PARTE III: ORIENTAÇÃO A OBJETOS E ESTUDO DE PLANOS DE CONSULTAS E DESEMPENHO DE CONSULTAS - COM E SEM INDEXAÇÃO

https://www.youtube.com/watch?v=5ftmS-80FW4

Artefato A)

```
-- Table 'posters' --
/*
- Objeto complexo

 O atributo complexo metadata contém informações pertinentes da imagem e que não são as imagens em si, como dimensões, formato, qualidade, etc... Como metadata é um campo que recebe dados de uma fonte externa, não temos garantias sobre quais metadados estarão disponíveis, nem a forma como estarão organizados. Assim, foi criado um campo do tipo JSONB (JSON binário, um dos tipos complexos possíveis do PostgreSQL) para podemos coletar essas informações de forma mais flexível
*/
CREATE TABLE posters (
 path text,
 metadata jsonb NOT NULL DEFAULT '{}'::jsonb,
 obra_id integer NOT NULL
);


-- Table 'usuario_premium' --
/*
 - Herança

 Em nosso sistema, pensamos num modelo 'Freemium', onde disponibilizamos as funcionalidades essenciais para todos os usuários, e para àqueles que gostariam de contribuir com um pagamento mensal, adicionamos algumas funcionalidades extras, como indicações visuais de usuário "patrocinador" do sistema, possibilidade de foto de perfil animada, assinatura personalizada nas avaliações das obras, etc...

Para a solução no banco de dados, incluímos a tabela abaixo, herdando os atributos de usuários comuns, e adicionando um campo extra, para indicar a data em que o modo Premium expira, sendo necessário novo pagamento para reativar as funcionalidades extras. A herança nesse caso é uma boa solução, pois os usuários Premium ainda possuem os mesmos relacionamentos de usuários comuns, não sendo necessários JOINS adicionais para resgatar as relações já existentes de usuários (esses JOINS seriam necessários caso optássemos por uma relação de especialização)

*/
CREATE TABLE usuario_premium (
  data_expira_em DATE NOT NULL,
```

) INHERITS(usuario)

Artefato B)

Consulta 1

```
SELECT DISTINCT o.nome
FROM obra o
   INNER JOIN diretores d ON o.diretores_id IN (
         SELECT id
         FROM diretores
         WHERE nome_diretor = 'Karole Pickens'
   )
   INNER JOIN avaliacao a ON o.id IN (
         SELECT obra_id
         FROM avaliacao
         GROUP BY obra_id
         HAVING AVG(nota) > 8
   );
```

```
HashAggregate  (cost=25274.92..25280.02 rows=510 width=20) (actual time=60.148..60.156 rows=2 loops=1)
  Group Key: o.nome
  ->  Hash Join  (cost=545.01..20426.70 rows=1939290 width=20) (actual time=28.710..41.156 rows=61080 loops=1)
        Hash Cond: (avaliacao.obra_id = o.id)
        ->  Nested Loop  (cost=92.90..2712.60 rows=207010 width=4) (actual time=1.511..2.188 rows=840 loops=1)
              ->  Seq Scan on diretores d  (cost=0.00..22.70 rows=1270 width=0) (actual time=0.012..0.031 rows=20 loops=1)
              ->  Materialize  (cost=92.90..102.68 rows=163 width=4) (actual time=0.075..0.100 rows=42 loops=20)
                    ->  HashAggregate  (cost=92.90..100.24 rows=163 width=4) (actual time=1.471..1.883 rows=42 loops=1)
                          Group Key: avaliacao.obra_id
                          Filter: (avg(avaliacao.nota) > '8'::numeric)
                          Rows Removed by Filter: 447
                          ->  Seq Scan on avaliacao  (cost=0.00..85.27 rows=1527 width=8) (actual time=0.007..0.495 rows=1527 loops=1)
        ->  Hash  (cost=406.12..406.12 rows=3679 width=24) (actual time=27.042..27.045 rows=39702 loops=1)
              Buckets: 65536 (originally 4096)  Batches: 1 (originally 1)  Memory Usage: 2660kB
              ->  Merge Join  (cost=54.19..406.12 rows=3679 width=24) (actual time=0.527..15.304 rows=39702 loops=1)
                    Merge Cond: (diretores.id = o.diretores_id)
                    ->  Nested Loop  (cost=0.15..273.99 rows=9162 width=4) (actual time=0.035..1.176 rows=1527 loops=1)
                          ->  Index Scan using diretores_pkey on diretores  (cost=0.15..70.38 rows=6 width=4) (actual time=0.016..0.031 rows=1 loops=1)
                                Filter: (nome_diretor = 'Karole Pickens'::text)
                                Rows Removed by Filter: 19
                          ->  Materialize  (cost=0.00..92.91 rows=1527 width=0) (actual time=0.017..0.910 rows=1527 loops=1)
                                ->  Seq Scan on avaliacao a  (cost=0.00..85.27 rows=1527 width=0) (actual time=0.011..0.470 rows=1527 loops=1)
                    ->  Sort  (cost=54.04..55.31 rows=510 width=28) (actual time=0.486..3.562 rows=39703 loops=1)
                          Sort Key: o.diretores_id
                          Sort Method: quicksort  Memory: 64kB
                          ->  Seq Scan on obra o  (cost=0.00..31.10 rows=510 width=28) (actual time=0.008..0.243 rows=510 loops=1)
Planning Time: 0.779 ms
Execution Time: 60.995 ms
```

Dado que fizemos muito mais inserções de avaliações do que inserções de diretores (o que é bem plausível para uma situação real), esperávamos que o maior custo da consulta seria durante o inner join na tabela de avaliação.

Essa suposição é evidenciada no plano de consulta gerado, onde a maior parte do custo está no primeiro **Hash Join.** Parte deste custo também é explicada pela necessidade

de agregarmos as tuplas realizando uma operação de média simultâneamente, para então ordená-las para fazermos a comparação da cláusula **HAVING**

Consulta 2

```
SELECT o.nome, COUNT(*) AS qtde_plataformas
FROM filmes f
   INNER JOIN obra o ON f.obra_id = o.id
   INNER JOIN disponibilidade_sites ds ON o.id = ds.obra_id
WHERE o.id IN (
   SELECT obra_id
   FROM esta_em ee
   WHERE ator_id IN (
         SELECT id
         FROM ator a
         WHERE a.nome_ator = 'Kevin Bacon'
   )
)
GROUP BY o.nome
ORDER BY COUNT(*) DESC
LIMIT 5
OFFSET 0;
```

```
Limit  (cost=71.11..71.13 rows=5 width=28) (actual time=0.057..0.059 rows=0 loops=1)
  -> Sort  (cost=71.11..71.29 rows=70 width=28) (actual time=0.057..0.058 rows=0 loops=1)
     Sort Key: (count(*)) DESC
     Sort Method: quicksort  Memory: 25kB
     -> HashAggregate  (cost=69.25..69.95 rows=70 width=28) (actual time=0.051..0.052 rows=0 loops=1)
        Group Key: o.nome
        -> Hash Join  (cost=46.45..68.90 rows=70 width=20) (actual time=0.050..0.051 rows=0 loops=1)
           Hash Cond: (ds.obra_id = f.obra_id)
           -> Seq Scan on disponibilidade_sites ds  (cost=0.00..17.45 rows=1145 width=4) (actual time=0.009..0.009 rows=1 loops=1)
           -> Hash  (cost=46.06..46.06 rows=31 width=32) (actual time=0.028..0.030 rows=0 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 8kB
              -> Hash Join  (cost=35.89..46.06 rows=31 width=32) (actual time=0.028..0.029 rows=0 loops=1)
                 Hash Cond: (f.obra_id = o.id)
                 -> Seq Scan on filmes f  (cost=0.00..8.00 rows=500 width=4) (actual time=0.005..0.005 rows=1 loops=1)
                 -> Hash  (cost=35.50..35.50 rows=31 width=28) (actual time=0.014..0.014 rows=0 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 8kB
                    -> Nested Loop  (cost=19.57..35.50 rows=31 width=28) (actual time=0.013..0.014 rows=0 loops=1)
                       -> HashAggregate  (cost=19.30..19.61 rows=31 width=4) (actual time=0.013..0.013 rows=0 loops=1)
                          Group Key: ee.obra_id
                          -> Nested Loop  (cost=4.52..19.22 rows=31 width=4) (actual time=0.012..0.012 rows=0 loops=1)
                             -> Seq Scan on ator a  (cost=0.00..2.00 rows=1 width=4) (actual time=0.012..0.012 rows=0 loops=1)
                                Filter: (nome_ator = 'Kevin Bacon'::text)
                                Rows Removed by Filter: 80
                             -> Bitmap Heap Scan on esta_em ee  (cost=4.52..16.91 rows=31 width=8) (never executed)
                                Recheck Cond: (ator_id = a.id)
                                -> Bitmap Index Scan on esta_em_pkey  (cost=0.00..4.51 rows=31 width=0) (never executed)
                                   Index Cond: (ator_id = a.id)
                       -> Index Scan using obra_pkey on obra o  (cost=0.27..0.53 rows=1 width=24) (never executed)
                          Index Cond: (id = ee.obra_id)
Planning Time: 0.857 ms
Execution Time: 0.164 ms
```

Considerando que fizemos muito mais inserções de filmes do que inserções de atores, esperávamos que o maior custo da consulta seria durante o inner join na tabela de filmes. Podemos observar que essa suposição é verdadeira ao analisar o plano de consulta gerado, onde a grande parte do custo está no primeiro Hash Join que obteve um custo=46.4.46.90. Além disso, como o resultado precisa estar ordenado, o custo total da consulta se eleva também, nesse caso o SGBD decidiu utilizar o algoritmo QuickSort para realizar a ordenação e isso custou 25 Kb.

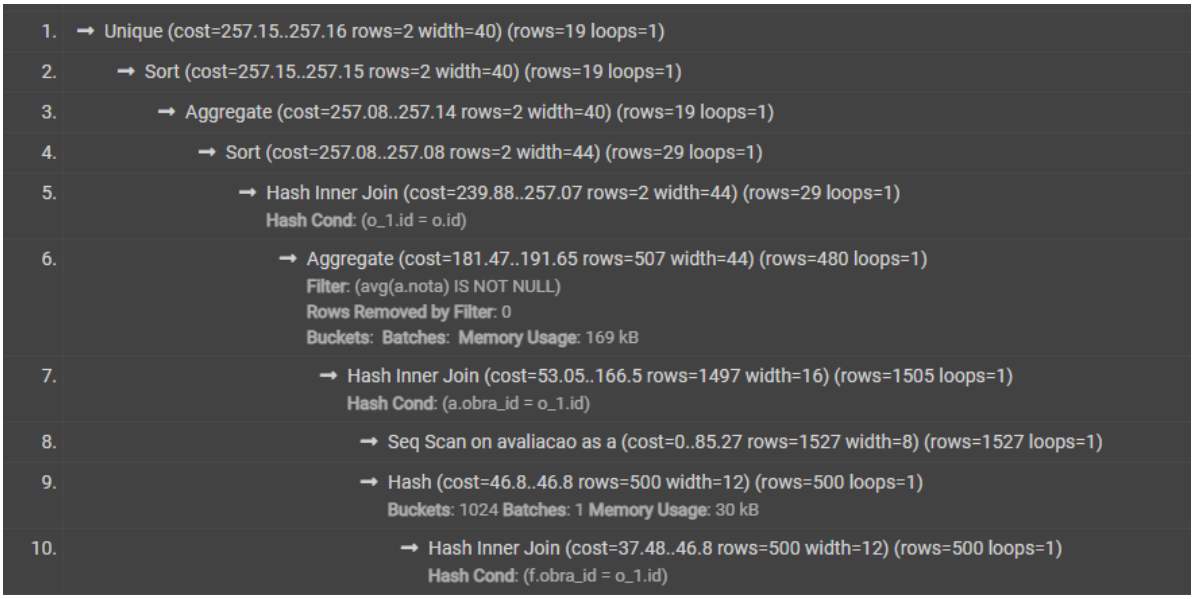
Consulta 3

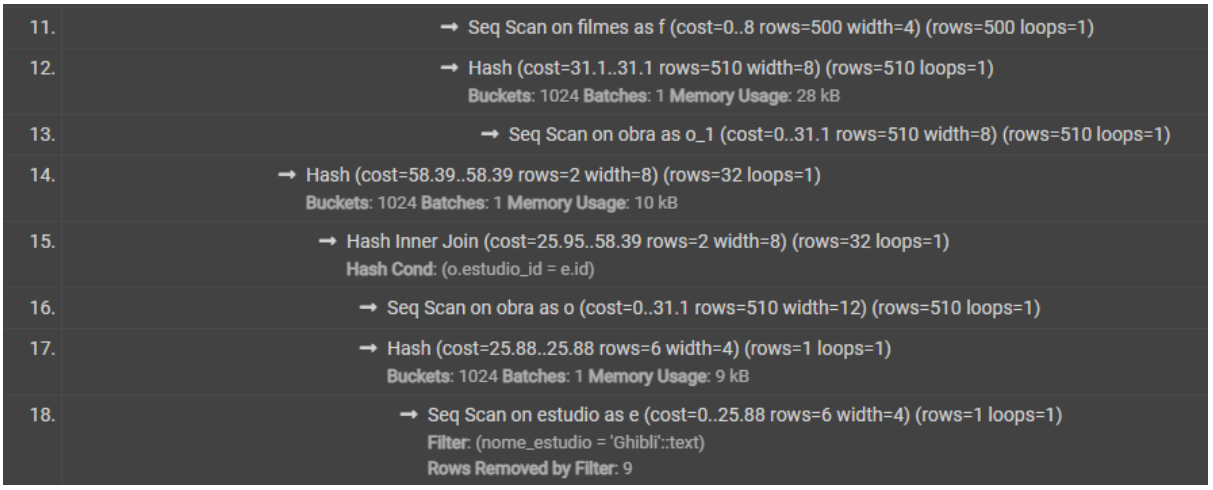
```
WITH t AS (
   SELECT g.tipo_de_genero, gob.obra_id
   FROM genero_obra gob
         INNER JOIN genero g ON g.id = gob.genero_id
)

SELECT o.nome, s.n_episodios, s.temporadas, (s.n_episodios * s.temporadas) AS
total_episodios FROM series s
   INNER JOIN obra o ON o.id = s.obra_id
WHERE (s.n_episodios * s.temporadas) < 50
AND o.id IN (
   SELECT t.obra_id
   FROM t
   WHERE t.tipo_de_genero = 'Comédia'
)
AND o.id NOT IN (
   SELECT t.obra_id
   FROM t
   WHERE t.tipo_de_genero = 'Romance'
);
```

```
Hash Join  (cost=126.22..169.40 rows=3 width=32) (actual time=0.584..0.587 rows=0 loops=1)
  Hash Cond: (s.obra_id = o.id)
  CTE t
    ->  Hash Join  (cost=38.58..53.65 rows=876 width=36) (actual time=0.034..0.298 rows=876 loops=1)
          Hash Cond: (gob.genero_id = g.id)
          ->  Seq Scan on genero_obra gob  (cost=0.00..12.76 rows=876 width=8) (actual time=0.006..0.075 rows=876 loops=1)
          ->  Hash  (cost=22.70..22.70 rows=1270 width=36) (actual time=0.012..0.012 rows=6 loops=1)
                Buckets: 2048  Batches: 1  Memory Usage: 17kB
                ->  Seq Scan on genero g  (cost=0.00..22.70 rows=1270 width=36) (actual time=0.003..0.003 rows=6 loops=1)
  ->  Seq Scan on series s  (cost=0.00..40.60 rows=680 width=12) (actual time=0.014..0.014 rows=1 loops=1)
        Filter: ((n_episodios * temporadas) < 50)
        Rows Removed by Filter: 3
  ->  Hash  (cost=72.55..72.55 rows=2 width=28) (actual time=0.556..0.557 rows=0 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 8kB
        ->  Hash Semi Join  (cost=39.48..72.55 rows=2 width=28) (actual time=0.556..0.557 rows=0 loops=1)
              Hash Cond: (o.id = t.obra_id)
              ->  Seq Scan on obra o  (cost=19.72..52.09 rows=255 width=24) (never executed)
                    Filter: (NOT (hashed SubPlan 2))
                    SubPlan 2
                      ->  CTE Scan on t t_1  (cost=0.00..19.71 rows=4 width=4) (never executed)
                            Filter: (tipo_de_genero = 'Romance'::text)
              ->  Hash  (cost=19.71..19.71 rows=4 width=4) (actual time=0.546..0.547 rows=0 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 8kB
                    ->  CTE Scan on t  (cost=0.00..19.71 rows=4 width=4) (actual time=0.546..0.546 rows=0 loops=1)
                          Filter: (tipo_de_genero = 'Comédia'::text)
                          Rows Removed by Filter: 876
Planning Time: 0.292 ms
```

Consulta 4

```sql
WITH medias AS (
    SELECT o.id AS id, AVG(a.nota) AS media,
    date_part('year', o.data_lancamento) AS ano
    FROM obra o
        INNER JOIN filmes f ON o.id = f.obra_id
        INNER JOIN avaliacao a ON o.id = a.obra_id
    GROUP BY o.id, ano
)

SELECT DISTINCT ano, avg(media) FROM
(SELECT date_part('year', o.data_lancamento) AS ano, m.media, m.id
FROM estudio e
    INNER JOIN obra o ON e.id = o.estudio_id
    LEFT JOIN medias m ON o.id = m.id
WHERE m.media NOTNULL
    AND e.nome_estudio = 'Ghibli'
ORDER BY date_part('year', o.data_lancamento)) as n
GROUP BY ano
```





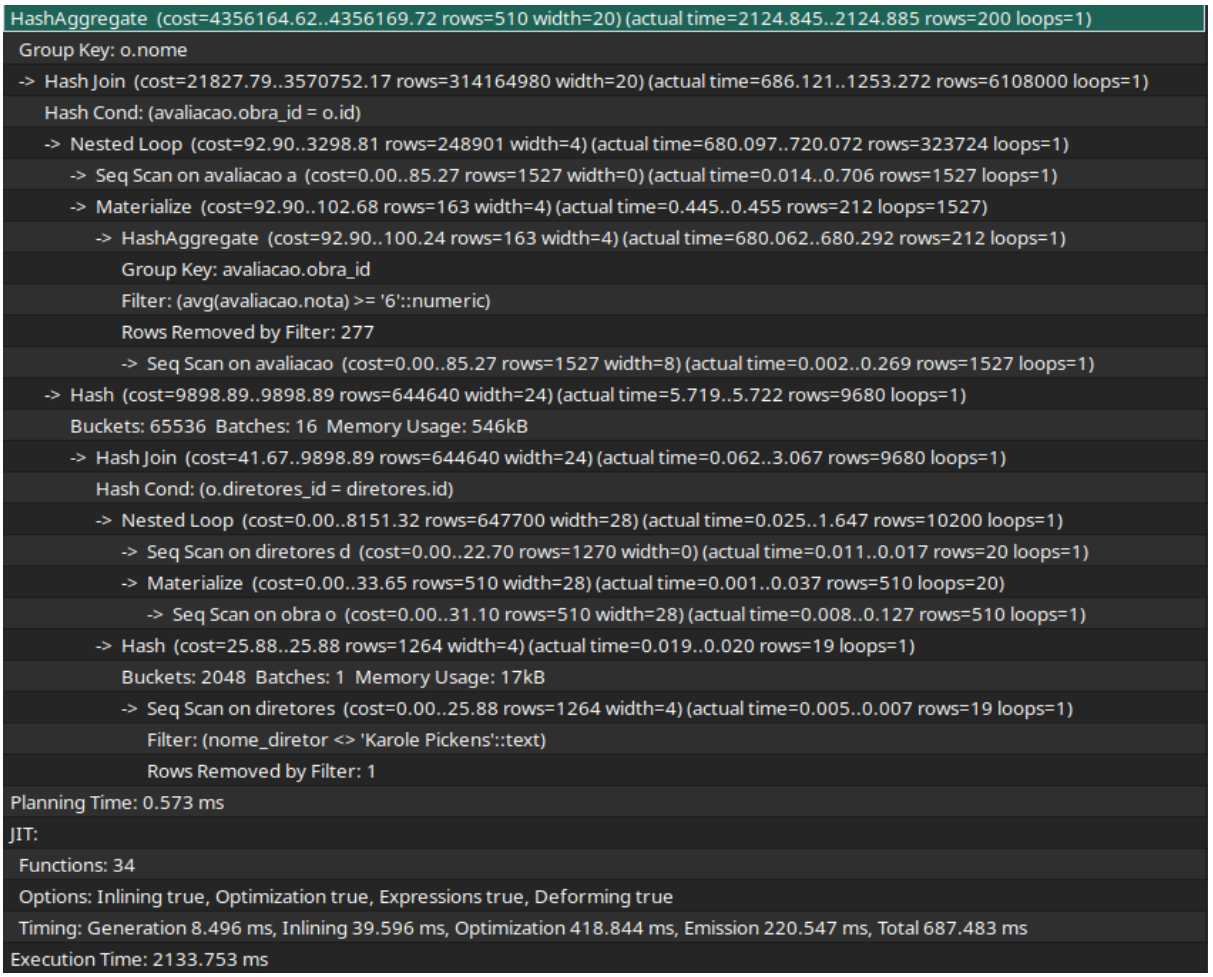
Artefato C)

Consulta 1

Modificação 1

As duas modificações feitas foram alterar o filtro do WHERE de '=' para '!=' e diminuir a nota mínima de 8 para 6, adicionando a condição de igualdade também

```
SELECT DISTINCT o.nome
FROM obra o
	INNER JOIN diretores d ON o.diretores_id IN (
	SELECT id
	FROM diretores
	WHERE nome_diretor != 'Karole Pickens'
)
INNER JOIN avaliacao a ON o.id IN (
	SELECT obra_id
	FROM avaliacao
	GROUP BY obra_id
	HAVING AVG(nota) >= 6
);
```



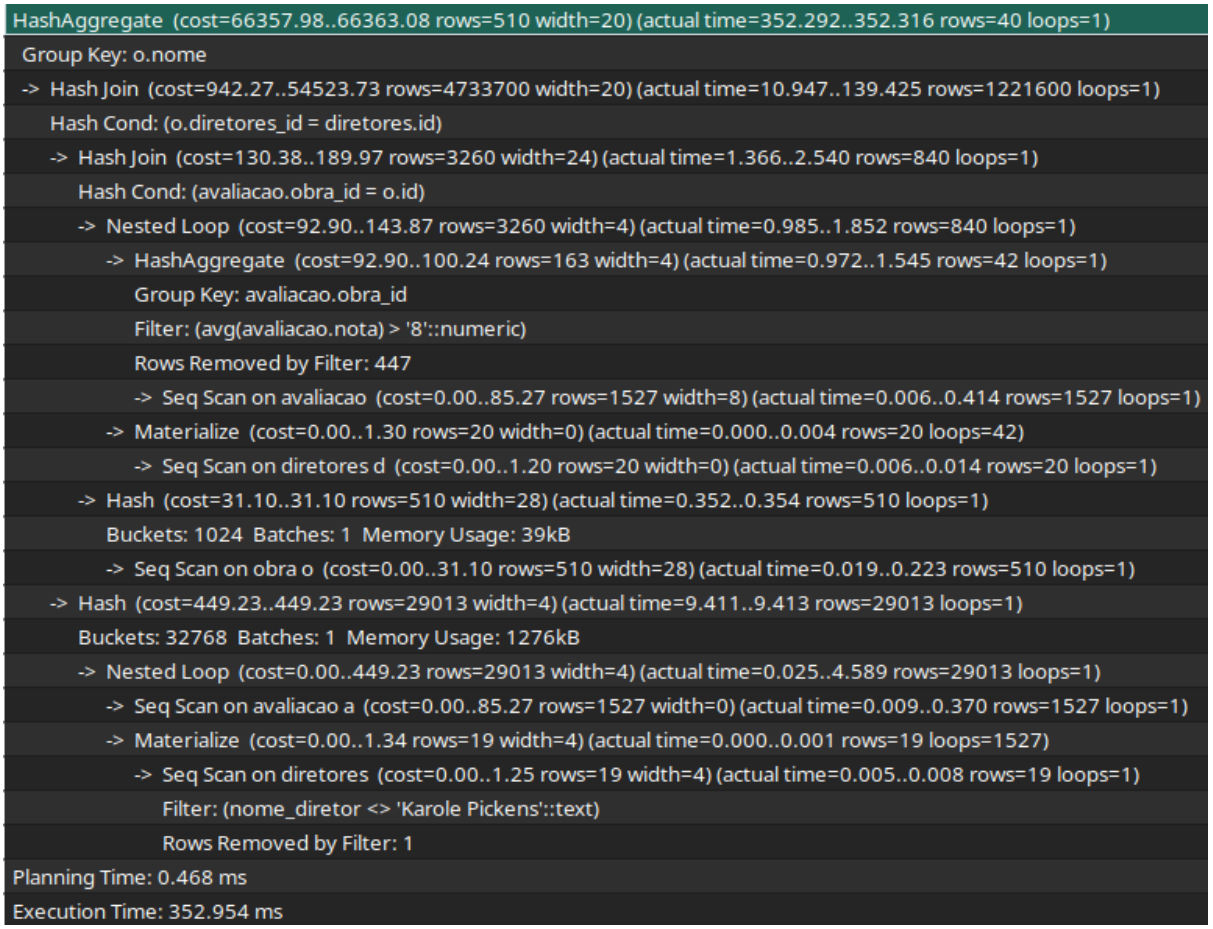
Comparando as duas consultas, nota-se que não há uma diferença significativa no custo das funções de **Materialize** e **HashAggregate** utilizadas para o filtro WHERE na tabela **avaliacao**.

Por outro lado, como a condição do WHERE no nome do diretor é claramente mais vaga do que a condição de igualdade que tínhamos anteriormente, há um aumento no tempo de execução da consulta. Pelo plano de consulta, podemos observar que o SGBD realiza o filtro apenas na última operação, sendo necessário iterar em mais tuplas para obter os valores desejados.

Modificação 2

Para a segunda modificação nessa consulta, foi mantida a primeira modificação e foi adicionado um **index** na coluna **nome_diretor** da tabela de diretores

```
CREATE UNIQUE INDEX nome_diretor_idx ON diretores (nome_diretor);
```



Nota-se um aumento significativo de performance na segunda parte da consulta ("Hash (cost=449.23…"), ao custo de um consumo maior de memória, característica esperada ao se criar um índice. Comparativamente, temos uma redução de quase 2 segundos no tempo de execução em relação à consulta anterior, ao custo de um consumo de memória 250% maior.

Consulta 2

Modificação 1

A modificação foi feita no filtro WHERE de **=** para **!= e** aumentando o limite de retorno da query de **5** para 1**00 tuplas.**

```
SELECT o.nome, COUNT(*) AS qtde_plataformas
FROM filmes f
   INNER JOIN obra o ON f.obra_id = o.id
   INNER JOIN disponibilidade_sites ds ON o.id = ds.obra_id
WHERE o.id IN (
   SELECT obra_id
   FROM esta_em ee
   WHERE ator_id IN (
      SELECT id
      FROM ator a
      WHERE a.nome_ator != 'Kevin Bacon'
```

```
    )
)
GROUP BY o.nome
ORDER BY COUNT(*) DESC
LIMIT 100
OFFSET 0;
```

	QUERY PLAN text
1	Limit (cost=189.08..189.33 rows=100 width=28) (actual time=3.444..3.460 rows=100 loops=1)
2	-> Sort (cost=189.08..190.35 rows=510 width=28) (actual time=3.442..3.450 rows=100 loops=1)
3	Sort Key: (count(*)) DESC
4	Sort Method: top-N heapsort Memory: 37kB
5	-> HashAggregate (cost=164.49..169.59 rows=510 width=28) (actual time=3.284..3.345 rows=473 loops=1)
6	Group Key: o.nome
7	-> Hash Join (cost=122.87..158.87 rows=1123 width=20) (actual time=2.229..2.922 rows=1116 loops=1)
8	Hash Cond: (o.id = f.obra_id)
9	-> Hash Join (cost=37.48..57.95 rows=1145 width=28) (actual time=0.385..0.820 rows=1145 loops=1)
10	Hash Cond: (ds.obra_id = o.id)
11	-> Seq Scan on disponibilidade_sites ds (cost=0.00..17.45 rows=1145 width=4) (actual time=0.018....
12	-> Hash (cost=31.10..31.10 rows=510 width=24) (actual time=0.354..0.354 rows=510 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 37kB
14	-> Seq Scan on obra o (cost=0.00..31.10 rows=510 width=24) (actual time=0.008..0.192 rows=5...
15	-> Hash (cost=79.15..79.15 rows=500 width=8) (actual time=1.824..1.826 rows=496 loops=1)
16	Buckets: 1024 Batches: 1 Memory Usage: 28kB
17	-> Hash Join (cost=64.27..79.15 rows=500 width=8) (actual time=1.558..1.741 rows=496 loops=1)
18	Hash Cond: (f.obra_id = ee.obra_id)
19	-> Seq Scan on filmes f (cost=0.00..8.00 rows=500 width=4) (actual time=0.008..0.046 rows=50...
20	-> Hash (cost=57.95..57.95 rows=506 width=4) (actual time=1.540..1.541 rows=506 loops=1)
21	Buckets: 1024 Batches: 1 Memory Usage: 26kB
22	-> HashAggregate (cost=52.89..57.95 rows=506 width=4) (actual time=1.389..1.452 rows=5...
23	Group Key: ee.obra_id
24	-> Hash Join (cost=2.99..46.74 rows=2457 width=4) (actual time=0.046..0.869 rows=248...
25	Hash Cond: (ee.ator_id = a.id)
26	-> Seq Scan on esta_em ee (cost=0.00..36.88 rows=2488 width=8) (actual time=0.006...
27	-> Hash (cost=2.00..2.00 rows=79 width=4) (actual time=0.033..0.033 rows=80 loops...
28	Buckets: 1024 Batches: 1 Memory Usage: 11kB
29	-> Seq Scan on ator a (cost=0.00..2.00 rows=79 width=4) (actual time=0.005..0.01...
30	Filter: (nome_ator <> 'Kevin Bacon'::text)
31	Planning Time: 1.422 ms
32	Execution Time: 3.750 ms

Comparando as duas consultas é possível observar que o tempo de execução aumentou consideravelmente. Sem as alterações o tempo total (planning time + execution time) foi de apenas 1,455 ms, já a segunda consulta levou um total de 5,172ms, um aumento de quase 30%.

Como a condição do WHERE no nome do diretor é claramente mais vaga do que a condição de igualdade que tínhamos anteriormente, há um aumento no tempo de execução da consulta.

Modificação 2

Para a segunda modificação nessa consulta, foi mantida a primeira modificação e foi adicionado um **index** na coluna **nome_ator** da tabela de ator:

```
CREATE UNIQUE INDEX nome_ator_idx ON ator (nome_ator);
```

```
1  Limit  (cost=189.08..189.33 rows=100 width=28) (actual time=3.213..3.229 rows=100 loops=1)
2  -> Sort  (cost=189.08..190.35 rows=510 width=28) (actual time=3.212..3.221 rows=100 loops=1)
3  Sort Key: (count(*)) DESC
4  Sort Method: top-N heapsort  Memory: 37kB
5  -> HashAggregate  (cost=164.49..169.59 rows=510 width=28) (actual time=3.031..3.126 rows=473 loops=1)
6  Group Key: o.nome
7  -> Hash Join  (cost=122.87..158.87 rows=1123 width=20) (actual time=2.008..2.698 rows=1116 loops=1)
8  Hash Cond: (o.id = f.obra_id)
9  -> Hash Join  (cost=37.48..57.95 rows=1145 width=28) (actual time=0.286..0.718 rows=1145 loops=1)
10 Hash Cond: (ds.obra_id = o.id)
11 -> Seq Scan on disponibilidade_sites ds  (cost=0.00..17.45 rows=1145 width=4) (actual time=0.011....
12 -> Hash  (cost=31.10..31.10 rows=510 width=24) (actual time=0.266..0.266 rows=510 loops=1)
13 Buckets: 1024  Batches: 1  Memory Usage: 37kB
14 -> Seq Scan on obra o  (cost=0.00..31.10 rows=510 width=24) (actual time=0.004..0.127 rows=5...
15 -> Hash  (cost=79.15..79.15 rows=500 width=8) (actual time=1.703..1.705 rows=496 loops=1)
16 Buckets: 1024  Batches: 1  Memory Usage: 28kB
17 -> Hash Join  (cost=64.27..79.15 rows=500 width=8) (actual time=1.470..1.629 rows=496 loops=1)
18 Hash Cond: (f.obra_id = ee.obra_id)
19 -> Seq Scan on filmes f  (cost=0.00..8.00 rows=500 width=4) (actual time=0.007..0.041 rows=50...
20 -> Hash  (cost=57.95..57.95 rows=506 width=4) (actual time=1.454..1.455 rows=506 loops=1)
21 Buckets: 1024  Batches: 1  Memory Usage: 26kB
22 -> HashAggregate  (cost=52.89..57.95 rows=506 width=4) (actual time=1.324..1.378 rows=5...
23 Group Key: ee.obra_id
24 -> Hash Join  (cost=2.99..46.74 rows=2457 width=4) (actual time=0.043..0.803 rows=248...
25 Hash Cond: (ee.ator_id = a.id)
26 -> Seq Scan on esta_em ee  (cost=0.00..36.88 rows=2488 width=8) (actual time=0.005...
27 -> Hash  (cost=2.00..2.00 rows=79 width=4) (actual time=0.031..0.032 rows=80 loops...
28 Buckets: 1024  Batches: 1  Memory Usage: 11kB
29 -> Seq Scan on ator a  (cost=0.00..2.00 rows=79 width=4) (actual time=0.004..0.01...
30 Filter: (nome_ator <> 'Kevin Bacon'::text)
31 Planning Time: 0.893 ms
32 Execution Time: 3.446 ms
```

Já nessa modificação, a criação de um index não realizou uma melhora de performance expressiva para a consulta.

Consulta 3

Modificação 1

Aqui, ao invés de fazermos o IN em (no caso real, os escolhidos foram “adventure” e “horror”) nas listas de gêneros, nós utilizamos a cláusula EXCEPT (equivalente a MINUS) entre essas duas listas e utilizamos um IN para ver se a série possui menos de 50 episódios. Neste caso, imaginamos como seria caso a identificação de gênero fosse obtida por fora:

```
1  SELECT obra.nome, series.n_episodios, series.temporadas
2  FROM (obra INNER JOIN series on obra.id = series.obra_id) WHERE
3  id IN
4      (SELECT obra_id FROM series WHERE (n_episodios * temporadas) < 50)
5  AND id IN
6      ((SELECT obra_id FROM genero_obra WHERE genero_id = 3)
7       EXCEPT
8       (SELECT obra_id FROM genero_obra WHERE genero_id = 6))
9  ;
10
```

```
1.  → Hash Inner Join (rows=2 loops=1)
      Hash Cond: (series.obra_id = obra.id)
2.    → Seq Scan on series as series (rows=10 loops=1)
3.    → Hash (rows=2 loops=1)
        Buckets: 1024 Batches: 1 Memory Usage: 9 kB
4.      → Hash Inner Join (rows=2 loops=1)
          Hash Cond: (obra.id = series_1.obra_id)
5.        → Hash Inner Join (rows=99 loops=1)
            Hash Cond: (obra.id = "ANY_subquery".obra_id)
6.          → Seq Scan on obra as obra (rows=510 loops=1)
7.          → Hash (rows=99 loops=1)
              Buckets: 1024 Batches: 1 Memory Usage: 12 kB
8.            → Subquery Scan (rows=99 loops=1)
9.              → Hash Except (rows=99 loops=1)
10.               → Append (rows=269 loops=1)
```

```
11.                 → Subquery Scan (rows=134 loops=1)
12.                   → Bitmap Heap Scan on genero_obra as genero_obra (rows=134 loops=1)
                        Recheck Cond: (genero_id = 3)
                        Heap Blocks: exact=4
13.                     → Bitmap Index Scan using genero_obra_pkey (rows=134 loops=1)
                          Index Cond: (genero_id = 3)
14.                 → Subquery Scan (rows=135 loops=1)
15.                   → Bitmap Heap Scan on genero_obra as genero_obra_1 (rows=135 loops=1)
                        Recheck Cond: (genero_id = 6)
                        Heap Blocks: exact=4
16.                     → Bitmap Index Scan using genero_obra_pkey (rows=135 loops=1)
                          Index Cond: (genero_id = 6)
17.           → Hash (rows=4 loops=1)
                Buckets: 1024 Batches: 1 Memory Usage: 9 kB
18.             → Aggregate (rows=4 loops=1)
                  Buckets:  Batches:  Memory Usage: 40 kB
19.               → Seq Scan on series as series_1 (rows=4 loops=1)
                    Filter: ((n_episodios * temporadas) < 50)
                    Rows Removed by Filter: 6
```

Modificação 2

É consideravelmente similar à modificação 1, porém, agora utilizando-se de EXISTS e não sabemos o índice do gênero por fora.

```sql
SELECT obra.nome, series.n_episodios, series.temporadas
FROM (obra INNER JOIN series on obra.id = series.obra_id)
WHERE
EXISTS(
    SELECT obra_id
    FROM series
    WHERE (n_episodios * temporadas) < 50
    AND obra.id = series.obra_id
)
AND
EXISTS(
    SELECT DISTINCT obra_id
    FROM genero_obra
    INNER JOIN genero
    ON genero.id = genero_obra.genero_id
    WHERE genero.tipo_de_genero = 'adventure' AND genero.tipo_de_genero != 'horror'
    AND obra.id = genero_obra.obra_id

)
;
```

```
1.  → Hash Inner Join (cost=91.59..129.72 rows=8 width=28) (rows=2 loops=1)
      Hash Cond: (series.obra_id = obra.id)
2.    → Seq Scan on series as series (cost=0..30.4 rows=2040 width=12) (rows=10 loops=1)
3.    → Hash (cost=91.57..91.57 rows=2 width=32) (rows=2 loops=1)
        Buckets: 1024 Batches: 1 Memory Usage: 9 kB
4.      → Hash Inner Join (cost=88.84..91.57 rows=2 width=32) (rows=2 loops=1)
          Hash Cond: (series_1.obra_id = obra.id)
5.        → Aggregate (cost=42.3..44.27 rows=197 width=4) (rows=4 loops=1)
            Buckets:  Batches:  Memory Usage: 40 kB
6.          → Seq Scan on series as series_1 (cost=0..40.6 rows=680 width=4) (rows=4 loops=1)
              Filter: ((n_episodios * temporadas) < 50)
              Rows Removed by Filter: 6
7.        → Hash (cost=46.49..46.49 rows=4 width=28) (rows=134 loops=1)
            Buckets: 1024 Batches: 1 Memory Usage: 17 kB
8.          → Nested Loop Inner Join (cost=44.48..46.49 rows=4 width=28) (rows=134 loops=1)
9.            → Aggregate (cost=44.21..44.25 rows=4 width=4) (rows=134 loops=1)
                Buckets:  Batches:  Memory Usage: 48 kB
10.             → Hash Inner Join (cost=29.13..44.2 rows=4 width=4) (rows=134 loops=1)
                  Hash Cond: (genero_obra.genero_id = genero.id)
11.               → Seq Scan on genero_obra as genero_obra (cost=0..12.76 rows=876 width=8) (rows=87...
12.               → Hash (cost=29.05..29.05 rows=6 width=4) (rows=1 loops=1)
                    Buckets: 1024 Batches: 1 Memory Usage: 9 kB
13.                 → Seq Scan on genero as genero (cost=0..29.05 rows=6 width=4) (rows=1 loops=1)
                      Filter: ((tipo_de_genero <> 'horror'::text) AND (tipo_de_genero = 'adventure'::text))
                      Rows Removed by Filter: 5
14.           → Index Scan using obra_pkey on obra as obra (cost=0.27..0.56 rows=1 width=24) (rows=1 loops=...
                Index Cond: (id = genero_obra.obra_id)
```

Consulta 4

Modificação 1

Nesta modificação, usamos um JOIN a menos e não nos utilizamos de EXISTS, para filtrar as obras do estúdio Ghibli.

```sql
SELECT ano, avg(medias) FROM (
SELECT date_part('year',data_lancamento) as ano,
    avg(nota) as medias, obra.id FROM avaliacao
INNER JOIN obra
ON avaliacao.obra_id = obra.id
GROUP BY ano, obra.id
) as t
WHERE
EXISTS
( SELECT obra.id FROM estudio
 INNER JOIN obra
 ON obra.estudio_id = estudio.id
 WHERE nome_estudio = 'Ghibli'
 AND t.id = obra.id
 GROUP BY t.ano, obra.id
)
GROUP BY t.ano;
```

	Node
1.	→ Aggregate
2.	→ Sort
3.	→ Hash Semi Join **Hash Cond**: (obra_1.id = obra.id)
4.	→ Aggregate
5.	→ Hash Inner Join **Hash Cond**: (avaliacao.obra_id = obra_1.id)
6.	→ Seq Scan on avaliacao as avaliacao
7.	→ Hash
8.	→ Seq Scan on obra as obra_1
9.	→ Hash
10.	→ Hash Inner Join **Hash Cond**: (obra.estudio_id = estudio.id)
11.	→ Seq Scan on obra as obra
12.	→ Hash
13.	→ Seq Scan on estudio as estudio **Filter**: (nome_estudio = 'Ghibli'::text)

Modificação 2

Agora, similar à modificação 1, mas agora trazendo no SELECT a coluna estudio_id, permitindo um IN com uma condição simples:

```sql
SELECT ano, avg(medias) FROM (
SELECT date_part('year',data_lancamento) as ano, avg(nota) as medias,
    obra.id, obra.estudio_id AS estd FROM avaliacao
INNER JOIN obra
ON avaliacao.obra_id = obra.id
GROUP BY ano, obra.id
) as t
WHERE estd IN
(SELECT id FROM estudio WHERE nome_estudio = 'Ghibli'
)
GROUP BY t.ano;
```

	Node
1.	→ Aggregate
2.	→ Sort
3.	→ Hash Inner Join **Hash Cond**: (obra.estudio_id = estudio.id)
4.	→ Aggregate
5.	→ Hash Inner Join **Hash Cond**: (avaliacao.obra_id = obra.id)
6.	→ Seq Scan on avaliacao as avaliacao
7.	→ Hash
8.	→ Seq Scan on obra as obra
9.	→ Hash
10.	→ Seq Scan on estudio as estudio **Filter**: (nome_estudio = 'Ghibli'::text)

Artefato E

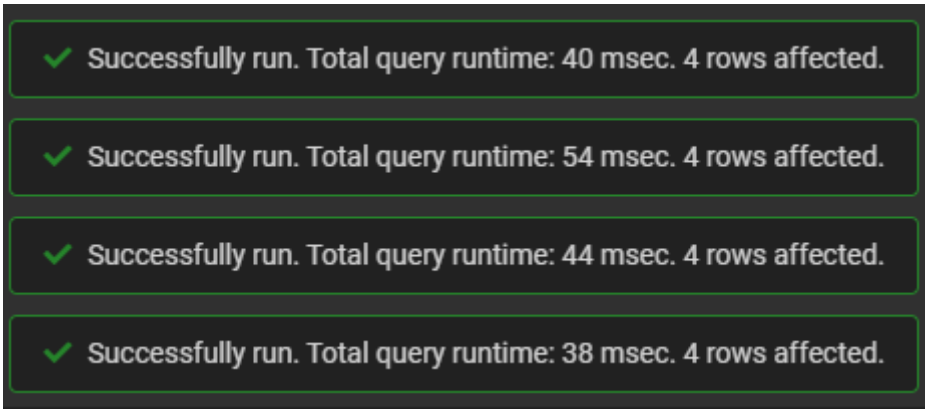
Para este, foi montada a seguinte pesquisa que se utiliza de subquery:

```sql
SELECT nome_ator
FROM ator
WHERE EXISTS(
    SELECT obra.nome
    FROM obra
    INNER JOIN esta_em
    ON esta_em.obra_id = obra.id
    WHERE ator.id = esta_em.ator_id
    AND obra.nome = 'Brocolis 99'
)
```

```
1. → Nested Loop Inner Join (cost=76.01..77.49 rows=5 width=13)
2.     → Aggregate (cost=75.87..75.92 rows=5 width=4)
3.         → Hash Inner Join (cost=32.39..75.85 rows=5 width=4)
             Hash Cond: (esta_em.obra_id = obra.id)
4.             → Seq Scan on esta_em as esta_em (cost=0..36.88 rows=2488 width=8)
5.             → Hash (cost=32.38..32.38 rows=1 width=4)
6.                 → Seq Scan on obra as obra (cost=0..32.38 rows=1 width=4)
                     Filter: (nome = 'Brocolis 99'::text)
7.     → Index Scan using ator_pkey on ator as ator (cost=0.14..0.31 rows=1 width=17)
         Index Cond: (id = esta_em.ator_id)
```

Podemos ver que primeiro, a subquery é executada, depois seu resultado é passado para o SELECT exterior, que é executado.

Em questão de tempo no caso temos:

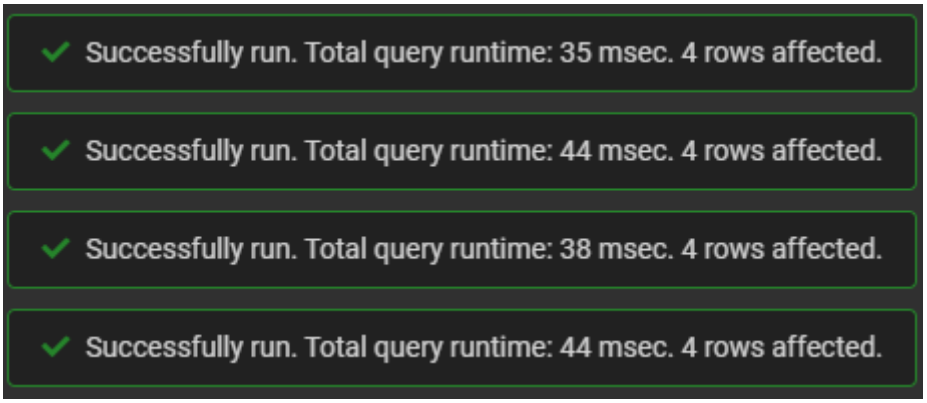


Agora, para que não necessitemos de subqueries, precisamos nos utilizar de vários INNER JOINs, portanto:

```
1  SELECT nome_ator FROM obra
2  INNER JOIN esta_em
3  ON obra.id = esta_em.obra_id
4  INNER JOIN ator
5  ON esta_em.ator_id = ator.id
6  WHERE obra.nome = 'Brocolis 99'
```

	Node
1.	→ Nested Loop Inner Join (cost=32.53..76.68 rows=5 width=13)
2.	→ Hash Inner Join (cost=32.39..75.85 rows=5 width=4) **Hash Cond**: (esta_em.obra_id = obra.id)
3.	→ Seq Scan on esta_em as esta_em (cost=0..36.88 rows=2488 width=8)
4.	→ Hash (cost=32.38..32.38 rows=1 width=4)
5.	→ Seq Scan on obra as obra (cost=0..32.38 rows=1 width=4) **Filter**: (nome = 'Brocolis 99'::text)
6.	→ Index Scan using ator_pkey on ator as ator (cost=0.14..0.16 rows=1 width=17) **Index Cond**: (id = esta_em.ator_id)

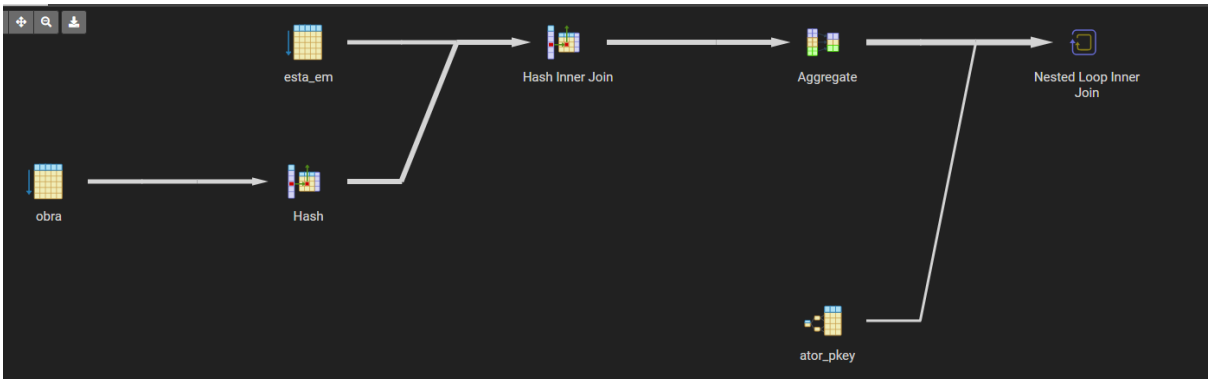
Que também possui um tempo similar (dada a pequena amostra de testes):

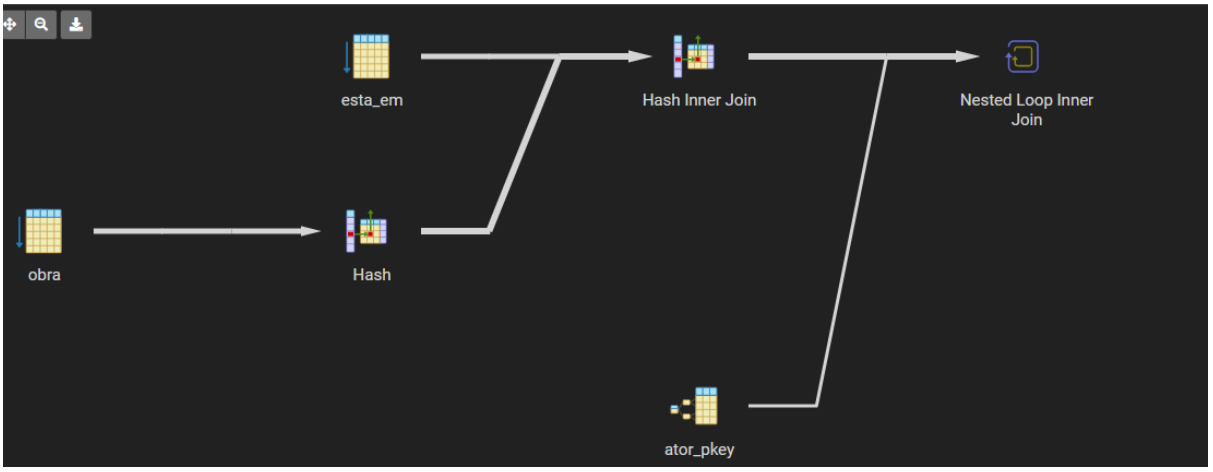


Comparando os dois, temos que (em cima o com subquery, em baixo o sem), vemos que ambas as consultas possuem planos e custos equivalentes, exceto pela necessidade de um aggregate, proporcionado por

ON esta_em.obra_id = obra.id
WHERE ator.id = esta_em.ator_id

Que é feito na chave comum a ambos.





Portanto, a query que não se utiliza da estratégia de subqueries, neste caso, tem um custo menor e, consequentemente, uma performance maior (mesmo que imperceptível neste caso).