CSE 460/560 - DMQL Project Final Report

I. 1. Project details

Name of your project, your team, and all team members, everyone's UB id(not the UB number);

Project Name: Preview before Review Movies
Group members:
Ryan O'Sullivan - rposulli
Sakshi Singhal - ss666
Ismail Ajaz - ismailaj

II. 2. Problem Statement [5 points]:

Form a title and problem statement that clearly state the problem and questions you are trying to answer. Why do you need a database instead of an Excel file?

The title of the project is "Preview before Review Movies" and the problem statement is "Lack of prior knowledge about a movie exposes viewers with unexpected and even disappointing entertainment options." The questions that we can answer by using this dataset would be "I am feeling low. Which is the best movie to watch? I don't wanna waste my time like I did for the XYZ movie. It's been a long time since I haven't watched an action movie, let's find something." Because of its scalability, concurrent access, data integrity, security, and capacity for managing data relationships, databases will be a much better option than Excel files. They can handle massive volumes of data, preserve data consistency, and ensure data security because they have advanced querying, backup, and recovery capabilities. Additionally, they would reduce the redundancy that we would otherwise face if we combined several entity sets into a single table. Due to the limitations of Excel files in these areas, databases are a more reliable option for complicated data management requirements.

A. Discuss the background of the problem leading to your objectives. Why is it a significant problem?

At the moment, people rely heavily on online reviews and suggestions, which frequently leads to surprising and potentially disappointing movie-watching experiences. Moviegoing consumers want to make sure that they are watching something that is worth their time and money in the theater, and without resources like movie databases, it will be much harder for them to make an informed decision about which movies to attend. This is a significant problem because the absence of resources like ours will worsen the moviegoer's average experience, and it may convince them to visit the theater less often. This will result in less revenue for the theater and less enjoyment from movie fans.

B. Explain the potential of your project to contribute to your problem domain. Discuss why this contribution is crucial.

The project will provide moviegoers with a way of attaining information much more efficiently (as opposed to endless scrolling through reviews/suggestions on a variety of sites) and with more quality. Scrolling through random reviews means the user will have to ingest many unbiased and unqualified opinions of many different people who may expect and desire different things in a movie. Our contribution can resolve these issues because all of our data will be in one place (the database) and the data itself is significantly more unbiased because the data consists of reviews and ratings that come from regular users and actual critics, whose opinions are much more valuable than some random person who gave a bad rating because they were upset that they did not get to see a particular scene that they were expecting.

III. Target user [5 points] :

Who will use your database? Who will administer the database? You are encouraged to give a real-life scenario;

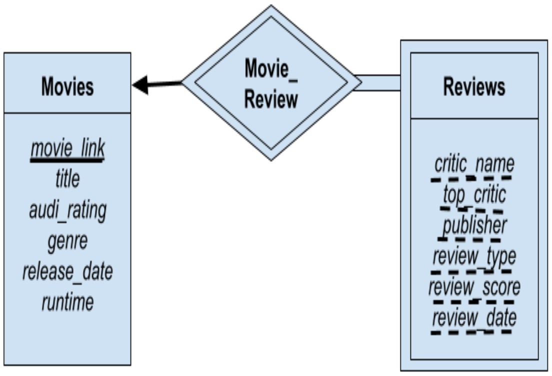
Anyone who wants to get a recommendation on a movie will be using our database. Suppose you are not someone who regularly sees movies, and you buy tickets to the Barbie movie without ever seeing a single review, just because of the branding of the Title. Imagine if the movie completely flopped because it was another corporate cash grab on a recognizable name. Imagine how disappointed you'd feel walking out of the theater. Wouldn't it have been so helpful to see the negative reviews about the movie? That way you could have made a more informed decision, like buying tickets to the Oppenheimer movie which was well-received. The people administering our database would be the popular theater businesses like Regal theaters, or alternatively, this could be presented by streaming services like Netflix or Hulu which have similar programs. They would get increased engagement because viewers would feel comfortable if they knew they were getting their money's worth before they subscribe, or continue their subscription to the service.

IV. E/R diagram [10 points]:

Draw an E/R diagram for your database and briefly describe the relationships between different tables. (Do not draw the figure by hand, you may use any tools to design or generate your E/R diagram

Movies is a strong entity set wherein each tuple corresponds to one movie, and other relevant information about that particular movie. It is identified by the primary key movie_link which is guaranteed to be unique for each movie on the site. Reviews is a weak entity set which depends on identifying set

Movies vía Movie_Review relationship, because each tuple is a movie review which logically depends on the existence of the movie being reviewed. This is also why all attributes of Reviews must be the partial key. One movie can receive many reviews, but each review can only be for one movie, so there is a one-to-many relationship. Additionally, every review must participate in Movie_Review because it always has a parent movie, but not every movie must have a review, hence the partial participation arrow.



V. Tasks (3 and 4) should be completed [10 Points]:

(Task 3) Acquire the large "production" dataset, either by downloading it from a real data source or generating it using a program. Make sure the dataset fits your schema. You might need to write programs/scripts to transform them into a suitable form for loading into a database for real datasets. For program-generated datasets, make sure they contain interesting enough "links" across rows of different tables to show the results of different Advanced SQL queries learned in class.

Done. See figures 1 and 2 at near the bottom of this report.

(Task 4) You are required to make sure all of your relations are in Boyce- Codd Normal Form. Provide a list of dependencies for each relation. Decompose them if the tables are not in BCNF. If you decide to keep it in 3NF instead of BCNF, justify the decision for a particular relation. Your report for this milestone should contain a separate section with the details of the transformation from the initial schema to the final schema where the relations are in BCNF

The only noteworthy FDs are: $movie\_link \rightarrow movie\_link, title, audi\_rating, genre, release\_date, runtime,$

and (movie_link, critic_name, top_critic, publisher, review_type, review_score, review_date) → (movie_link, critic_name, top_critic, publisher, review_type, review_score, review_date).

For the Reviews table, because all attributes are needed to uniquely identify a tuple, there are no nontrivial functional dependencies here. Only the 2nd FD mentioned, the trivial one, holds. Thus, Reviews are in BCNF.

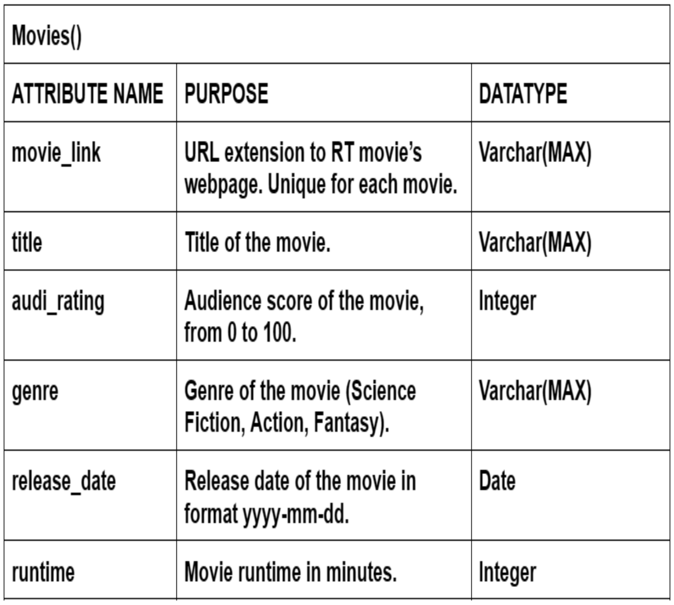
For the Movies table, the only related FD is that movie_link -¿ movie_link, title, audi_rating, genre, release_date, runtime. movie_link is the primary key/superkey, and none of the remaining attributes are in the LHS of an FD, so Movies is also in BCNF

A. 4.1 Define a list of relations and their attributes.

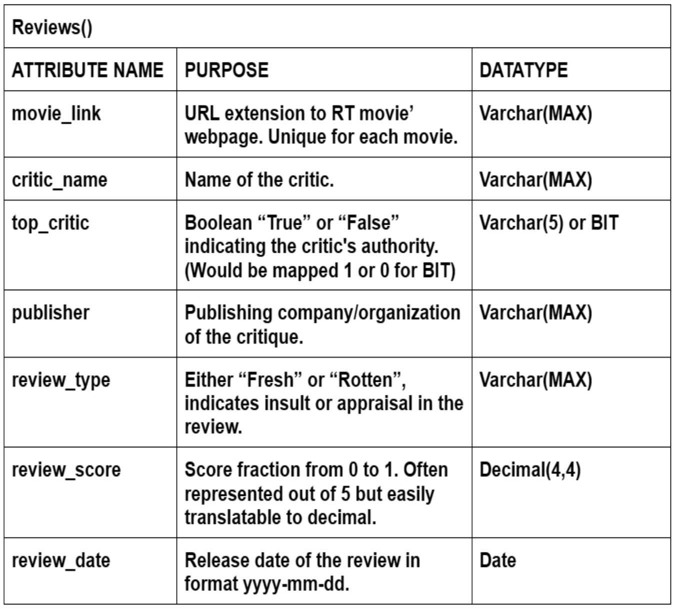
The relation schemas for each relation is as follows: Movies(movie_link, title, audi_rating, genre, release_date, runtime) Reviews(movie_link, critic_name, top_critic, publisher, review_type, review_score, review_date) Indicate the primary key and foreign keys (if any) for each relation. Justify your choice; For Movies, the primary key is movie_link because each movie is hosted at a different URL on the site, and thus is guaranteed to be a unique identifier for each movie. For Reviews, a weak entity set, there is no single attribute primary key, or even an obvious single-attribute partial key. This is because reviews can have the same critic name, review the same movie, suggest the same score on the same date, etc. Because there are no true identifiers in the weak entity set, we must use ALL attributes as a primary key for reviews. This is common in weak entity sets without an obvious partial key. (movie_link, critic_name, top_critic, publisher, review_type, review_score, review_date)

B. 4.2:

Write a detailed description of each attribute (for each table), its purpose, and datatype;



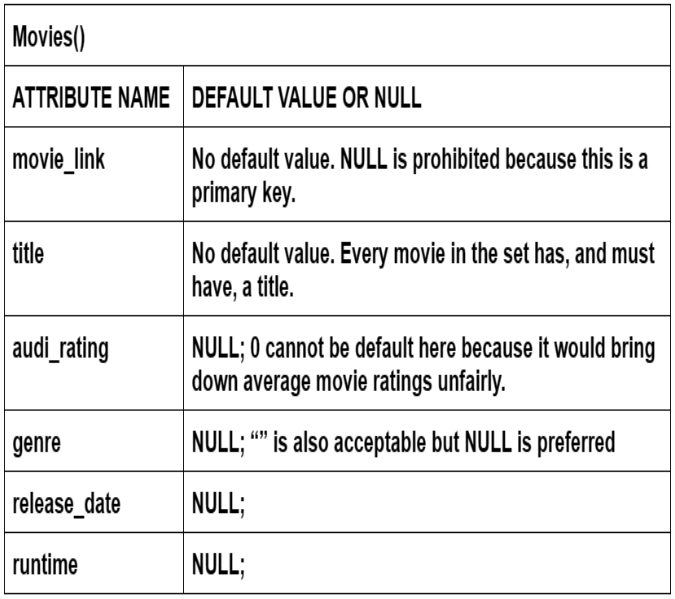
Movies()		
ATTRIBUTE NAME	PURPOSE	DATATYPE
movie_link	URL extension to RT movie's webpage. Unique for each movie.	Varchar(MAX)
title	Title of the movie.	Varchar(MAX)
audi_rating	Audience score of the movie, from 0 to 100.	Integer
genre	Genre of the movie (Science Fiction, Action, Fantasy).	Varchar(MAX)
release_date	Release date of the movie in format yyyy-mm-dd.	Date
runtime	Movie runtime in minutes.	Integer



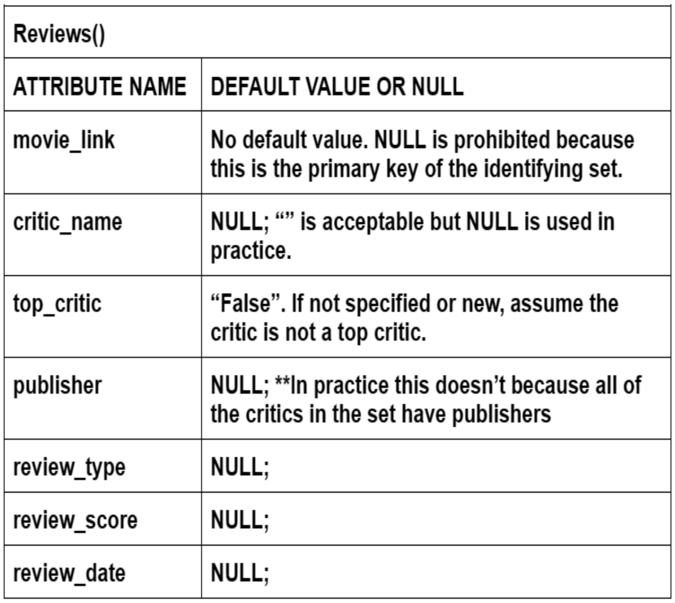
Reviews()		
ATTRIBUTE NAME	PURPOSE	DATATYPE
movie_link	URL extension to RT movie' webpage. Unique for each movie.	Varchar(MAX)
critic_name	Name of the critic.	Varchar(MAX)
top_critic	Boolean "True" or "False" indicating the critic's authority. (Would be mapped 1 or 0 for BIT)	Varchar(5) or BIT
publisher	Publishing company/organization of the critique.	Varchar(MAX)
review_type	Either "Fresh" or "Rotten", indicates insult or appraisal in the review.	Varchar(MAX)
review_score	Score fraction from 0 to 1. Often represented out of 5 but easily translatable to decimal.	Decimal(4,4)
review_date	Release date of the review in format yyyy-mm-dd.	Date

C. 4.3

Indicate each attribute's default value (if any) or if the attribute can be set to 'null';



Movies()	
ATTRIBUTE NAME	DEFAULT VALUE OR NULL
movie_link	No default value. NULL is prohibited because this is a primary key.
title	No default value. Every movie in the set has, and must have, a title.
audi_rating	NULL; 0 cannot be default here because it would bring down average movie ratings unfairly.
genre	NULL; "" is also acceptable but NULL is preferred
release_date	NULL;
runtime	NULL;



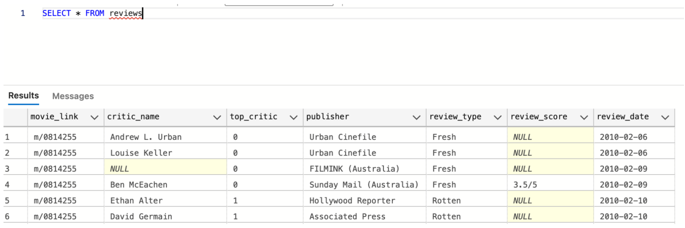
Reviews()	
ATTRIBUTE NAME	DEFAULT VALUE OR NULL
movie_link	No default value. NULL is prohibited because this is the primary key of the identifying set.
critic_name	NULL; "" is acceptable but NULL is used in practice.
top_critic	"False". If not specified or new, assume the critic is not a top critic.
publisher	NULL; **In practice this doesn't because all of the critics in the set have publishers
review_type	NULL;
review_score	NULL;
review_date	NULL;

D. 4.4

Explain the actions taken on any foreign key when the primary key (that the foreign key refer to) is deleted (e.g., no action, delete cascade, set null, set default)

When a particular movie_link is deleted from Movies, you would have to remove the entire tuple because that attribute is the primary key of Movies. Thus, the only occurrence of this would be if the entire tuple were deleted from the identifying entity set. Even then, we would take "NO ACTION". Just because a movie is deleted from Movies, doesn't mean that the movie review information is useless. We want to keep the correlated tuples in Reviews for further analysis. Additionally if the related movie_link value is inserted again later in Movies, we will still have the review data that goes along with it.

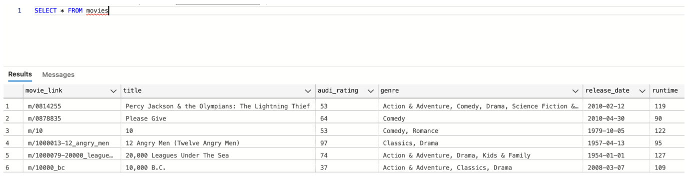
This concludes Phase 2 of the Final Report.



```
SELECT * FROM reviews
```

	movie_link	critic_name	top_critic	publisher	review_type	review_score	review_date
1	m/0814255	Andrew L. Urban	0	Urban Cinefile	Fresh	NULL	2010-02-06
2	m/0814255	Louise Keller	0	Urban Cinefile	Fresh	NULL	2010-02-06
3	m/0814255	NULL	0	FILMINK (Australia)	Fresh	NULL	2010-02-09
4	m/0814255	Ben McEachen	0	Sunday Mail (Australia)	Fresh	3.5/5	2010-02-09
5	m/0814255	Ethan Alter	1	Hollywood Reporter	Rotten	NULL	2010-02-10
6	m/0814255	David Germain	1	Associated Press	Rotten	NULL	2010-02-10

Fig. 1. Query sample from Reviews() table.



```
SELECT * FROM movies
```

	movie_link	title	audi_rating	genre	release_date	runtime
1	m/0814255	Percy Jackson & the Olympians: The Lightning Thief	53	Action & Adventure, Comedy, Drama, Science Fiction &…	2010-02-12	119
2	m/0878835	Please Give	64	Comedy	2010-04-30	90
3	m/10	10	53	Comedy, Romance	1979-10-05	122
4	m/1000013-12_angry_men	12 Angry Men (Twelve Angry Men)	97	Classics, Drama	1957-04-13	95
5	m/1000079-20000_league…	20,000 Leagues Under The Sea	74	Action & Adventure, Drama, Kids & Family	1954-01-01	127
6	m/10000_bc	10,000 B.C.	37	Action & Adventure, Classics, Drama	2008-03-07	109

Fig. 2. Query sample from Movies() table.

VI. Task 5: Problems and Indexing Concepts

"Do you specifically run into any problems while handling the larger dataset? Did you try to adopt some indexing concepts to resolve this? Briefly describe the questions you faced and how you solved them."

Even on our larger dataset, we did not run into any problems handling it using PostGreSQL. However, to demonstrate our understanding, we will discuss some common indexing methods used to handle large datasets. On ordered files like our Movies() table, which is ordered by movie_link, it would be implemented as a primary index, and NOT a clustered index. This is because records close in the index would also be close in data. Within this primary index, we can choose either a dense index model or a sparse index model. The main difference is that dense indices contain a record for every search-key value in the data file (in our case, an index record for every data record). However, sparse indexes have an advantage because they only include one index key per data block. Because of this and the fact that the data file is only sorted by the search key, we would prefer a sparse index for Movies(). The Reviews() table is a weak entity set and while it is ordered with respect to movie_link(), it is UNordered with respect to critic_name, which is a common search key for movie reviews. Because it's sorted on one of these dimensions, a secondary index would not be necessary. In this case, we would want to use a dense clustered index (variation of a primary index) in order to manage this large dataset. Dense clustered indexes are ideal for this case because the Reviews() table has many possible search keys, including movie_link, critic_name and publisher.

VII. Task 6: Testing the Database on SQL Queries

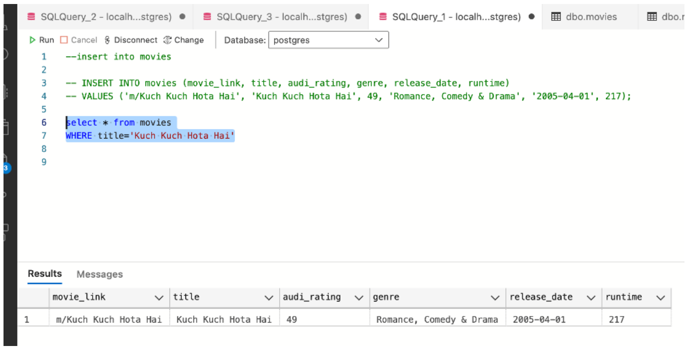
"Test your database with more than 10 SQL queries. You are supposed to design 1 or 2 queries for each inserting, deleting, and updating operation in your dataset."

For demonstration, we have performed over ten sample queries on the database, and we have displayed their results below. Figure 3 inserts a new movie "Kuch Kuch Hota Hai" into the Movies() table, Figure 4 updates the genre and runtime of the newly inserted tuple, and Figure 5 deletes this movie from the Movies() table entirely. Similarly, Figures 6, 7 and 8 are an insertion, update and deletion of a review for the movie "Kuch Kuch Hota Hai." In this case, the update changed the review_score and publisher name to 'Buffalo News'. In figures Figure 9 and 10, we performed the inner join on the Movies() and Reviews() table.

In figures 11 we first order the movies on their titles from Z-A (descending order). Then in figure 12, we ordered the movies in ascending order by genre.

We performed some similar queries for the Reviews() table. In figure 13, we ordered the reviews by the last name of their critics, from Z to A. Then in reference 14, we reordered the table alphabetically by release date.

Finally, we performed three grouping queries. In figure 15, we count the number of movies with the genre "Comedy" and find it to be 1263 total movies. Figure 16 is more general and shows us the movie count for every genre. Lastly, in figure 17, we group by movie_links to get the total number of reviews for each movie.

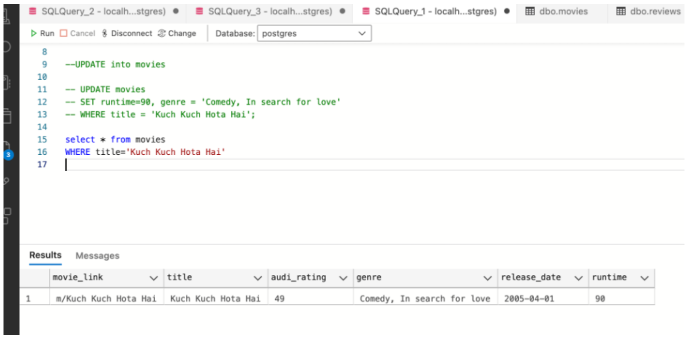


SQLQuery_2 - localh...stgres)
SQLQuery_3 - localh...stgres)
SQLQuery_1 - localh...stgres)
dbo.movies
dbo.r
Run
Cancel
Disconnect
Change
Database:
postgres
3
Results
Messages

```
--insert into movies

-- INSERT INTO movies (movie_link, title, audi_rating, genre, release_date, runtime)
-- VALUES ('m/Kuch Kuch Hota Hai', 'Kuch Kuch Hota Hai', 49, 'Romance, Comedy & Drama', '2005-04-01', 217);

select * from movies
WHERE title='Kuch Kuch Hota Hai'
```

	movie_link	title	audi_rating	genre	release_date	runtime
1	m/Kuch Kuch Hota Hai	Kuch Kuch Hota Hai	49	Romance, Comedy & Drama	2005-04-01	217

Fig. 3. Insertion of a tuple in the Movies() table.

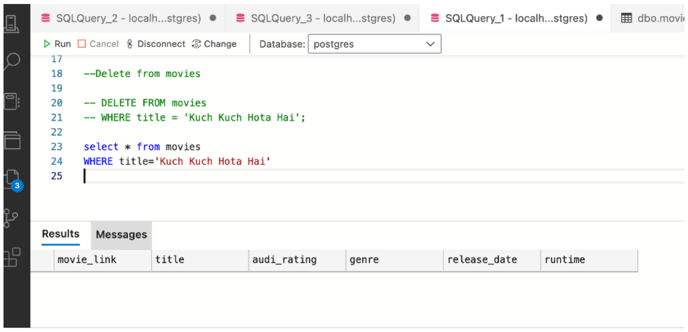


SQLQuery_2 - localh...stgres)
SQLQuery_3 - localh...stgres)
SQLQuery_1 - localh...stgres)
dbo.movies
dbo.reviews
Run
Cancel
Disconnect
Change
Database:
postgres
3
Results
Messages

```

--UPDATE into movies

-- UPDATE movies
-- SET runtime=90, genre = 'Comedy, In search for love'
-- WHERE title = 'Kuch Kuch Hota Hai';

select * from movies
WHERE title='Kuch Kuch Hota Hai'

```

	movie_link	title	audi_rating	genre	release_date	runtime
1	m/Kuch Kuch Hota Hai	Kuch Kuch Hota Hai	49	Comedy, In search for love	2005-04-01	90

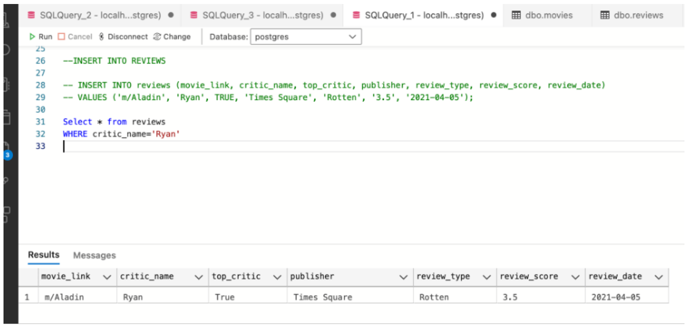
Fig. 4. Updating in the Movies() table



```
--Delete from movies

-- DELETE FROM movies
-- WHERE title = 'Kuch Kuch Hota Hai';

select * from movies
WHERE title='Kuch Kuch Hota Hai'
```

movie_link	title	audi_rating	genre	release_date	runtime

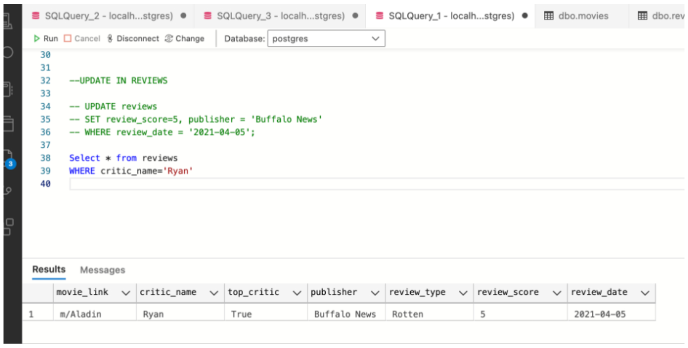
Fig. 5. Deletion of a tuple from the Movies() table



```
--INSERT INTO REVIEWS

-- INSERT INTO reviews (movie_link, critic_name, top_critic, publisher, review_type, review_score, review_date)
-- VALUES ('m/Aladin', 'Ryan', TRUE, 'Times Square', 'Rotten', '3.5', '2021-04-05');

Select * from reviews
WHERE critic_name='Ryan'
```

	movie_link	critic_name	top_critic	publisher	review_type	review_score	review_date
1	m/Aladin	Ryan	True	Times Square	Rotten	3.5	2021-04-05

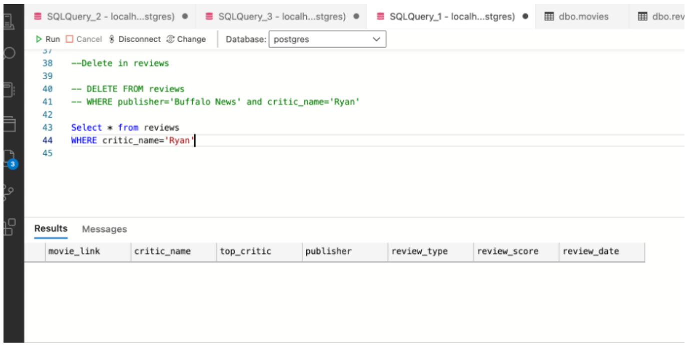
Fig. 6. Insertion of a tuple in the Reviews() table



```
--UPDATE IN REVIEWS

-- UPDATE reviews
-- SET review_score=5, publisher = 'Buffalo News'
-- WHERE review_date = '2021-04-05';

Select * from reviews
WHERE critic_name='Ryan'
```

	movie_link	critic_name	top_critic	publisher	review_type	review_score	review_date
1	m/Aladin	Ryan	True	Buffalo News	Rotten	5	2021-04-05

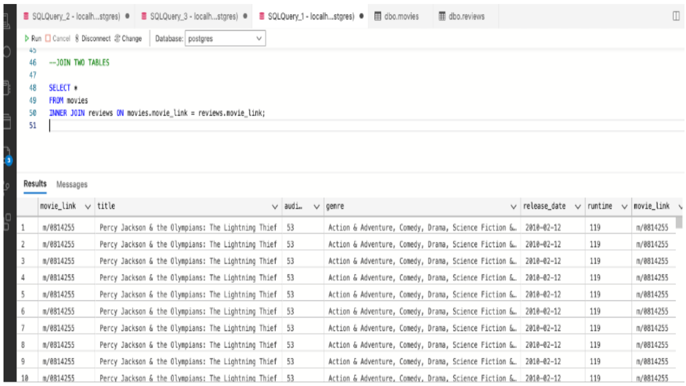
Fig. 7. Updating in Reviews() table



```
--Delete in reviews

-- DELETE FROM reviews
-- WHERE publisher='Buffalo News' and critic_name='Ryan'

Select * from reviews
WHERE critic_name='Ryan'
```

movie_link	critic_name	top_critic	publisher	review_type	review_score	review_date

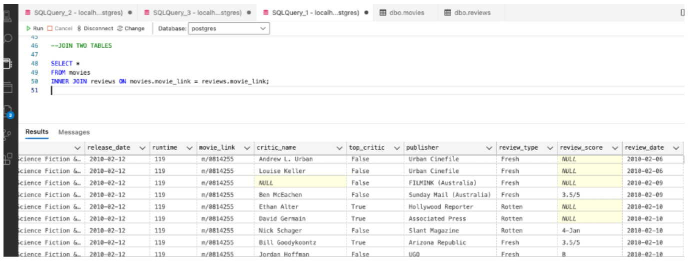
Fig. 8. Deletion of a tuple from the Review() table



```
--JOIN TWO TABLES

SELECT *
FROM movies
INNER JOIN reviews ON movies.movie_link = reviews.movie_link;
```

	movie_link	title	audi…	genre	release_date	runtime	movie_link
1	m/0814255	Percy Jackson & the Olympians: The Lightning Thief	53	Action & Adventure, Comedy, Drama, Science Fiction &…	2010-02-12	119	m/0814255
2	m/0814255	Percy Jackson & the Olympians: The Lightning Thief	53	Action & Adventure, Comedy, Drama, Science Fiction &…	2010-02-12	119	m/0814255
3	m/0814255	Percy Jackson & the Olympians: The Lightning Thief	53	Action & Adventure, Comedy, Drama, Science Fiction &…	2010-02-12	119	m/0814255
4	m/0814255	Percy Jackson & the Olympians: The Lightning Thief	53	Action & Adventure, Comedy, Drama, Science Fiction &…	2010-02-12	119	m/0814255
5	m/0814255	Percy Jackson & the Olympians: The Lightning Thief	53	Action & Adventure, Comedy, Drama, Science Fiction &…	2010-02-12	119	m/0814255
6	m/0814255	Percy Jackson & the Olympians: The Lightning Thief	53	Action & Adventure, Comedy, Drama, Science Fiction &…	2010-02-12	119	m/0814255
7	m/0814255	Percy Jackson & the Olympians: The Lightning Thief	53	Action & Adventure, Comedy, Drama, Science Fiction &…	2010-02-12	119	m/0814255
8	m/0814255	Percy Jackson & the Olympians: The Lightning Thief	53	Action & Adventure, Comedy, Drama, Science Fiction &…	2010-02-12	119	m/0814255
9	m/0814255	Percy Jackson & the Olympians: The Lightning Thief	53	Action & Adventure, Comedy, Drama, Science Fiction &…	2010-02-12	119	m/0814255
10	m/0814255	Percy Jackson & the Olympians: The Lightning Thief	53	Action & Adventure, Comedy, Drama, Science Fiction &…	2010-02-12	119	m/0814255

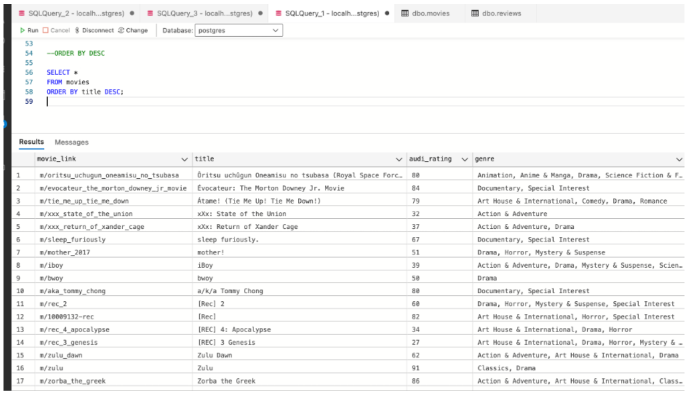
Fig. 9. Joining the Movies() and Reviews() tables.



```
--JOIN TWO TABLES

SELECT *
FROM movies
INNER JOIN reviews ON movies.movie_link = reviews.movie_link;
```

	release_date	runtime	movie_link	critic_name	top_critic	publisher	review_type	review_score	review_date
cience Fiction &…	2010-02-12	119	m/0814255	Andrew L. Urban	False	Urban Cinefile	Fresh	NULL	2010-02-06
cience Fiction &…	2010-02-12	119	m/0814255	Louise Keller	False	Urban Cinefile	Fresh	NULL	2010-02-06
cience Fiction &…	2010-02-12	119	m/0814255	NULL	False	FILMINK (Australia)	Fresh	NULL	2010-02-09
cience Fiction &…	2010-02-12	119	m/0814255	Ben McEachen	False	Sunday Mail (Australia)	Fresh	3.5/5	2010-02-09
cience Fiction &…	2010-02-12	119	m/0814255	Ethan Alter	True	Hollywood Reporter	Rotten	NULL	2010-02-10
cience Fiction &…	2010-02-12	119	m/0814255	David Germain	True	Associated Press	Rotten	NULL	2010-02-10
cience Fiction &…	2010-02-12	119	m/0814255	Nick Schager	False	Slant Magazine	Rotten	4-Jan	2010-02-10
cience Fiction &…	2010-02-12	119	m/0814255	Bill Goodykoontz	True	Arizona Republic	Fresh	3.5/5	2010-02-10
cience Fiction &…	2010-02-12	119	m/0814255	Jordan Hoffman	False	UGO	Fresh	B	2010-02-10

Fig. 10. Joining the Movies() and Reviews() tables, continued.



```
--ORDER BY DESC

SELECT *
FROM movies
ORDER BY title DESC;
```

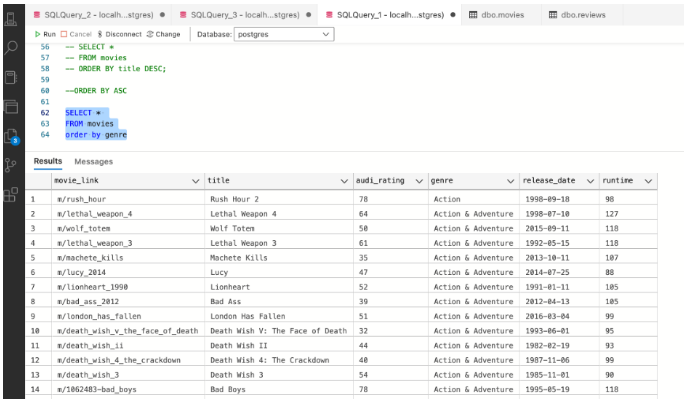
	movie_link	title	audi_rating	genre
1	m/oritsu_uchugun_oneamisu_no_tsubasa	Ôritsu uchûgun Oneamisu no tsubasa (Royal Space Forc…	80	Animation, Anime & Manga, Drama, Science Fiction & F…
2	m/evocateur_the_morton_downey_jr_movie	Évocateur: The Morton Downey Jr. Movie	84	Documentary, Special Interest
3	m/tie_me_up_tie_me_down	Átame! (Tie Me Up! Tie Me Down!)	79	Art House & International, Comedy, Drama, Romance
4	m/xxx_state_of_the_union	xXx: State of the Union	32	Action & Adventure
5	m/xxx_return_of_xander_cage	xXx: Return of Xander Cage	37	Action & Adventure, Drama
6	m/sleep_furiously	sleep furiously.	67	Documentary, Special Interest
7	m/mother_2017	mother!	51	Drama, Horror, Mystery & Suspense
8	m/iboy	iBoy	39	Action & Adventure, Drama, Mystery & Suspense, Scien…
9	m/bwoy	bwoy	50	Drama
10	m/aka_tommy_chong	a/k/a Tommy Chong	80	Documentary, Special Interest
11	m/rec_2	[Rec] 2	60	Drama, Horror, Mystery & Suspense, Special Interest
12	m/10009132-rec	[Rec]	82	Art House & International, Horror, Special Interest
13	m/rec_4_apocalypse	[REC] 4: Apocalypse	34	Art House & International, Drama, Horror
14	m/rec_3_genesis	[REC] 3 Genesis	27	Art House & International, Drama, Horror, Mystery & …
15	m/zulu_dawn	Zulu Dawn	62	Action & Adventure, Art House & International, Drama
16	m/zulu	Zulu	91	Classics, Drama
17	m/zorba_the_greek	Zorba the Greek	86	Action & Adventure, Art House & International, Class…

Fig. 11. Ordering in Movies() table in descending order.

VIII. Task 7: Problematic Queries and Performance Improvements

”Query execution analysis: identify three problematic queries (show their cost), where the performance can be improved. Provide a detailed execution plan (you may use EXPLAIN in PostgreSQL) on how you plan to improve these queries.”

Our first problematic query, shown in figure 18, is a simple query with a glaring inefficiency: We join Reviews() and Tables(), but we only look for attributes from Reviews()! In a database with a small schema, this may be obvious, but in larger schemas it is important to only join tables which would provide you with the desired attributes. This strategy of ”only necessary joins” is how we would improve

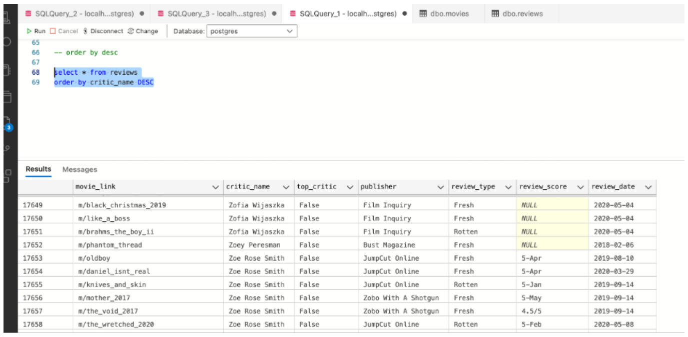


SQLQuery_2 - localh...stgres)
SQLQuery_3 - localh...stgres)
SQLQuery_1 - localh...stgres)
dbo.movies
dbo.reviews
Run
Cancel
Disconnect
Change
Database: postgres
Results
Messages

```
-- SELECT *
-- FROM movies
-- ORDER BY title DESC;

--ORDER BY ASC

SELECT *
FROM movies
order by genre
```

	movie_link	title	audi_rating	genre	release_date	runtime
1	m/rush_hour	Rush Hour 2	78	Action	1998-09-18	98
2	m/lethal_weapon_4	Lethal Weapon 4	64	Action & Adventure	1998-07-10	127
3	m/wolf_totem	Wolf Totem	50	Action & Adventure	2015-09-11	118
4	m/lethal_weapon_3	Lethal Weapon 3	61	Action & Adventure	1992-05-15	118
5	m/machete_kills	Machete Kills	35	Action & Adventure	2013-10-11	107
6	m/lucy_2014	Lucy	47	Action & Adventure	2014-07-25	88
7	m/lionheart_1990	Lionheart	52	Action & Adventure	1991-01-11	105
8	m/bad_ass_2012	Bad Ass	39	Action & Adventure	2012-04-13	105
9	m/london_has_fallen	London Has Fallen	51	Action & Adventure	2016-03-04	99
10	m/death_wish_v_the_face_of_death	Death Wish V: The Face of Death	32	Action & Adventure	1993-06-01	95
11	m/death_wish_ii	Death Wish II	44	Action & Adventure	1982-02-19	93
12	m/death_wish_4_the_crackdown	Death Wish 4: The Crackdown	40	Action & Adventure	1987-11-06	99
13	m/death_wish_3	Death Wish 3	54	Action & Adventure	1985-11-01	90
14	m/1062483-bad_boys	Bad Boys	78	Action & Adventure	1995-05-19	118

Fig. 12. Ordering in Movies() table in ascending order.

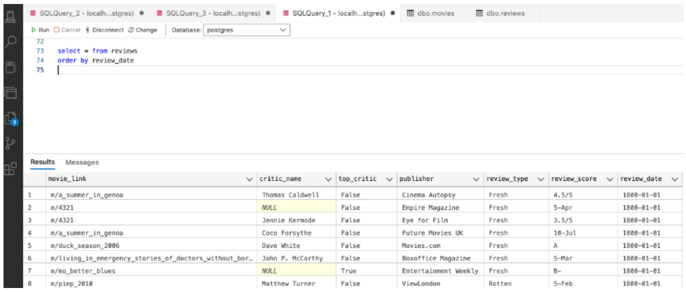


SQLQuery_2 - localh...stgres)
SQLQuery_3 - localh...stgres)
SQLQuery_1 - localh...stgres)
dbo.movies
dbo.reviews
Run
Cancel
Disconnect
Change
Database: postgres
Results
Messages

```
-- order by desc

select * from reviews
order by critic_name DESC
```

	movie_link	critic_name	top_critic	publisher	review_type	review_score	review_date
17649	m/black_christmas_2019	Zofia Wijaszka	False	Film Inquiry	Fresh	NULL	2020-05-04
17650	m/like_a_boss	Zofia Wijaszka	False	Film Inquiry	Fresh	NULL	2020-05-04
17651	m/brahms_the_boy_ii	Zofia Wijaszka	False	Film Inquiry	Rotten	NULL	2020-05-04
17652	m/phantom_thread	Zoey Peresman	False	Bust Magazine	Fresh	NULL	2018-02-06
17653	m/oldboy	Zoe Rose Smith	False	JumpCut Online	Fresh	5-Apr	2019-08-10
17654	m/daniel_isnt_real	Zoe Rose Smith	False	JumpCut Online	Fresh	5-Apr	2020-03-29
17655	m/knives_and_skin	Zoe Rose Smith	False	JumpCut Online	Rotten	5-Jan	2019-09-14
17656	m/mother_2017	Zoe Rose Smith	False	Zobo With A Shotgun	Fresh	5-May	2019-09-14
17657	m/the_void_2017	Zoe Rose Smith	False	Zobo With A Shotgun	Fresh	4.5/5	2019-09-14
17658	m/the_wretched_2020	Zoe Rose Smith	False	JumpCut Online	Rotten	5-Feb	2020-05-08

Fig. 13. Ordering in Reviews() table in descending order.



SQLQuery_2 - localh...stgres)
SQLQuery_3 - localh...stgres)
SQLQuery_1 - localh...stgres)
dbo.movies
dbo.reviews
Run
Cancel
Disconnect
Change
Database: postgres
Results
Messages

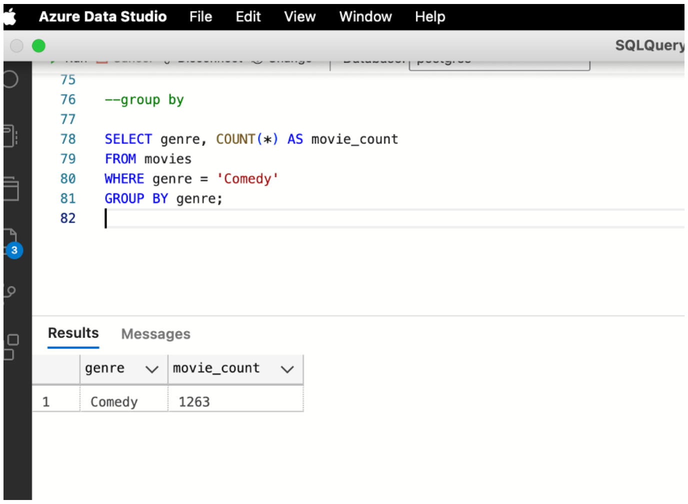
```
select * from reviews
order by review_date
```

	movie_link	critic_name	top_critic	publisher	review_type	review_score	review_date
1	m/a_summer_in_genoa	Thomas Caldwell	False	Cinema Autopsy	Fresh	4.5/5	1800-01-01
2	m/4321	NULL	False	Empire Magazine	Fresh	5-Apr	1800-01-01
3	m/4321	Jennie Kermode	False	Eye for Film	Fresh	3.5/5	1800-01-01
4	m/a_summer_in_genoa	Coco Forsythe	False	Future Movies UK	Fresh	10-Jul	1800-01-01
5	m/duck_season_2006	Dave White	False	Movies.com	Fresh	A	1800-01-01
6	m/living_in_emergency_stories_of_doctors_without_bor…	John P. McCarthy	False	Boxoffice Magazine	Fresh	5-Mar	1800-01-01
7	m/mo_better_blues	NULL	True	Entertainment Weekly	Fresh	B-	1800-01-01
8	m/pimp_2010	Matthew Turner	False	ViewLondon	Rotten	5-Feb	1800-01-01

Fig. 14. Ordering in Reviews() table in ascending order.

on queries like this in large databases. Otherwise, as shown in the EXPLAIN function, you will incur extra time costs from the join operation.

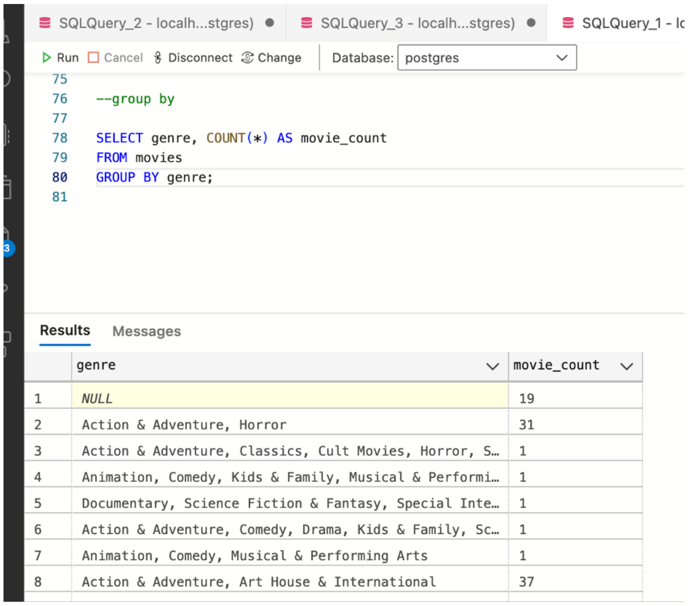
The second problematic query we present is shown in figure 19. In this case, we join Reviews() and Movies(), only to later apply a condition on the release date of the movie in question. This is problematic because the joined table will have more total tuples than Movies(), and will require a longer scan than if we had just applied the release date condition to the Movies() table BEFORE joining it, and this inefficiency is reflected in the JOIN cost in the figure. In bigger databases it is important to apply conditions before joins when possible, in order to shrink the tables and lower the cost of the join. Applying conditions before joins is another useful data strategy to manage queries like this one.



```
--group by

SELECT genre, COUNT(*) AS movie_count
FROM movies
WHERE genre = 'Comedy'
GROUP BY genre;
```

	genre	movie_count
1	Comedy	1263

Fig. 15. Performing GroupBy operation to get the total count of Comedy Movies in the Movies() table.

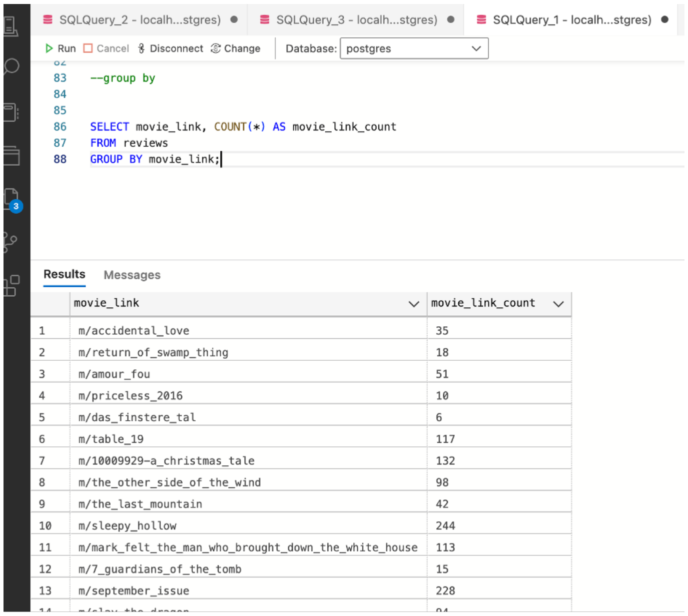


```
--group by

SELECT genre, COUNT(*) AS movie_count
FROM movies
GROUP BY genre;
```

	genre	movie_count
1	NULL	19
2	Action & Adventure, Horror	31
3	Action & Adventure, Classics, Cult Movies, Horror, S…	1
4	Animation, Comedy, Kids & Family, Musical & Performi…	1
5	Documentary, Science Fiction & Fantasy, Special Inte…	1
6	Action & Adventure, Comedy, Drama, Kids & Family, Sc…	1
7	Animation, Comedy, Musical & Performing Arts	1
8	Action & Adventure, Art House & International	37

Fig. 16. Performing GroupBy operation in the Movies() table, grouping movies by genre.

Our third and final query, presented in figure 20, has a number of problematic components that increase the total execution time of the process. For one thing, including the phrase "HAVING COUNT ¿ 0" is redundant because if a movie exists at all, it will have a title since we're grouping by movie link. Additionally, we scan on two conditions in a nested select statement instead of just applying them together. This is cumbersome and leads to many of the inefficiencies you see in the explanation table associated with the figure. In this case, the query is problematic because it includes redundant clauses and unnecessary nesting which bloats the runtime of the query. To combat this, we would modify the query to put both WHERE clauses in a single SELECT statement, and



SQLQuery_2 - localh...stgres) ●
SQLQuery_3 - localh...stgres) ●
SQLQuery_1 - localh...stgres) ●
▷ Run
□ Cancel
Disconnect
Change
Database: postgres
Results
Messages

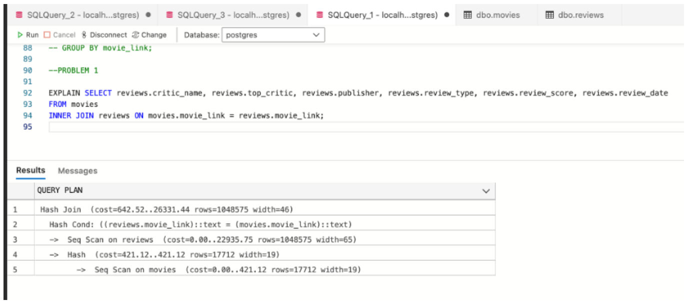
```sql
--group by


SELECT movie_link, COUNT(*) AS movie_link_count
FROM reviews
GROUP BY movie_link;
```

	movie_link	movie_link_count
1	m/accidental_love	35
2	m/return_of_swamp_thing	18
3	m/amour_fou	51
4	m/priceless_2016	10
5	m/das_finstere_tal	6
6	m/table_19	117
7	m/10009929-a_christmas_tale	132
8	m/the_other_side_of_the_wind	98
9	m/the_last_mountain	42
10	m/sleepy_hollow	244
11	m/mark_felt_the_man_who_brought_down_the_white_house	113
12	m/7_guardians_of_the_tomb	15
13	m/september_issue	228

Fig. 17. Performing GroupBy operation using to acquire the total number of movie links for each movie.

remove the HAVING clause entirely. In general you want to simplify overcomplicated queries to prevent running excessive scans, and this is especially true for larger databases where each scan takes much longer.

This concludes Phase 2 of the Final Report.



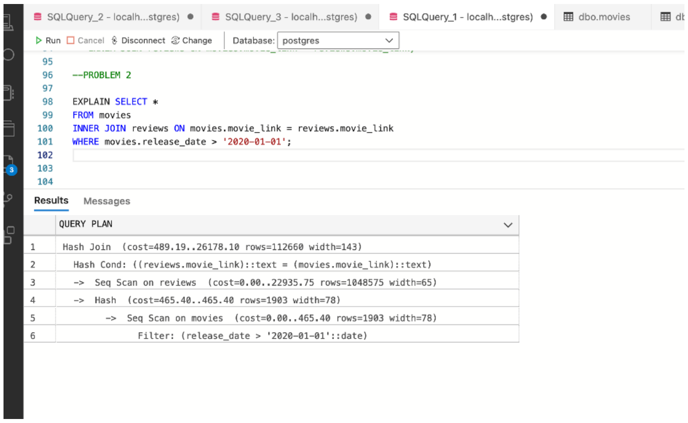
SQLQuery_2 - localh...stgres) ●
SQLQuery_3 - localh...stgres) ●
SQLQuery_1 - localh...stgres) ●
dbo.movies
dbo.reviews
▷ Run
□ Cancel
Disconnect
Change
Database: postgres
Results
Messages

```sql
-- GROUP BY movie_link;

--PROBLEM 1

EXPLAIN SELECT reviews.critic_name, reviews.top_critic, reviews.publisher, reviews.review_type, reviews.review_score, reviews.review_date
FROM movies
INNER JOIN reviews ON movies.movie_link = reviews.movie_link;
```

```
   QUERY PLAN
1  Hash Join  (cost=642.52..26331.44 rows=1048575 width=46)
2    Hash Cond: ((reviews.movie_link)::text = (movies.movie_link)::text)
3    ->  Seq Scan on reviews  (cost=0.00..22935.75 rows=1048575 width=65)
4    ->  Hash  (cost=421.12..421.12 rows=17712 width=19)
5          ->  Seq Scan on movies  (cost=0.00..421.12 rows=17712 width=19)
```

Fig. 18. Problem Query 1: It is inefficient to join several tables when you're only searching for values in one of them.

References :- Link to Rotten Tomatoes Dataset

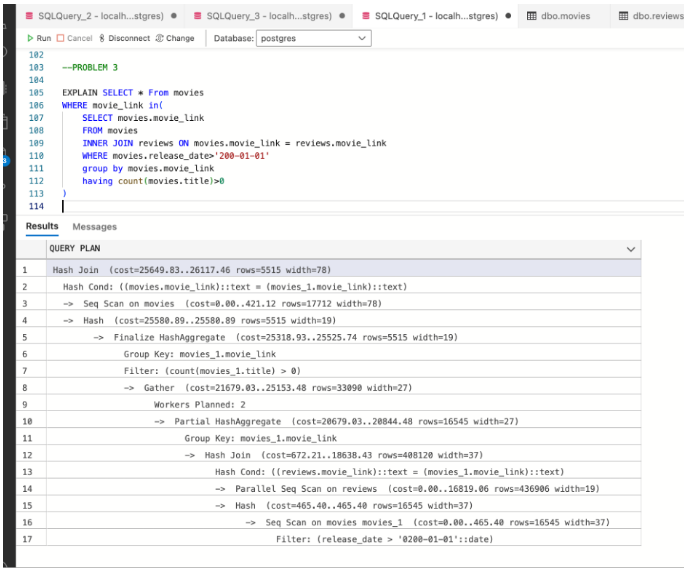


SQLQuery_2 - localh...stgres) ●
SQLQuery_3 - localh...stgres) ●
SQLQuery_1 - localh...stgres) ●
dbo.movies
▷ Run
□ Cancel
Disconnect
Change
Database: postgres
Results
Messages

```sql
--PROBLEM 2

EXPLAIN SELECT *
FROM movies
INNER JOIN reviews ON movies.movie_link = reviews.movie_link
WHERE movies.release_date > '2020-01-01';
```

```
   QUERY PLAN
1  Hash Join  (cost=489.19..26178.10 rows=112660 width=143)
2    Hash Cond: ((reviews.movie_link)::text = (movies.movie_link)::text)
3    ->  Seq Scan on reviews  (cost=0.00..22935.75 rows=1048575 width=65)
4    ->  Hash  (cost=465.40..465.40 rows=1903 width=78)
5          ->  Seq Scan on movies  (cost=0.00..465.40 rows=1903 width=78)
6                Filter: (release_date > '2020-01-01'::date)
```

Fig. 19. Problem Query 2: It is inefficient to join before applying a condition; We should apply the condition to lower the amount of tuples before joining.



SQLQuery_2 - localh...stgres) ●
SQLQuery_3 - localh...stgres) ●
SQLQuery_1 - localh...stgres) ●
dbo.movies
dbo.reviews
▷ Run
□ Cancel
Disconnect
Change
Database: postgres
Results
Messages

```sql
--PROBLEM 3

EXPLAIN SELECT * From movies
WHERE movie_link in(
    SELECT movies.movie_link
    FROM movies
    INNER JOIN reviews ON movies.movie_link = reviews.movie_link
    WHERE movies.release_date>'200-01-01'
    group by movies.movie_link
    having count(movies.title)>0
)
```

```
    QUERY PLAN
1   Hash Join  (cost=25649.83..26117.46 rows=5515 width=78)
2     Hash Cond: ((movies.movie_link)::text = (movies_1.movie_link)::text)
3     ->  Seq Scan on movies  (cost=0.00..421.12 rows=17712 width=78)
4     ->  Hash  (cost=25580.89..25580.89 rows=5515 width=19)
5           ->  Finalize HashAggregate  (cost=25318.93..25525.74 rows=5515 width=19)
6                 Group Key: movies_1.movie_link
7                 Filter: (count(movies_1.title) > 0)
8                 ->  Gather  (cost=21679.03..25153.48 rows=33090 width=27)
9                       Workers Planned: 2
10                      ->  Partial HashAggregate  (cost=20679.03..20844.48 rows=16545 width=27)
11                            Group Key: movies_1.movie_link
12                            ->  Hash Join  (cost=672.21..18638.43 rows=408120 width=37)
13                                  Hash Cond: ((reviews.movie_link)::text = (movies_1.movie_link)::text)
14                                  ->  Parallel Seq Scan on reviews  (cost=0.00..16819.06 rows=436906 width=19)
15                                  ->  Hash  (cost=465.40..465.40 rows=16545 width=37)
16                                        ->  Seq Scan on movies movies_1  (cost=0.00..465.40 rows=16545 width=37)
17                                              Filter: (release_date > '0200-01-01'::date)
```

Fig. 20. Problem Query 3: There are several inefficiencies here; The conditions are separated and the addition of a COUNT call should be optimized out.