Part B

6.1:

Notes: If the query contains nested selects is ok. Creating permanent tables is not ok.

1) Top 5 highest rated movies per user gender

```
SELECT * FROM (
  SELECT u.gender,
         r.movie_id,
         AVG(rating),
         ROW_NUMBER() OVER(PARTITION BY u.gender ORDER BY u.gender, avg(rating)
DESC) AS RowNumber
  FROM rating r
  INNER JOIN user_profile u ON r.user_profile_id = u.user_profile_id
  GROUP BY u.gender, r.movie_id
  ORDER BY u.gender, avg(rating) DESC
) top_movies
WHERE RowNumber <= 5
```

Data Output | Explain | Messages | Query History

	gender character varying	movie_id integer	avg numeric	rownumber bigint
1	F	3817	5.0000000000000000	1
2	F	3292	5.0000000000000000	2
3	F	2675	5.0000000000000000	3
4	F	681	5.0000000000000000	4
5	F	1450	5.0000000000000000	5
6	M	3656	5.0000000000000000	1
7	M	3280	5.0000000000000000	2
8	M	3233	5.0000000000000000	3
9	M	3172	5.0000000000000000	4
10	M	439	5.0000000000000000	5

2) Top 5 highest rated movies per user age group

```
SELECT * FROM (
  SELECT u.age,
            r.movie_id,
            AVG(rating),
            ROW_NUMBER() OVER(PARTITION BY u.age ORDER BY u.age, avg(rating) DESC) AS
RowNumber
  FROM rating r
  INNER JOIN user_profile u ON r.user_profile_id = u.user_profile_id
  GROUP BY u.age, r.movie_id
  ORDER BY u.age, avg(rating) DESC
) top_movies
WHERE RowNumber <= 5
```

Data Output | Explain | Messages | Query History

	age integer	movie_id integer	avg numeric	rownumber bigint
1	1	3789	5.0000000000000000	1
2	1	57	5.0000000000000000	2
3	1	937	5.0000000000000000	3
4	1	2518	5.0000000000000000	4
5	1	299	5.0000000000000000	5
6	18	2905	5.0000000000000000	1
7	18	2192	5.0000000000000000	2
8	18	2215	5.0000000000000000	3
9	18	3038	5.0000000000000000	4
10	18	3280	5.0000000000000000	5
11	25	3640	5.0000000000000000	1
12	25	167	5.0000000000000000	2
13	25	853	5.0000000000000000	3
14	25	3232	5.0000000000000000	4
15	25	1420	5.0000000000000000	5
16	35	598	5.0000000000000000	1
17	35	1486	5.0000000000000000	2
18	35	1164	5.0000000000000000	3
19	35	2503	5.0000000000000000	4
20	35	2911	5.0000000000000000	5
21	45	1659	5.0000000000000000	1
22	45	1421	5.0000000000000000	2
23	45	3245	5.0000000000000000	3
24	45	1859	5.0000000000000000	4
25	45	84	5.0000000000000000	5
26	50	38	5.0000000000000000	1
27	50	2760	5.0000000000000000	2
28	50	3645	5.0000000000000000	3
29	50	771	5.0000000000000000	4
30	50	3233	5.0000000000000000	5
31	56	1570	5.0000000000000000	1
32	56	1812	5.0000000000000000	2
33	56	298	5.0000000000000000	3
34	56	1348	5.0000000000000000	4
35	56	3866	5.0000000000000000	5

3) Top 5 highest rated movies per use ocupation

```
SELECT * FROM (
  SELECT u.occupation,
            r.movie_id,
            AVG(rating),
            ROW_NUMBER() OVER(PARTITION BY u.occupation ORDER BY u.occupation,
avg(rating) DESC) AS RowNumber
  FROM rating r
  INNER JOIN user_profile u ON r.user_profile_id = u.user_profile_id
  GROUP BY u.occupation, r.movie_id
  ORDER BY u.occupation, avg(rating) DESC
) top_movies
WHERE RowNumber <= 5
```

	occupation integer	movie_id integer	avg numeric	rownumber bigint
1	0	146	00000000000	1
2	0	570	00000000000	2
3	0	598	00000000000	3
4	0	2503	00000000000	4
5	0	2538	00000000000	5
6	1	801	00000000000	1
7	1	854	00000000000	2
8	1	939	00000000000	3
9	1	966	00000000000	4
10	1	1002	00000000000	5
11	2	2131	00000000000	1
12	2	2156	00000000000	2
13	2	2175	00000000000	3
14	2	2293	00000000000	4
15	2	2309	00000000000	5
16	3	77	00000000000	1
17	3	97	00000000000	2
18	3	106	00000000000	3
19	3	297	00000000000	4
20	3	304	00000000000	5
21	4	53	00000000000	1
22	4	439	00000000000	2
23	4	462	00000000000	3
24	4	583	00000000000	4
25	4	602	00000000000	5
26	5	650	00000000000	1

6.2

Notes: Step 1 and 4 can be combined. In my solution I combine the steps. Alternatively, one could create the table in step 1) and populate it with a select.

1) Create a table called movie_genre containing the fields movie_id, genre

```
SELECT movie_id, unnest(string_to_array(genres,'|')) as genre
INTO movie_genre
FROM movie
```

2) Define a foreign key for movie_id

```
ALTER TABLE movie_genre ADD  FOREIGN KEY(movie_id) REFERENCES movie(movie_id);
```

3) Create a btree index for the field movie_id.

```
CREATE INDEX movie_genre_movie_id ON movie_genre USING btree (movie_id);
```

4) Already done in 1.

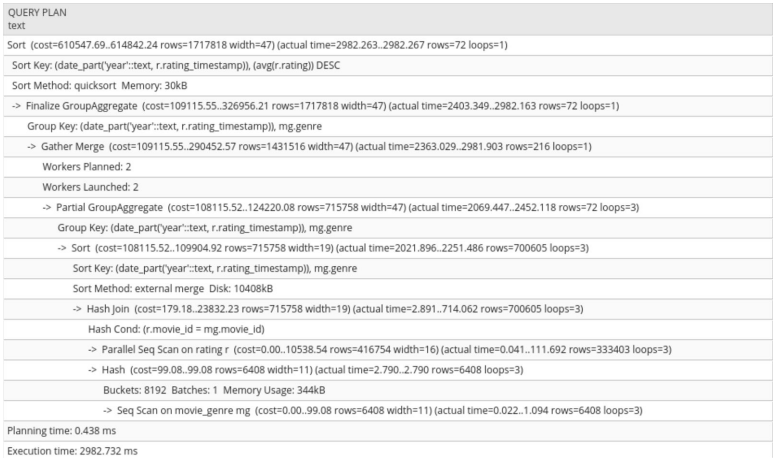
6.3

Notes: For 1.1, any query that does not uses a temporal table is ok. For 1.2, Students should use the cost to compare the queries. If the student's proposed approach improves the cost from the original query, no further explanation is required. If not, the student should explain why the index did not help.

1.1) Provide the SQL for the query:

```
SELECT extract(year from r.rating_timestamp), mg.genre, avg(r.rating)
FROM rating r
INNER JOIN movie_genre mg ON r.movie_id = mg.movie_id
GROUP BY extract(year from r.rating_timestamp), mg.genre
ORDER BY extract(year from r.rating_timestamp) ASC, avg(r.rating) DESC
```

1.2) The execution plan for the query:



QUERY PLAN text
Sort (cost=610547.69..614842.24 rows=1717818 width=47) (actual time=2982.263..2982.267 rows=72 loops=1)
Sort Key: (date_part('year'::text, r.rating_timestamp)), (avg(r.rating)) DESC
Sort Method: quicksort Memory: 30kB
-> Finalize GroupAggregate (cost=109115.55..326956.21 rows=1717818 width=47) (actual time=2403.349..2982.163 rows=72 loops=1)
Group Key: (date_part('year'::text, r.rating_timestamp)), mg.genre
-> Gather Merge (cost=109115.55..290452.57 rows=1431516 width=47) (actual time=2363.029..2981.903 rows=216 loops=1)
Workers Planned: 2
Workers Launched: 2
-> Partial GroupAggregate (cost=108115.52..124220.08 rows=715758 width=47) (actual time=2069.447..2452.118 rows=72 loops=3)
Group Key: (date_part('year'::text, r.rating_timestamp)), mg.genre
-> Sort (cost=108115.52..109904.92 rows=715758 width=19) (actual time=2021.896..2251.486 rows=700605 loops=3)
Sort Key: (date_part('year'::text, r.rating_timestamp)), mg.genre
Sort Method: external merge Disk: 10408kB
-> Hash Join (cost=179.18..23832.23 rows=715758 width=19) (actual time=2.891..714.062 rows=700605 loops=3)
Hash Cond: (r.movie_id = mg.movie_id)
-> Parallel Seq Scan on rating r (cost=0.00..10538.54 rows=416754 width=16) (actual time=0.041..111.692 rows=333403 loops=3)
-> Hash (cost=99.08..99.08 rows=6408 width=11) (actual time=2.790..2.790 rows=6408 loops=3)
Buckets: 8192 Batches: 1 Memory Usage: 344kB
-> Seq Scan on movie_genre mg (cost=0.00..99.08 rows=6408 width=11) (actual time=0.022..1.094 rows=6408 loops=3)
Planning time: 0.438 ms
Execution time: 2982.732 ms

2.1) Provide the SQL for your new index:

CREATE INDEX rating_timestamp_year ON rating USING btree (extract(year from rating_timestamp));

REINDEX INDEX rating_timestamp_year

2.2) New execution plan and description:

Sort (cost=610547.69..614842.24 rows=1717818 width=47) (actual time=3000.906..3000.910 rows=72 loops=1)	
Sort Key: (date_part('year'::text, r.rating_timestamp)), (avg(r.rating)) DESC	9 Sort using quicksort
Sort Method: quicksort Memory: 30kB	
-> Finalize GroupAggregate (cost=109115.55..326956.21 rows=1717818 width=47) (actual time=2351.261..3000.785 rows=72 loops=1)	8 Partial aggregation of PSQL - Final Aggregate
Group Key: (date_part('year'::text, r.rating_timestamp)), mg.genre	
-> Gather Merge (cost=109115.55..290452.57 rows=1431516 width=47) (actual time=2314.402..3000.538 rows=216 loops=1)	7 Combine workers
Workers Planned: 2	
Workers Launched: 2	
-> Partial GroupAggregate (cost=108115.52..124220.08 rows=715758 width=47) (actual time=2213.183..2818.710 rows=72 loops=3)	6 Partial aggregation of PSQL - Partial Aggregate
Group Key: (date_part('year'::text, r.rating_timestamp)), mg.genre	
-> Sort (cost=108115.52..109904.92 rows=715758 width=19) (actual time=2119.055..2473.910 rows=700605 loops=3)	5 Sort based on the year and genre
Sort Key: (date_part('year'::text, r.rating_timestamp)), mg.genre	
Sort Method: external merge Disk: 37456kB	
-> Hash Join (cost=179.18..23832.23 rows=715758 width=19) (actual time=7.592..767.078 rows=700605 loops=3)	4 Rows are hashed in-memory and checked if the condition is met
Hash Cond: (r.movie_id = mg.movie_id)	
-> Parallel Seq Scan on rating r (cost=0.00..10538.54 rows=416754 width=16) (actual time=0.046..164.497 rows=333403 loops=3)	3 Parallel scan on rating, as the table is larger
-> Hash (cost=99.08..99.08 rows=6408 width=11) (actual time=7.466..7.466 rows=6408 loops=3)	2 Creates hashes
Buckets: 8192 Batches: 1 Memory Usage: 344kB	
-> Seq Scan on movie_genre mg (cost=0.00..99.08 rows=6408 width=11) (actual time=0.022..1.021 rows=6408 loops=3)	1 PSQL Scans the full table as it considers it small
Planning time: 0.864 ms	
Execution time: 3003.030 ms	

2.3) Does it improve the execution time at all?

It did not improve the efficiency of the query. The cost is the same for both queries. Possible reasons:

- The query optimizer is not using the index. We can confirm this by checking select * from pg_stat_user_indexes.
- The b-tree index is too small (4 years: 2000, 2001, 2002, 2003)
- The Group Key function requires two keys in the aggregation.

6.4:

Notes: If the students created queries *without* json functions and the result is correct, it counts only 50% of the points.

1.1) Write a query that counts the number times each property occurs:

```
SELECT json_key, count(*) json_key_count  FROM
(SELECT movie_id, json_object_keys(dbpedia_content) AS json_key FROM dbpedia) keys
GROUP BY json_key
ORDER BY json_key_count DESC
```

	json_key text	json_key_count bigint
1	abstract	3266
2	wikiPageR...	3265
3	wikiPageID	3265
4	subject	3135
5	director	3115

1.2.1) Average rating of the starring actors

```
SELECT actor, avg(rating) FROM
        (SELECT
                ma.movie_id,
                cast(ma.actor as text),
                r.rating
         FROM (SELECT
                        movie_id,
                        json_array_elements(dbpedia_content->'starring') as actor
                        FROM dbpedia
                        WHERE json_typeof(dbpedia_content->'starring')='array') ma
        INNER JOIN rating r ON r.movie_id = ma.movie_id) mar
GROUP BY actor
ORDER BY avg(rating) DESC
```

	actor text	avg numeric
1	"Han_Dongfang"	5.0000000000000000
2	"Ding_Zilin"	5.0000000000000000
3	"Dai_Qing"	5.0000000000000000
4	"Ben_Becker"	5.0000000000000000
5	"Lucille_Ball"	5.0000000000000000

1.2.2) Average rating of the starring actors per user gender:

```
SELECT gender, actor, avg(rating) FROM
        (SELECT
                ma.movie_id,
                cast(ma.actor as text),
                r.rating,
                u.gender
         FROM (SELECT
                        movie_id,
                        json_array_elements(dbpedia_content->'starring') as actor
                        FROM dbpedia
                        WHERE json_typeof(dbpedia_content->'starring')='array') ma
         INNER JOIN rating r ON r.movie_id = ma.movie_id
         INNER JOIN user_profile u ON r.user_profile_id = u.user_profile_id
        ) mar
GROUP BY gender, actor
ORDER BY avg(rating) DESC
```

	gender character varying	actor text	avg numeric
1	F	"Carmelo_Di_Mazzarelli"	5.0000000000000000
2	F	"Charles_K._French"	5.0000000000000000
3	F	"Carl_Miller_(actor)"	5.0000000000000000
4	F	"Adolphe_Menjou"	5.0000000000000000
5	F	"Clarence_Geldart"	5.0000000000000000

1.2.3) Make a join of both queries and give actor, average_rating, averating_rating_M, average_rating_F:

```
SELECT
	ar.actor,
	average_rating,
	average_rating_F,
	average_rating_M
FROM
(SELECT actor, avg(mar.rating) as average_rating FROM
	(SELECT
		ma.movie_id,
		cast(ma.actor as text),
		r.rating
	 FROM (SELECT
			movie_id,
			json_array_elements(dbpedia_content->'starring') as actor
			FROM dbpedia
			WHERE json_typeof(dbpedia_content->'starring')='array') ma
	 INNER JOIN rating r ON r.movie_id = ma.movie_id
	 INNER JOIN user_profile u ON r.user_profile_id = u.user_profile_id
	) mar
GROUP BY actor
) ar
INNER JOIN
(SELECT actor,
	  avg(rating_F) as average_rating_F,
	  avg(rating_M) as average_rating_M
 FROM (SELECT
		ma.movie_id,
		cast(ma.actor as text),
		CASE u.gender WHEN 'F' THEN r.rating END as rating_F,
	CASE u.gender WHEN 'M' THEN r.rating END as rating_M
	 FROM (SELECT
			movie_id,
			json_array_elements(dbpedia_content->'starring') as actor
			FROM dbpedia
			WHERE json_typeof(dbpedia_content->'starring')='array') ma
	 INNER JOIN rating r ON r.movie_id = ma.movie_id
	 INNER JOIN user_profile u ON r.user_profile_id = u.user_profile_id
	 ) gar
GROUP BY actor
) afm
ON ar.actor = afm.actor
ORDER BY average_rating DESC
```

	actor text	average_rating numeric	average_rating_f numeric	average_rating_m numeric
1	"Han_Dongfang"	5.0000000000000000	5.0000000000000000	5.0000000000000000
2	"Ding_Zilin"	5.0000000000000000	5.0000000000000000	5.0000000000000000
3	"Dai_Qing"	5.0000000000000000	5.0000000000000000	5.0000000000000000
4	"Ben_Becker"	5.0000000000000000	[null]	5.0000000000000000
5	"Lucille_Ball"	5.0000000000000000	[null]	5.0000000000000000

1.2.4) If these queries were executed often, how would you improve the execution time?

- Change from JSON to JSONB
- Create an index on dbpedia_content->'starring'
- Check with the explain analyze if the new index is helping
- Check for postgreSQL JSON functions that would work with the new index
- I would create materialized views from the previous queries

2.1) How frequent the actors participate together? Make a query that returns actor_1, actor_2, total_movies_together

Note: In my solution, the pair (actor_1,actor_2) is duplicated as (actor_2,actor_1).

```
SELECT
        t1.actor as actor_1,
        t2.actor as actor_2,
        count(*) as total_movies_together
FROM
 (SELECT movie_id, cast(json_array_elements(dbpedia_content->'starring') as text) actor
  FROM dbpedia
  WHERE json_typeof(dbpedia_content->'starring')='array') t1
INNER JOIN
 (SELECT movie_id, cast(json_array_elements(dbpedia_content->'starring') as text) as actor
  FROM dbpedia
  WHERE json_typeof(dbpedia_content->'starring')='array') t2
ON t1.movie_id = t2.movie_id
WHERE t1.actor != t2.actor
GROUP BY actor_1, actor_2
ORDER BY total_movies_together DESC
```

	actor_1 text	actor_2 text	total_movies_together bigint
1	"James_Doohan"	"Walter_Koenig"	7
2	"Dave_Goelz"	"Frank_Oz"	7
3	"Frank_Oz"	"Dave_Goelz"	7
4	"Walter_Koenig"	"James_Doohan"	7
5	"Nichelle_Nichols"	"George_Takei"	6

2.2) If this query was executed often, how would you improve the execution time? (Use the concepts seen in class)

- Change from JSON to JSONB
- Create an index on dbpedia_content->'starring'
- Check with the explain analyze if the new index is helping
- Check for postgreSQL JSON functions that would work with the new index
- I would create materialized views from the previous query