCase 11 : Deletion and Update.

Query :
delete from project.person_names where name_id = 'test';
update project.person_name set full_name = 'aditya' where name_id = 'test';

13. Query execution analysis :

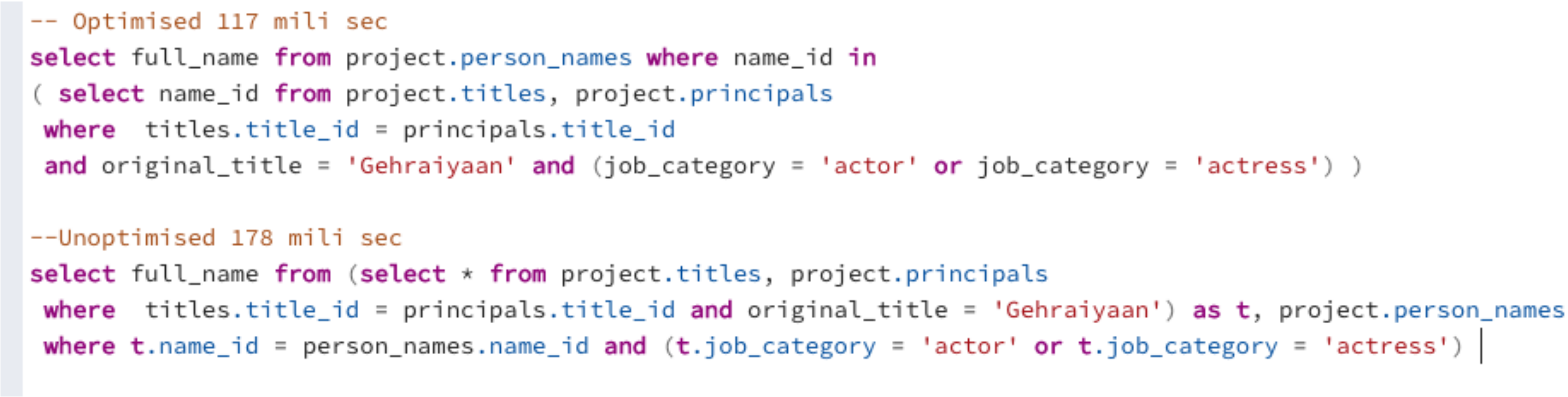
1. **Find the average rating for titles by Ram Gopal Varma**



```
--Unoptimised with lot of joins (90 milisecs)
select  t1.avg_rating from ( select t.name_id , avg(average_rating) as avg_rating
from ( select person_names.name_id,person_names.full_name,average_rating from
project.titles, project.title_ratings,project.directors, project.person_names
where titles.title_id =  title_ratings.title_id
and directors.title_id = titles.title_id and directors.name_id = person_names.name_id) as t
group by t.name_id ) as t1 , project.person_names
where person_names.name_id = t1.name_id and person_names.full_name = 'Ram Gopal Varma'

-- optimsed with less joins (64 milisecs)
select avg (average_rating) from project.title_ratings where title_id in
(select title_id from project.directors
 where name_id  = (select name_id from project.person_names
                   where person_names.full_name = 'Ram Gopal Varma')
 )
```

Here the unoptimised query does joins first and then uses filtering. We should fliter our results first and then query.

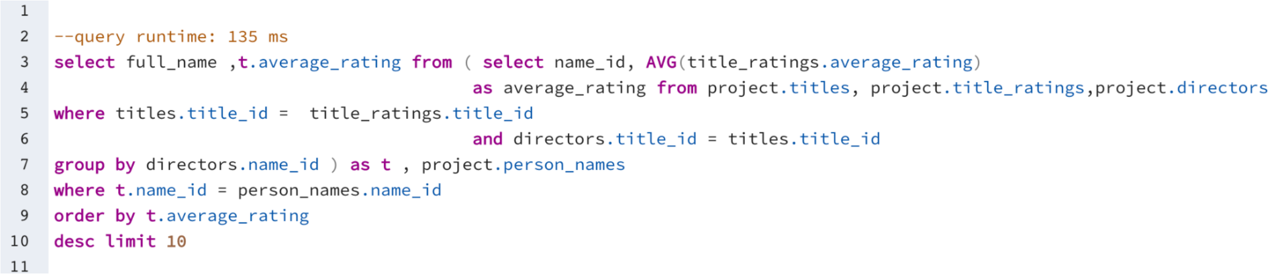
2. **Cast of Gehraiyaan movie**



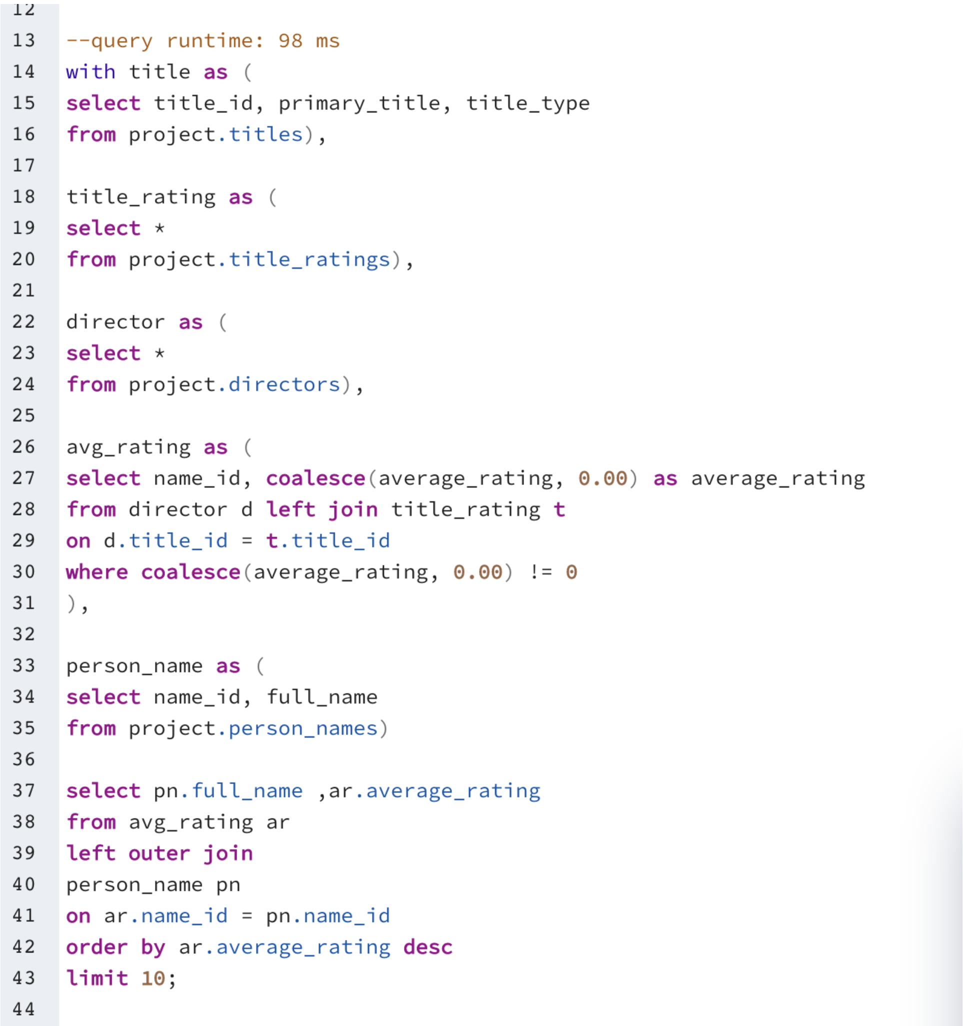
```
-- Optimised 117 mili sec
select full_name from project.person_names where name_id in
( select name_id from project.titles, project.principals
 where  titles.title_id = principals.title_id
 and original_title = 'Gehraiyaan' and (job_category = 'actor' or job_category = 'actress') )

--Unoptimised 178 mili sec
select full_name from (select * from project.titles, project.principals
 where  titles.title_id = principals.title_id and original_title = 'Gehraiyaan') as t, project.person_names
 where t.name_id = person_names.name_id and (t.job_category = 'actor' or t.job_category = 'actress')
```

Instead of projecting all the columns we can project only the column that we need, it will save some execution time.

3. Top10 directors of 2022 comparison



```
--query runtime: 135 ms
select full_name ,t.average_rating from ( select name_id, AVG(title_ratings.average_rating)
                                          as average_rating from project.titles, project.title_ratings,project.directors
where titles.title_id =  title_ratings.title_id
                                          and directors.title_id = titles.title_id
group by directors.name_id ) as t , project.person_names
where t.name_id = person_names.name_id
order by t.average_rating
desc limit 10
```



```
--query runtime: 98 ms
with title as (
select title_id, primary_title, title_type
from project.titles),

title_rating as (
select *
from project.title_ratings),

director as (
select *
from project.directors),

avg_rating as (
select name_id, coalesce(average_rating, 0.00) as average_rating
from director d left join title_rating t
on d.title_id = t.title_id
where coalesce(average_rating, 0.00) != 0
),

person_name as (
select name_id, full_name
from project.person_names)

select pn.full_name ,ar.average_rating
from avg_rating ar
left outer join
person_name pn
on ar.name_id = pn.name_id
order by ar.average_rating desc
limit 10;
```

CTE query is faster because the projection is done based on only on the columns required and not on unnecessary columns unlike first query. As the number of records is more, this makes a lot of difference in query execution time.