IMDB STEP 2

INF553 Database Management Systems

24 novembre 2022

Jia Jean LAW and Alhussein JAMIL

1 Introduction

1.1 Experimental conditions

1. Postgres version 15
2. Operating system: Windows 11 Home 64-bit
3. Disk size : 1TB
4. Transfer rate : 131.95 Mb/s (read), 112.83 Mb/s (write)
5. Memory size 32768 MB DDR4

1.2 Selection of five queries

For each of the seven queries, we executed "explain analyze + QUERY" 20 times consecutively using a Java script to ensure that we obtain stable query times (these execution times can be found in Section 3, "Final Results"). Thereafter, we chose to optimize queries 1, 3, 4, 6 and 7: queries 2 and 5 are simpler queries that have a short query time (2.3s and 0.2s respectively), hence they have less room for optimization.

To summarize, we use the following strategies to optimize the queries.

1. Creating indexes for columns that we select tuples on: this allows the database to find and retrieve the requested rows more quickly
2. Rewriting query
3. Disabling sequential scan to manually privilege index scans or parallel scans

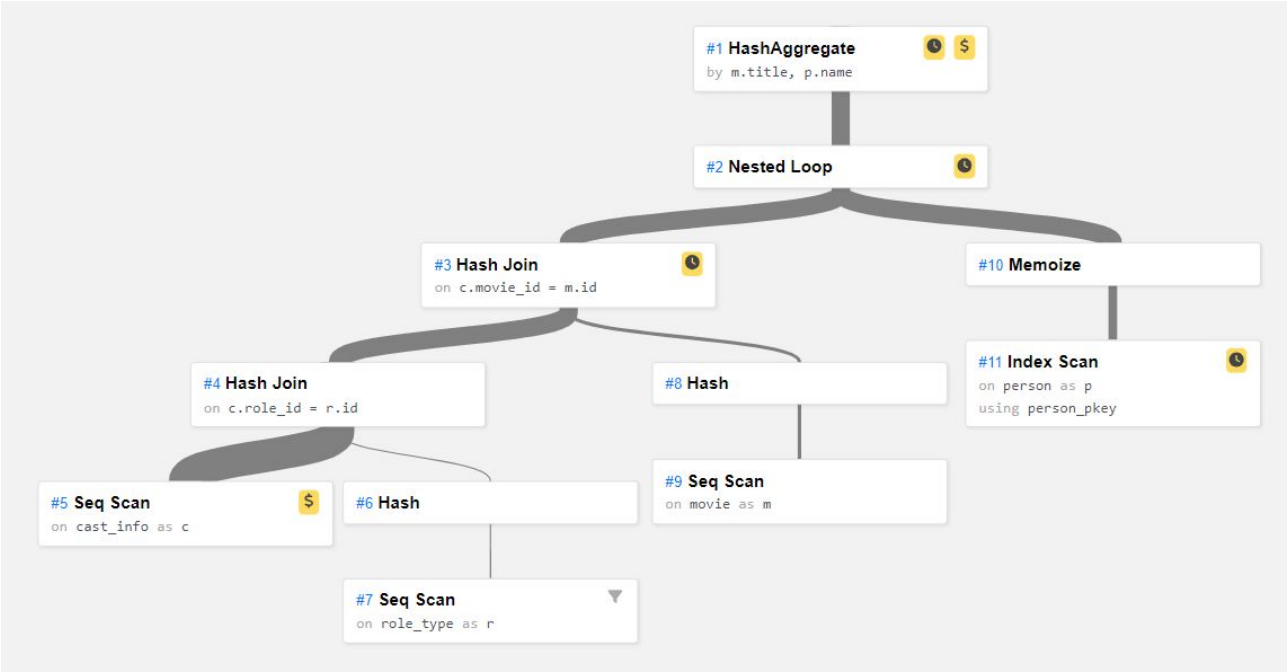
In section 2, we present, for each query, the optimizations proposed, and the query plans before and after optimization. In section 3, we present our final results, notably the query times obtained before and after optimization, and in section 4, we present some other optimizations that we tried that did not produce results.

2 Details of optimizations

2.1 Query 1

Original query:

```
SELECT DISTINCT m.title, p.name from movie as m
        join cast_info as c on c.movie_id=m.id
        join person as p on p.id = c.person_id
        join role_type as r on r.id = c.role_id
                where r.role = 'actor' or r.role = 'actress';
```



Optimizations :

- **"set enable_seqscan = OFF" (Cost :** $2.738 \cdot 10^{-4}s$**):** We tried adding indexes on cast_info.role_id and we found that the query takes a longer time. We finally decided to parallelize the sequential scan, making use of more workers, by turning off sequential scan.
- **Query modification "or" to "where r.role in {'actor','actress'"}:** "in" in the WHERE clause is more efficient as there is an index on role_type (explained in query 3). When "in" is used, the values in the list are sorted and retrieved using a binary search.

Plan after optimizations

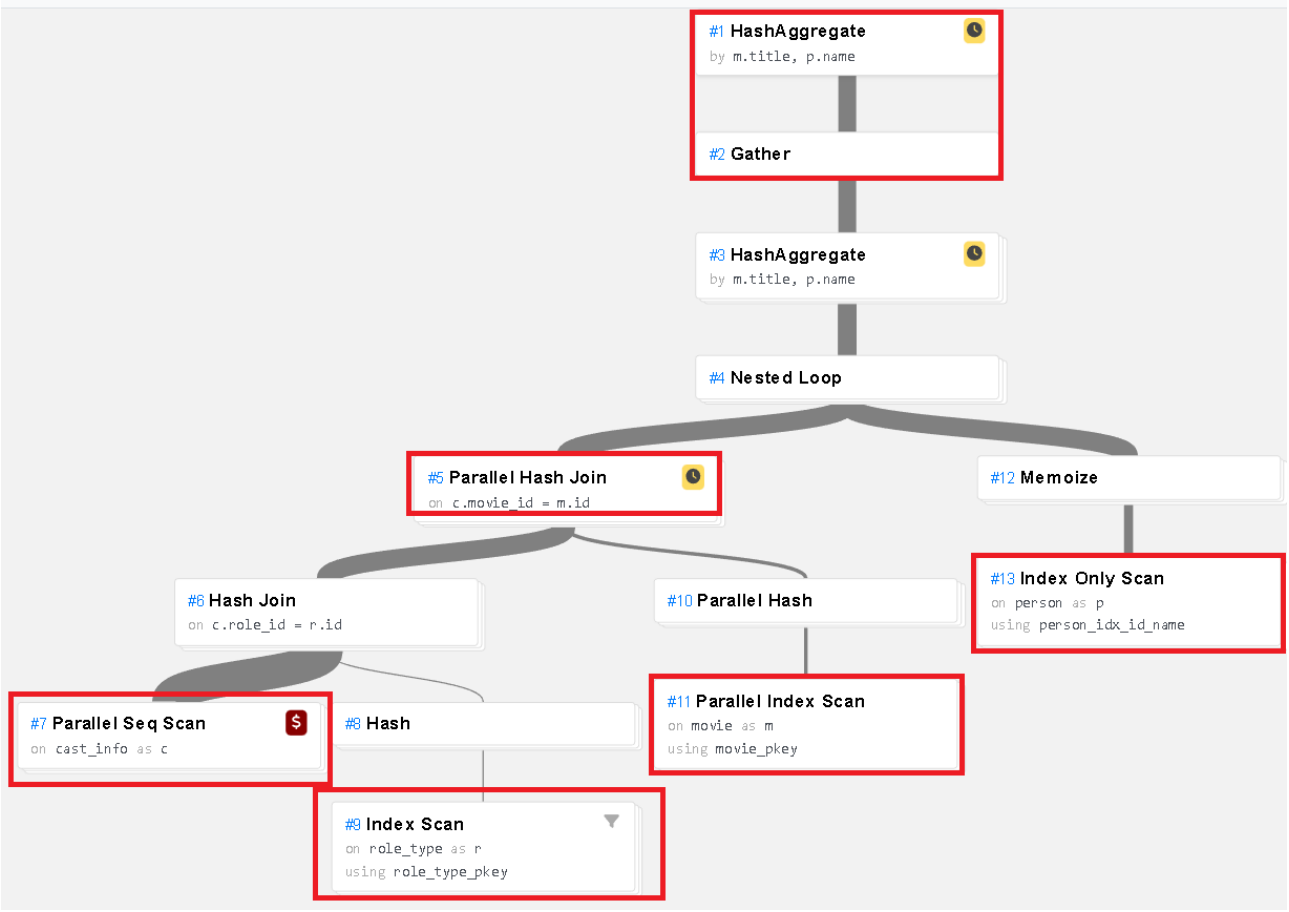
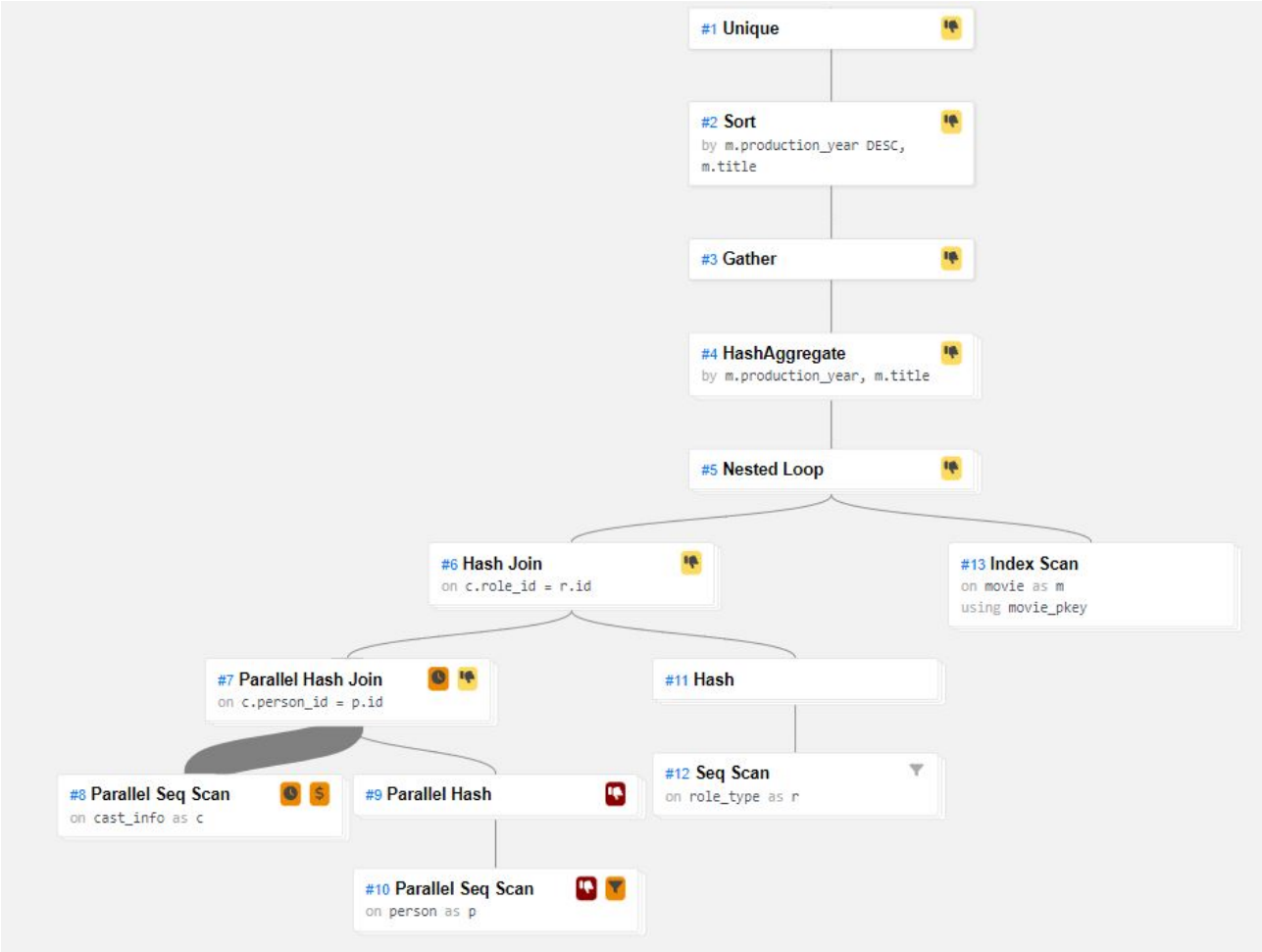


Figure 1: Parallel scans and index scans are prioritized after optimizations.

2.2 Query 3

```
SELECT DISTINCT m.title, m.production_year from movie as m
        join cast_info as c on c.movie_id=m.id
        join person as p on p.id = c.person_id
        join role_type as r on r.id = c.role_id
                where (r.role = 'actor' or r.role = 'actress') and p.name =
                    'Cage, Nicolas'
                ORDER BY m.production_year DESC;
```



Optimizations :

- **Creating an index on person.name (Cost :** 26.69s**) :** We are only interested in movies played by Nicolas Cage, hence we added an index on person.name to be able to retrieve all tuples corresponding to Nicolas Cage more quickly. This index is also useful for other queries that select based on person.name.
- **Query modification "or" to "in"** (cf Query 1).

Plan after optimizations

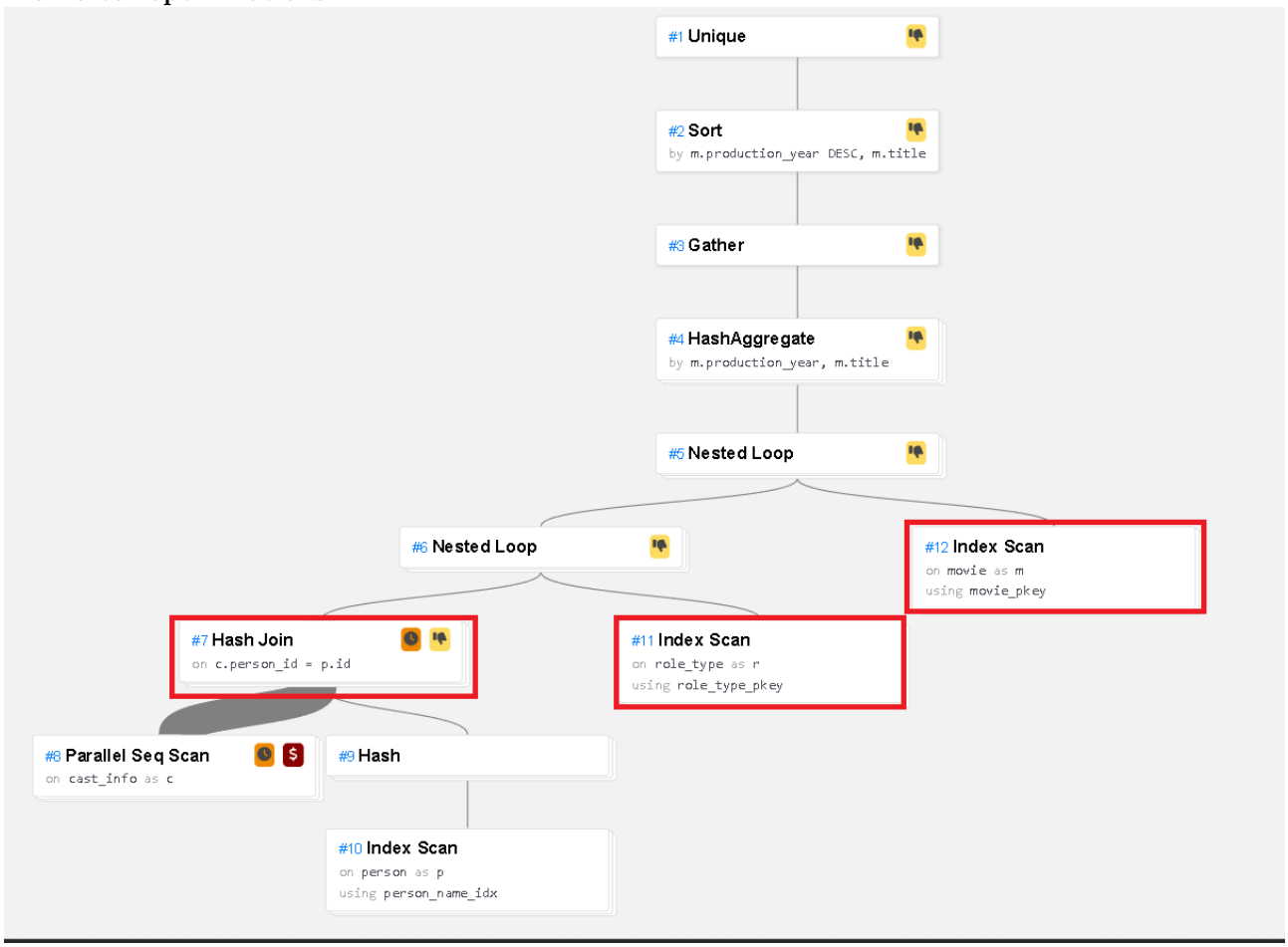
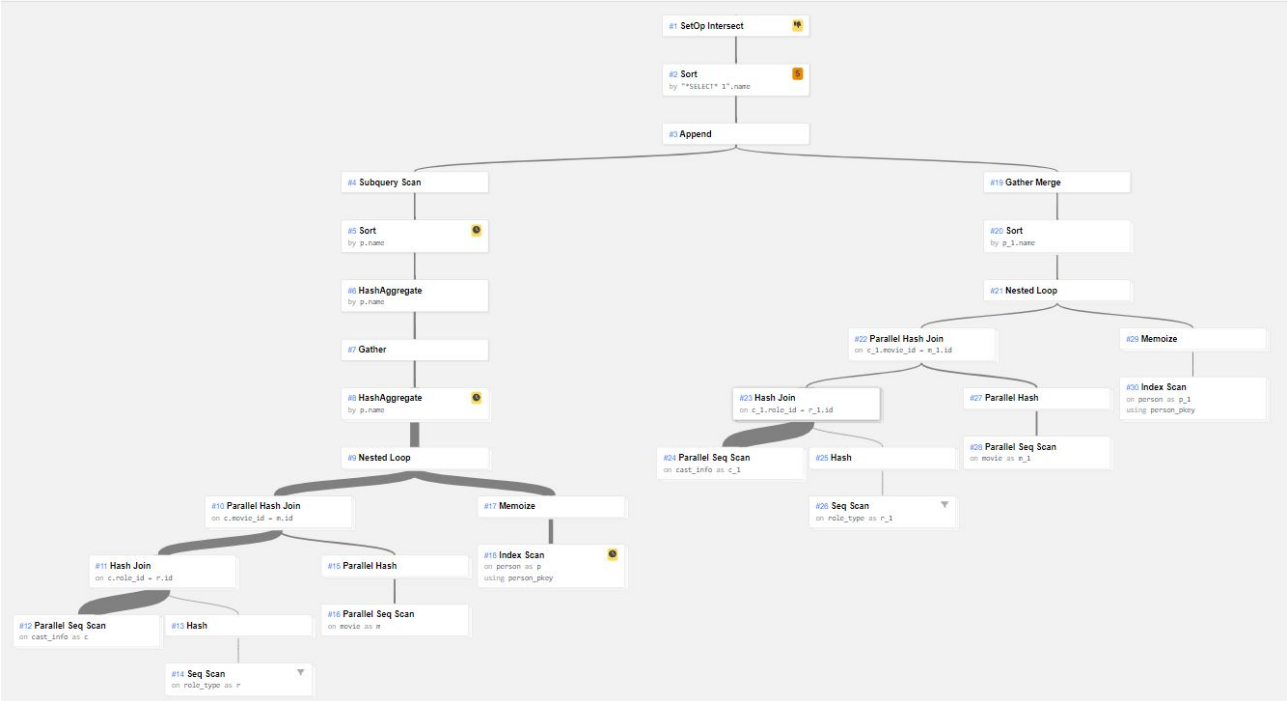


Figure 2: Index scans are prioritized after optimizations (which were afterwards joined on differently).

2.3 Query 4

```
(SELECT DISTINCT p.name from movie as m
        join cast_info as c on c.movie_id=m.id
        join person as p on p.id = c.person_id
        join role_type as r on r.id = c.role_id
                where (r.role = 'actor' or r.role = 'actress') order by p.
                    name)
Intersect
(SELECT DISTINCT p.name from movie as m
        join cast_info as c on c.movie_id=m.id
        join person as p on p.id = c.person_id
        join role_type as r on r.id = c.role_id
                where r.role = 'director' order by p.name);
```



Optimizations :

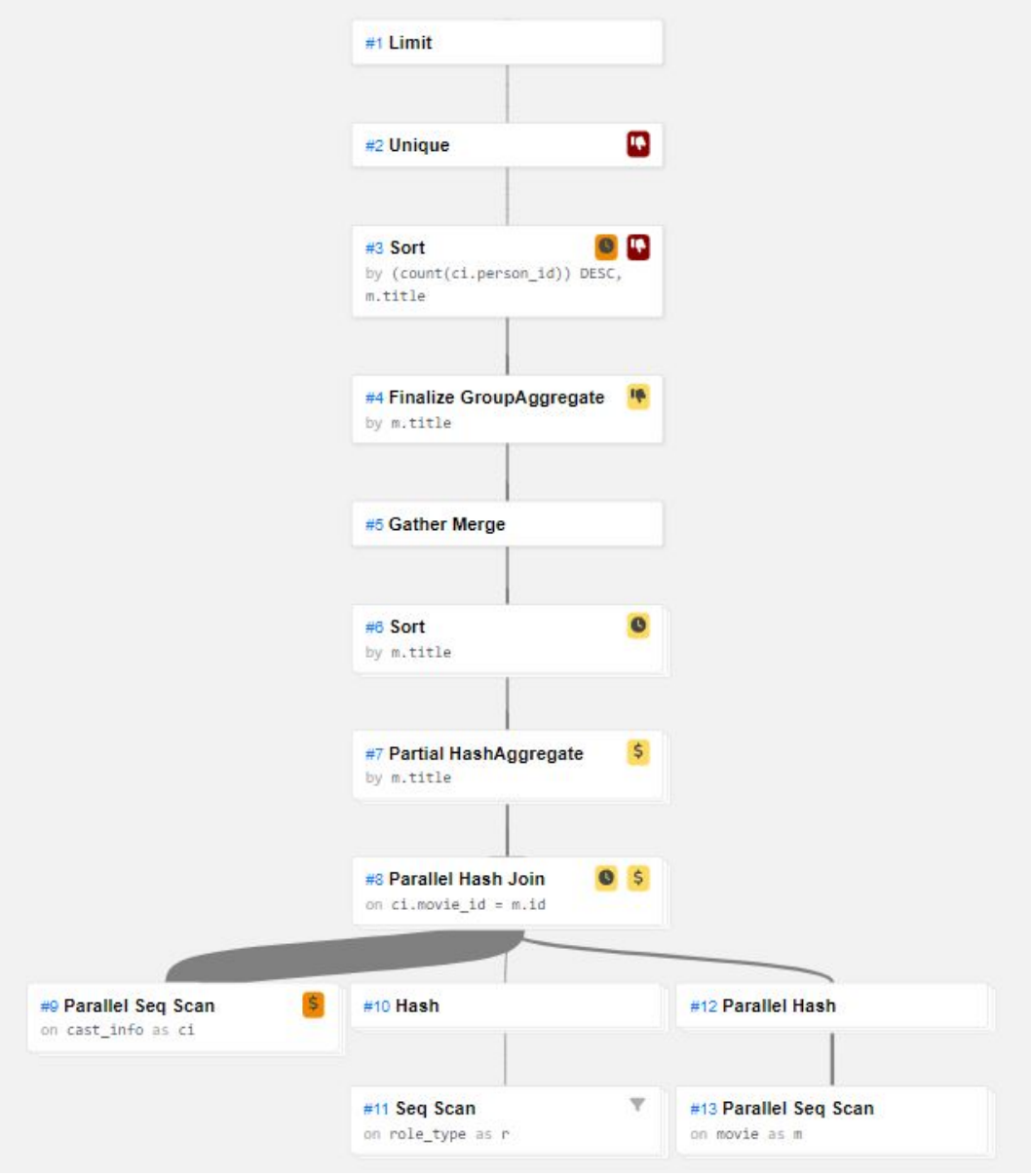
- **Query modification "or" to "in"** (cf Query 1).
- **Creating an index on role in role_type (Cost : 13.27 s) :** Since in both subqueries we select over role_type.role.

Plan after optimizations

Figure 3: The type of index scans used were modified after optimizations.

2.4 Query 6

```
SELECT DISTINCT m.title, COUNT(person_id) as d_coun from cast_info as ci
        join movie as m on m.id = ci.movie_id
        join role_type as r on r.id = ci.role_id
        where r.role = 'director' group by m.title order by d_coun DESC
           LIMIT(20)
```



Optimizations :

- **Query modification: removing "DISTINCT" from query**: since we group by the title of the movie and we project on each title, each title is necessarily unique.

- **Creating an index on title in movie (Cost : 9.59 s) :** Since we group by title at the end of the query, it is possible that an index on movie.title optimizes the query. This is indeed the case.
- **Disable sequential scan** (cf Query 1).

Plan after optimizations

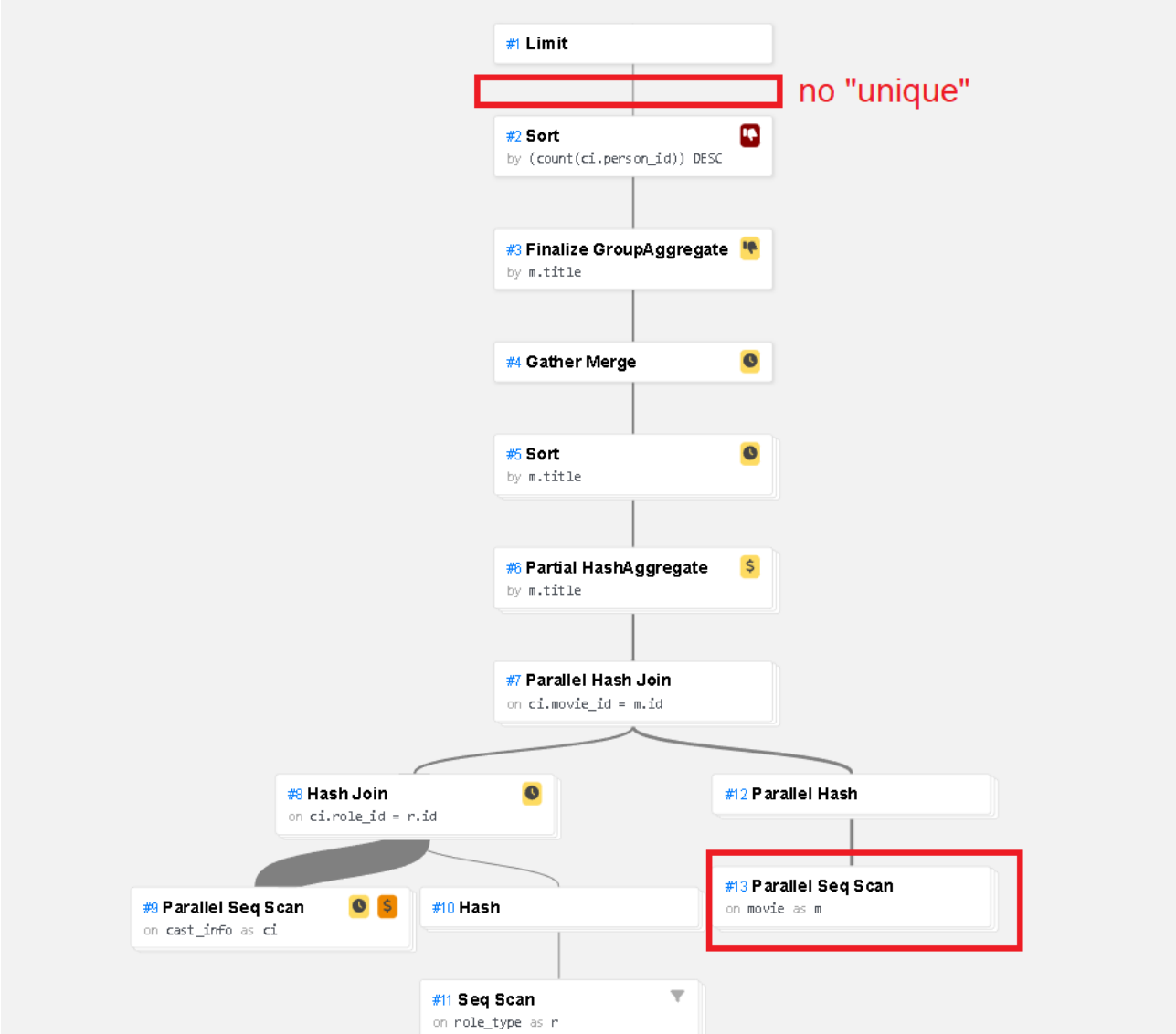


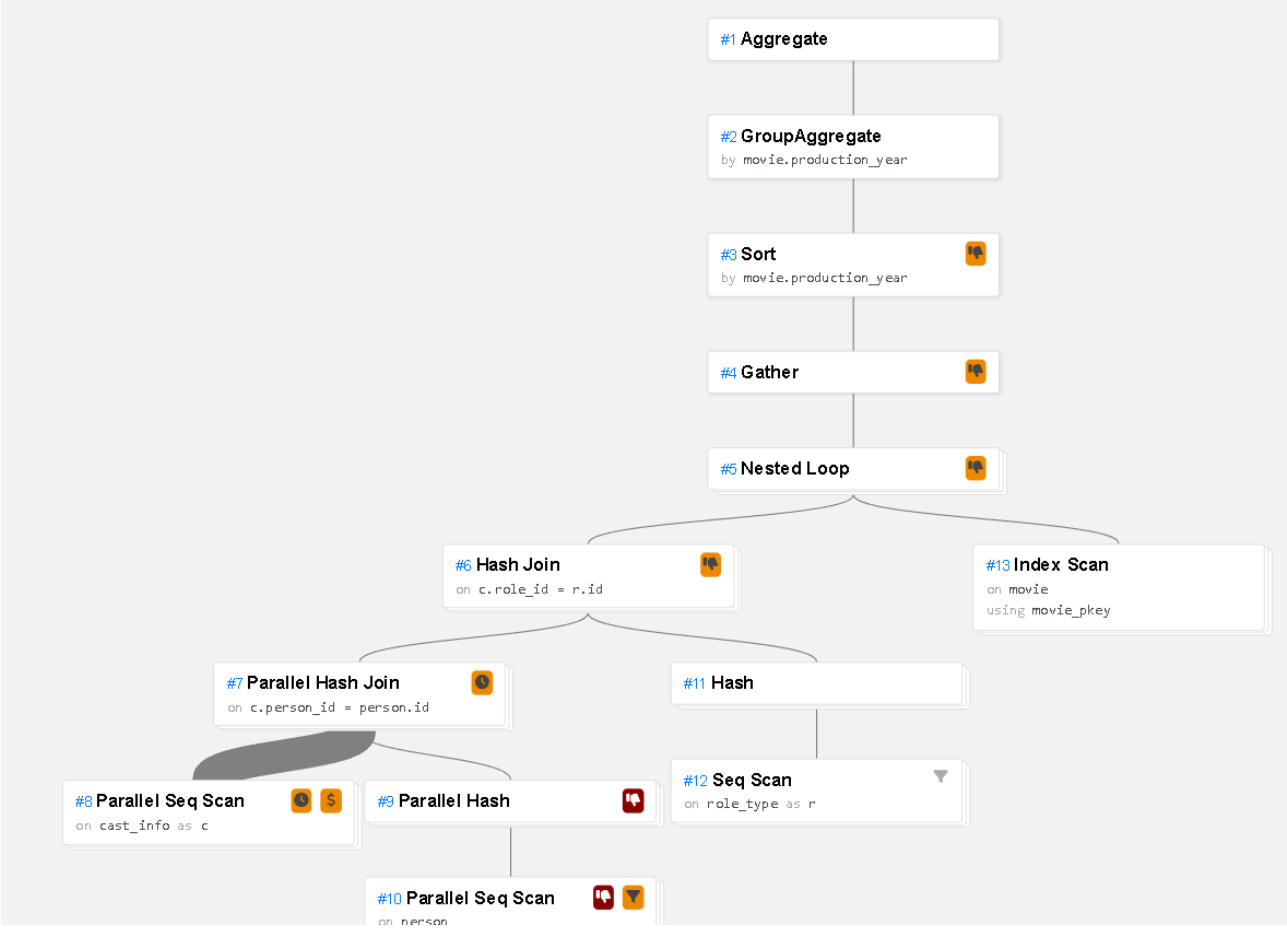
Figure 4: Index scans are prioritized over sequential scans.

2.5 Query 7

```
SELECT AVG(count) FROM

(SELECT COUNT(movie.id), movie.production_year FROM movie JOIN cast_info ON
    movie.id = cast_info.movie_id
JOIN person ON person.id = cast_info.person_id
WHERE person.name = 'Covino, Layla'
AND r.role = 'actress'
GROUP BY movie.production_year) AS count;
```

Original



Optimizations :

- **query modification: replace COUNT(movie.id) by COUNT(*)**: this avoids doing an unnecessary projection on movie.id
- **Creating an index on person.name** (cf Query 3).

Plan after optimizations

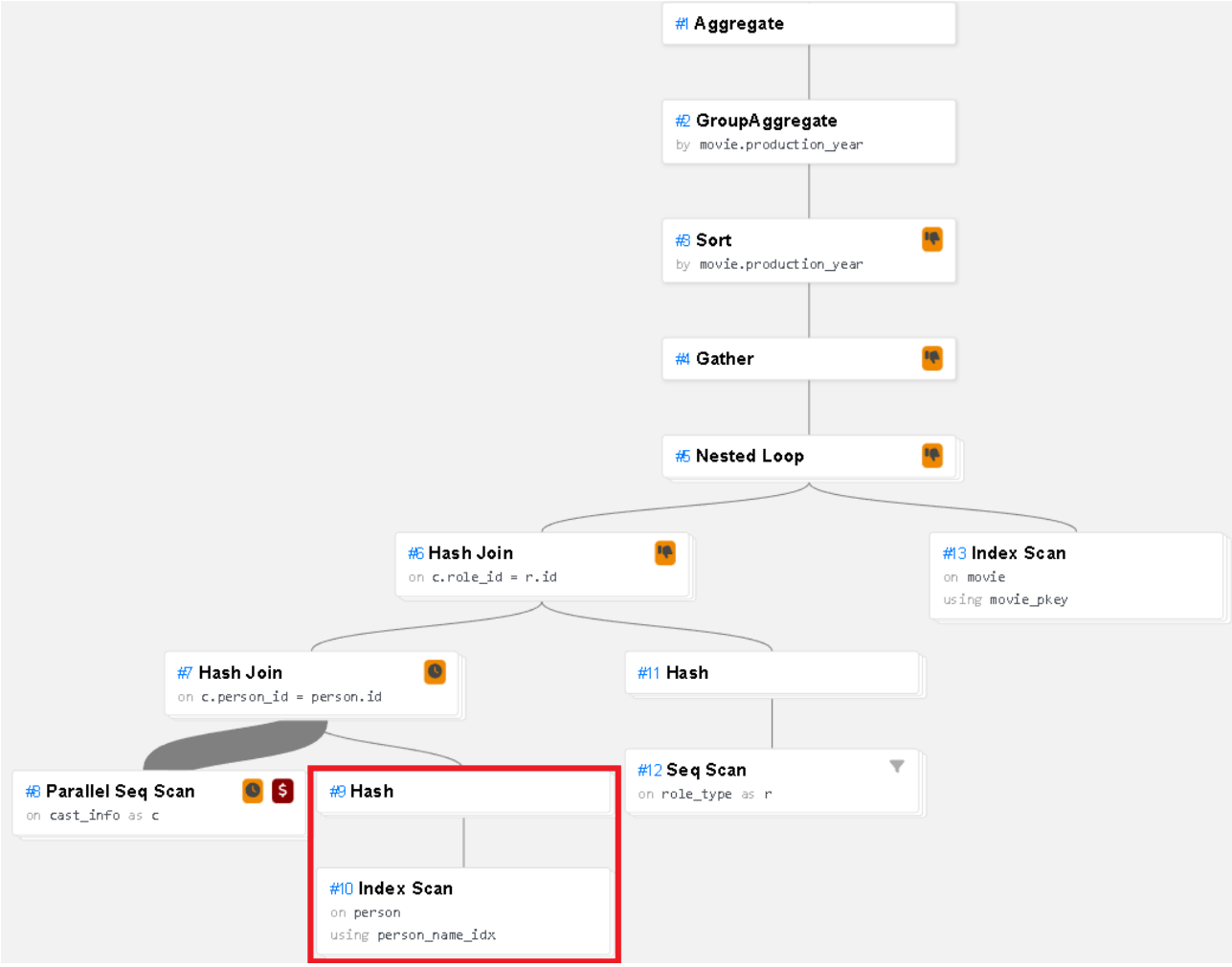


Figure 5: Index scans are prioritized after optimizations.

3 Final results

We have managed to improve all query times by at least 9%, with query 6 improving by 33%.

Query	Time (before)	Time (after)	Percentage improved
1	94.1± 1.2	81.5 ± 5.6	0.13
3	2.53 ± 0.02	2.22 ± 0.01	0.12
4	75.4 ± 2.3	68.6 ± 1.9	0.09
6	15.3 ± 0.2	10.3 ± 0.0	0.33
7	2.53 ± 0.02	2.23 ± 0.02	0.12

Table 1: Average query times: each query was executed 20 times, and the uncertainty is calculated by taking its standard deviation.

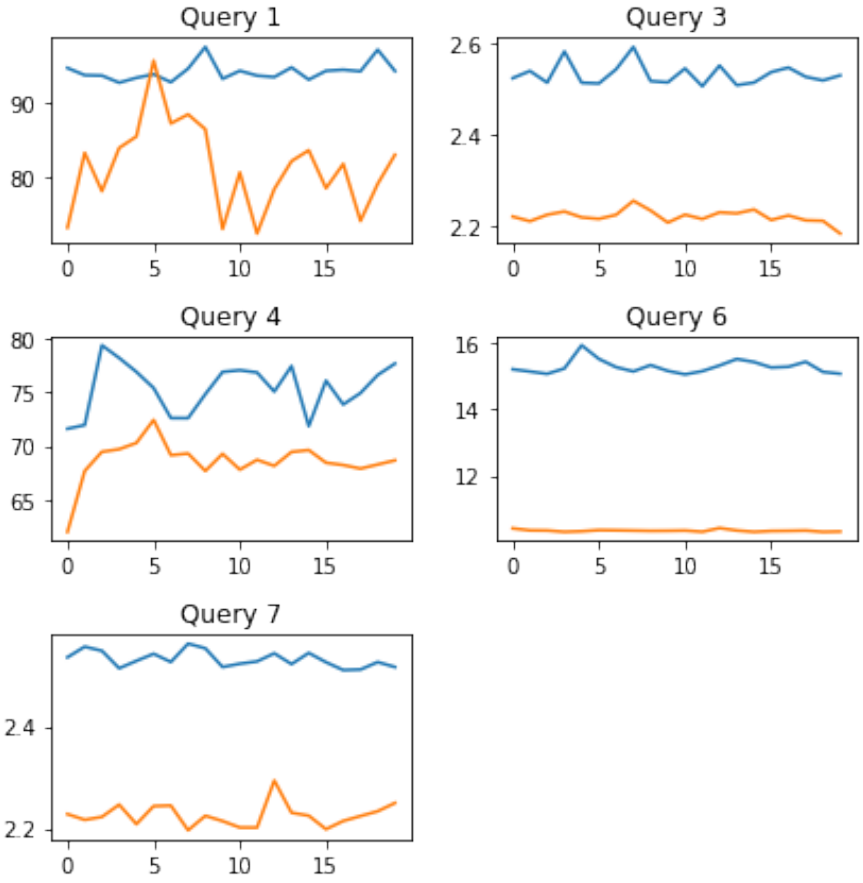


Figure 6: Query times before and after optimization. Each query was executed 20 times.
Blue : Before optimization.
Orange : After optimization.

4 Optimizations that did not work

1. Query 1: Rewriting query: "where (r.role like 'act%')": slowed query by 0.2s
2. Query 3: Rewriting order of joins to join smaller relations first: slowed query by 0.2s
3. Lowering random_page_cost from 4 to a lower value: this setting should in theory make the threshold for index scan lower and prioritize it over sequential scan more frequently, but we did not see any significant improvement for Query 3.
4. Setting the effective_cache_size = '30 GB': again, a larger value would make it more likely that index scans would be used, but we did not see any significant improvement for Query 3.
5. Adding clusters on the different indices that we proposed did not improve query times.