



Universidade do Minho

MESTRADO EM ENGENHARIA INFORMÁTICA

ADMINISTRAÇÃO DE BASES DE DADOS

Configuração, Otimização e Avaliação de um *Benchmark*

Ana Murta (PG50184)

Beatriz Oliveira (PG50942)

Gonçalo Soares (PG50393)

Joana Alves (PG50457)

Vicente Moreira (PG50799)

Junho 2023

Conteúdo

1	Introdução	3
2	Otimização das Interrogações Analíticas no <i>PostgreSQL</i>	4
2.1	Benchmarks utilizados	4
2.1.1	Scripts desenvolvidos	4
2.2	Otimizações	5
2.2.1	Interrogação analítica 1	5
2.2.2	Interrogação analítica 2	10
2.2.3	Interrogação analítica 3	16
3	Otimização das Interrogações Analíticas no <i>Spark</i>	20
3.1	<i>Setup</i> Automático do <i>Spark</i>	20
3.2	Exportação dos dados PostgreSQL	20
3.3	Leitura dos Ficheiros TSV	20
3.4	Otimizações	21
3.4.1	Modificações da Interrogação Analítica 1	21
3.4.2	Modificações da Interrogação Analítica 2	22
3.4.3	Modificações da Interrogação Analítica 3	22
3.4.4	Método de Otimização	23
3.4.5	Resultados Agregados	23
4	Otimização da Carga Transaccional	25
4.1	Número de clientes ótimo	25
4.2	Otimização dos diferentes parâmetros de configuração	26
4.3	Settings	26
4.3.1	Parâmetro <i>fsync</i>	26
4.3.2	Parâmetro <i>synchronous_commit</i>	26
4.3.3	Parâmetro <i>WAL_sync_method</i>	27
4.3.4	Parâmetro <i>full_page_writes</i>	27
4.3.5	Parâmetro <i>commit_delay</i>	27
4.3.6	Parâmetro <i>commit_siblings</i>	28
4.4	<i>Checkpoints</i>	28
4.4.1	Parâmetro <i>checkpoint_timeouts</i>	28
4.4.2	Parâmetro <i>max_wal_size</i>	29
4.4.3	Parâmetro <i>min_wal_size</i>	29
4.4.4	Parâmetro <i>checkpoint_completion_target</i>	29
4.4.5	Parâmetro <i>checkpoint_warning</i>	30
4.5	<i>Archiving</i>	30
4.5.1	Parâmetro <i>archive_mode</i>	30
4.6	Configuração final	30
5	Conclusões e Trabalho Futuro	32

Lista de Figuras

1	Plano de execução da <i>query</i> 1 original.	6
2	Plano de execução da <i>query</i> 1 com a tabela <i>avg-rating</i>	8
3	Plano de execução da <i>query</i> 1 com a vista materializada.	9
4	Plano de execução da <i>query</i> 2 original.	11
5	Plano de execução da <i>query</i> 2 com a tabela <i>titleGenresGroup</i>	12
6	Plano de execução da <i>query</i> 2 com os índices.	13
7	Plano de execução da <i>query</i> 2 com a vista materializada.	14
8	Plano de execução da <i>query</i> 2 com os novos índices.	15
9	Plano de execução da <i>query</i> 2 com a segunda versão da vista materializada.	16
10	Plano de execução da <i>query</i> 3 original.	17
11	Plano de execução da <i>query</i> 3 com os índices.	18
12	Plano de execução da <i>query</i> 3 com a tabela <i>actors_roles</i>	19
13	25

1 Introdução

Este trabalho prático desenvolvido no âmbito da unidade curricular Administração de Bases de Dados, do perfil de Engenharia de Aplicações que está integrado no Mestrado em Engenharia Informática da Universidade do Minho.

O objeto de estudo deste projeto corresponde a um *benchmark* que possui dados IMDb e que contém operações transacionais e analíticas. Para além disso, este trabalho tem como objetivos a otimização do desempenho das interrogações analíticas tanto em *PostgreSQL* como em *Spark* e a otimização do desempenho da carga transacional.

Assim sendo, ao longo deste relatório, apresentamos todas as decisões realizadas enquanto grupo e as consequentes justificações, metodologias e processos seguidos para atingir estes objetivos.

2 Otimização das Interrogações Analíticas no *PostgreSQL*

Tal como já foi referido, um dos objetivos deste trabalho prático consiste na otimização da *performance* das interrogações analíticas no *PostgreSQL*, considerando a redundância e os respetivos planos. Deste modo, com o intuito de diminuir os tempos de execução das *queries*, o grupo começou por analisar o plano de execução das mesmas. Para isto, foi utilizado o comando **EXPLAIN ANALYZE** do *PostgreSQL*, que permitiu identificar as operações redundantes e as áreas com problemas na aplicação. Em seguida, o grupo procurou resolver estes problemas através da aplicação de índices, de vistas materializadas e outras soluções.

Assim sendo, de modo a obter uma análise mais precisa e eficiente das *queries*, foram usados as ferramentas *explain.depesz* e *explain.dalibo*. Estas permitiram determinar mais facilmente os problemas das *queries*, visto que forneceram dados relativos às mesmas, como o tempo total gasto em cada operação do plano da *query*, entre outros.

Durante este processo, para além de se considerar a maximização do desempenho das interrogações analíticas, as estratégias aplicadas também foram orientadas pela preservação da flexibilidade das mesmas, evitando-se limitar as condições das variáveis, com valores arbitrários, a um único valor. Desta forma, continua a ser possível que as *queries* sejam aplicadas em diversas situações, uma vez que o utilizador consegue definir os valores que pretende, não prejudicando as otimizações realizadas.

Por fim, enquanto grupo, asseguramos também, constantemente, a consistência e a integridade dos resultados das *queries*, isto é, foi assegurado que estes não sofriam alterações a cada decisão realizada, visto que existiram algumas tentativas em que, efetivamente, existia uma mudança relativa ao resultado esperado.

2.1 Benchmarks utilizados

Enquanto grupo, consideramos que as medidas de desempenho mais relevantes correspondem ao tempo de execução *cached* e não *non-cached*. Pelo que foram estas as medidas tidas maioritariamente em conta, ao longo do processo de otimização das diversas *queries*. No entanto, entre estas duas medidas, priorizamos os valores do tempo de execução *cached*.

2.1.1 Scripts desenvolvidos

Com o intuito de automatizar algumas das etapas deste processo de otimização, criamos três *scripts*. De seguida, explicitamos a função de cada um.

- **query_benchmark.sh** - Este é responsável pela execução de uma interrogação analítica um determinado número de vezes, sendo considerado como *default* cinco execuções. Para além disto, também pode ser passado como segundo parâmetro o nome da base de dados. Neste parâmetro foi considerado o valor *default imdb*. Por fim, como terceiro parâmetro temos a opção de limpar dados *cached* antes de cada execução. O *script* desliga o servidor *postgres*, limpa tanto a memória RAM, como a *cache* e volta a ligar o servidor. Isto, mais uma vez, acontece antes de cada execução da interrogação analítica. Esta pode ser ativada com a *flag* *-no-cache*.
- **query_output.sh** - Este *script* escreve o output da *query* executada para um ficheiro, de modo a facilitar a comparação entre o resultado atual e o resultado original.

2.2 Otimizações

2.2.1 Interrogação analítica 1

```
SELECT *
FROM (
    SELECT t.id,
           left(t.primary_title, 30),
           ((start_year / 10) * 10)::int AS decade,
           avg(uh.rating) AS rating,
           rank() over (
               PARTITION by ((start_year / 10) * 10) :: int
               ORDER BY avg(uh.rating) DESC, t.id
           ) AS rank
    FROM title t
    JOIN userHistory uh ON uh.title_id = t.id
    WHERE t.title_type = 'movie'
           AND ((start_year / 10) * 10)::int >= 1980
           AND t.id IN (
               SELECT title_id
               FROM titleGenre tg
               JOIN genre g ON g.id = tg.genre_id
               WHERE g.name IN (
                   'Drama'
               )
           )
           AND t.id IN (
               SELECT title_id
               FROM titleAkas
               WHERE region IN (
                   'US', 'GB', 'ES', 'DE', 'FR', 'PT'
               )
           )
    GROUP BY t.id
    HAVING count(uh.rating) >= 3
    ORDER BY decade, rating DESC
) t_
WHERE rank <= 10;
```

Esta interrogação analítica tem como objetivo calcular a classificação média de todos os filmes do género, Drama, lançados depois de 1980. Este resultado é ordenado pela década e pela classificação média por ordem decrescente. No final, são apenas selecionados os dez melhores filmes em cada década, ou seja, os que obtiveram uma classificação mais alta.

Primeiramente, começou-se por utilizar o comando **EXPLAIN ANALYZE** para obter o plano de execução gerado pelo *PostgreSQL* e o tempo de execução da *query* original. Este valor correspondeu, em média, a 4223.24 ms.

```

QUERY PLAN
-----
Subquery Scan on t_ (cost=413874.29..413891.72 rows=137 width=86) (actual time=4324.819..4394.139 rows=50 loops=1)
  Filter: (t_rank <= 10)
  Rows Removed by Filter: 2859
  -> WindowAgg (cost=413874.29..413886.59 rows=410 width=86) (actual time=4324.814..4394.020 rows=2109 loops=1)
    -> Sort (cost=413874.29..413875.52 rows=410 width=86) (actual time=4324.755..4392.297 rows=2109 loops=1)
      Sort Key: (((t.start_year / 10) * 10)), (avg(uh.rating)) DESC, t.id
      Sort Method: quicksort Memory: 286kB
      -> Finalize GroupAggregate (cost=413699.90..413856.50 rows=410 width=66) (actual time=4262.697..4388.180 rows=2109 loops=1)
        Group Key: t.id
        Filter: (count(uh.rating) >= 3)
        Rows Removed by Filter: 40845
        -> Gather Merge (cost=413699.90..413828.34 rows=1024 width=74) (actual time=4262.538..4357.149 rows=44895 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          -> Partial GroupAggregate (cost=412699.88..412710.12 rows=512 width=74) (actual time=4236.821..4251.168 rows=14965 loops=3)
            Group Key: t.id
            -> Sort (cost=412699.88..412701.16 rows=512 width=38) (actual time=4236.766..4240.578 rows=28354 loops=3)
              Sort Key: t.id
              Sort Method: quicksort Memory: 2433kB
              Worker 0: Sort Method: quicksort Memory: 2392kB
              Worker 1: Sort Method: quicksort Memory: 2446kB
              -> Nested Loop Semi Join (cost=188123.19..412676.84 rows=512 width=38) (actual time=1344.984..4207.146 rows=28354 loops=3)
                -> Nested Loop Semi Join (cost=188122.50..393763.50 rows=9265 width=58) (actual time=1344.711..3500.871 rows=56261 loops=3)
                  -> Parallel Hash Join (cost=188122.03..299810.07 rows=53022 width=48) (actual time=1344.615..1969.673 rows=95932 loops=3)
                    Hash Cond: ((uh.title_id)::text = (t.id)::text)
                    -> Parallel Seq Scan on userhistory uh (cost=0.00..88627.96 rows=2499496 width=14) (actual time=0.063..330.204 rows=1999487 loops=3)
                    -> Parallel Hash (cost=186331.18..186331.18 rows=88140 width=34) (actual time=535.742..535.744 rows=129753 loops=3)
                      Buckets: 65536 Batches: 8 Memory Usage: 3840kB
                      -> Parallel Bitmap Heap Scan on title t (cost=6992.86..186331.18 rows=88140 width=34) (actual time=64.186..486.526 rows=129753 loops=3)
                        Recheck Cond: ((title_type)::text = 'movie'::text)
                        Rows Removed by Index Recheck: 830665
                        Filter: (((start_year / 10) * 10) >= 1980)
                        Rows Removed by Filter: 84918
                        Heap Blocks: exact=22996 lossy=12036
                        -> Bitmap Index Scan on idx_title_type (cost=0.00..6939.97 rows=634605 width=0) (actual time=50.354..50.354 rows=644013 loops=1)
                          Index Cond: ((title_type)::text = 'movie'::text)
                        -> Index Scan using titleleakas_play on titleleakas (cost=0.56..36.02 rows=35 width=10) (actual time=0.016..0.016 rows=1 loops=287796)
                          Index Cond: ((title_id)::text = (uh.title_id)::text)
                          Filter: ((region)::text = ANY ('{US,GB,ES,DE,FR,PT}'::text[]))
                          Rows Removed by Filter: 1
                        -> Nested Loop (cost=0.70..2.03 rows=1 width=10) (actual time=0.012..0.012 rows=0 loops=174783)
                          -> Index Only Scan using titlegenre_pkey on titlegenre tg (cost=0.56..1.38 rows=4 width=14) (actual time=0.010..0.010 rows=1 loops=174783)
                            Index Cond: (title_id = (uh.title_id)::text)
                            Heap Fetches: 0
                          -> Index Scan using genre_pkey on genre g (cost=0.14..0.16 rows=1 width=4) (actual time=0.001..0.001 rows=0 loops=242140)
                            Index Cond: (id = tg.genre_id)
                            Filter: ((name)::text = 'Drama'::text)
                            Rows Removed by Filter: 1
Planning Time: 1.405 ms
JIT:
  Functions: 114
  Options: inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 6.767 ms, Inlining 0.000 ms, Optimization 3.255 ms, Emission 67.102 ms, Total 77.124 ms
Execution Time: 4396.832 ms
(54 rows)

```

Figura 1: Plano de execução da *query* 1 original.

Após a uma análise inicial tanto do plano, como da *query*, observou-se que a interrogação minimiza o seu desempenho, uma vez que contém várias operações de *JOIN* para determinar. Mais especificamente, as múltiplas operações existentes nesta *query* de combinação/junção das tabelas, que têm como intuito determinar o *average rating*, estão a degradar a *performance* da mesma. Consequentemente, decidimos criar uma tabela para registar estes valores à medida que fossem alterando. Desta forma, durante a execução da consulta, os dados já calculados são acedidos em vez de serem calculados sempre que a *query* é executada. Apesar desta solução ter um custo acrescido quando um utilizador adiciona uma *review*, esta parece ser a melhor solução quando comparada à criação de uma vista materializada, visto que a tabela criada é atualizada de forma incremental, ao contrário da vista materializada que teria que ser totalmente recalculada. Posteriormente, criou-se também um *trigger* para atualizar esta tabela.

```

CREATE
  TABLE avg_rating AS
SELECT title_id, avg(rating) AS avg_rating, count(rating) AS count
FROM userHistory
GROUP BY title_id;

CREATE OR REPLACE FUNCTION update_avg_rating()
RETURNS TRIGGER AS $$
BEGIN
  IF (TG_OP = 'INSERT') THEN
    UPDATE avg_rating
      SET avg_rating = ((avg_rating * count) + NEW.rating) / (count + 1)
      , count = count + 1

```

```

        WHERE title_id = NEW.title_id;
        RETURN NEW;
    ELSIF (TG_OP = 'DELETE') THEN
        UPDATE avg_rating
        SET avg_rating = ((avg_rating * count) - OLD.rating) / (count - 1)
            , count = count - 1
        WHERE title_id = OLD.title_id;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        UPDATE avg_rating
        SET avg_rating = ((avg_rating * count) - OLD.rating + NEW.rating) / count
        WHERE title_id = OLD.title_id;
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER update_avg_rating AFTER INSERT OR UPDATE OR DELETE ON userHistory
FOR EACH ROW EXECUTE FUNCTION update_avg_rating();

```

```

SELECT *
FROM (SELECT t.id, left (t.primary_title, 30), ((start_year / 10) * 10):: int
AS decade, avg_r.avg_rating AS rating, rank() over (
    PARTITION by ((start_year / 10) * 10):: int
    ORDER BY avg_r.avg_rating DESC, t.id
) AS rank
FROM title t
    JOIN avg_rating avg_r
ON avg_r.title_id = t.id
    JOIN titleAkas ta ON ta.title_id = t.id
WHERE t.title_type = 'movie'
    AND ((start_year / 10) * 10):: int >= 1980
    AND ta.region IN ('US'
        , 'GB'
        , 'ES'
        , 'DE'
        , 'FR'
        , 'PT')
    AND t.id IN (
        SELECT title_id
        FROM titleGenre tg
        JOIN genre g ON g.id = tg.genre_id
        WHERE g.name = 'Drama'
    )
    AND avg_r.count >= 3
GROUP BY t.id, avg_r.avg_rating
ORDER BY decade, rating DESC) t_

```



```
WHERE rank <= 10;
```

Através da execução da interrogação, foi possível comprovar que esta estratégia teve um impacto positivo no desempenho da mesma, dado que resultou numa diminuição do tempo de execução, que, neste ponto, corresponde, em média, a 841.59 ms.

```

QUERY PLAN
Subquery Scan on t_ (cost=246981.35..247829.00 rows=374 width=59) (actual time=883.359..811.264 rows=59 loops=1)
  Filter: (t_.rank <= 10)
  Rows Removed by Filter: 2859
  -> WindowAgg (cost=246981.35..247814.98 rows=1121 width=59) (actual time=883.354..811.151 rows=2109 loops=1)
    -> Sort (cost=246981.35..246984.16 rows=1121 width=39) (actual time=883.314..809.511 rows=2109 loops=1)
      Sort Key: (((t.start_year / 10) * 10)), avg_r.avg_rating DESC, t.id
      Sort Method: quicksort Memory: 286kB
      -> Group (cost=246882.99..246924.57 rows=1121 width=39) (actual time=797.555..885.581 rows=2109 loops=1)
        Group Key: t.id, avg_r.avg_rating
        -> Gather Merge (cost=246882.99..246914.30 rows=934 width=39) (actual time=797.550..884.849 rows=2109 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          -> Group (cost=245882.96..245886.47 rows=467 width=39) (actual time=773.064..773.575 rows=783 loops=3)
            Group Key: t.id, avg_r.avg_rating
            -> Sort (cost=245882.96..245884.13 rows=467 width=39) (actual time=773.027..773.126 rows=1759 loops=3)
              Sort Key: t.id, avg_r.avg_rating DESC
              Sort Method: quicksort Memory: 216kB
              Worker 0: Sort Method: quicksort Memory: 198kB
              Worker 1: Sort Method: quicksort Memory: 189kB
              -> Nested Loop (cost=57753.24..245782.26 rows=467 width=39) (actual time=256.213..771.726 rows=1759 loops=3)
                Join Filter: ((avg_r.title_id)::text = (ta.title_id)::text)
                -> Nested Loop Semi Join (cost=57752.67..242318.15 rows=185 width=59) (actual time=255.715..746.883 rows=1181 loops=3)
                  -> Parallel Hash Join (cost=57751.97..237428.73 rows=1896 width=49) (actual time=253.952..782.347 rows=3255 loops=3)
                    Hash Cond: ((t.id)::text = (avg_r.title_id)::text)
                    -> Parallel Bitmap Heap Scan on title t_ (cost=6992.86..186331.18 rows=88140 width=34) (actual time=63.645..461.868 rows=129753 loops=3)
                      Recheck Cond: ((title_type)::text = 'movie'::text)
                      Rows Removed by Index Recheck: 838665
                      Filter: (((start_year / 10) * 10) >= 1980)
                      Rows Removed by Filter: 84918
                      Heap Blocks: exact=22730 lossy=12052
                      -> Bitmap Index Scan on idx_title_type (cost=0.00..6939.97 rows=634685 width=0) (actual time=48.961..48.961 rows=644013 loops=1)
                        Index Cond: ((title_type)::text = 'movie'::text)
                    -> Parallel Hash (cost=49658.53..49658.53 rows=88847 width=15) (actual time=184.967..184.969 rows=67271 loops=3)
                      Buckets: 262144 Batches: 1 Memory Usage: 12640kB
                      -> Parallel Seq Scan on avg_rating avg_r_ (cost=0.00..49658.53 rows=88847 width=15) (actual time=24.985..155.184 rows=67271 loops=3)
                        Filter: (count >= 3)
                        Rows Removed by Filter: 1345243
                  -> Nested Loop (cost=0.70..2.57 rows=1 width=10) (actual time=0.813..0.813 rows=0 loops=9766)
                    -> Index Only Scan using titlegenre_pkey on titlegenre tg_ (cost=0.56..1.92 rows=4 width=14) (actual time=0.010..0.011 rows=1 loops=9766)
                      Index Cond: (title_id = (avg_r.title_id)::text)
                      Heap Fetches: 0
                    -> Index Scan using genre_pkey on genre g_ (cost=0.14..0.16 rows=1 width=4) (actual time=0.001..0.001 rows=0 loops=12384)
                      Index Cond: (id = tg.genre_id)
                      Filter: ((name)::text = 'Drama'::text)
                      Rows Removed by Filter: 1
                  -> Index Scan using titleakas_pkey on titleakas ta_ (cost=0.56..32.55 rows=35 width=10) (actual time=0.019..0.021 rows=1 loops=3544)
                    Index Cond: ((title_id)::text = (tg.title_id)::text)
                    Filter: ((region)::text = ANY ('{US,GB,ES,DE,FR,PT}'::text[]))
                    Rows Removed by Filter: 4
Planning Time: 1.460 ms
JIT:
  Functions: 118
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 7.192 ms, Inlining 0.000 ms, Optimization 3.283 ms, Emission 71.681 ms, Total 82.156 ms
Execution Time: 813.852 ms
(55 rows)

```

Figura 2: Plano de execução da *query* 1 com a tabela *avg_rating*.

Por último, uma outra alternativa que adotámos com o propósito de maximizar a *performance* da *query* foi a aplicação de uma vista materializada, que calcula os *average ratings*. A sua introdução deve-se às mesmas razões já mencionadas anteriormente relativas às várias operações *JOIN* existentes. Contudo, como já foi referido, a estratégia da tabela é mais eficiente que a esta. No entanto, enquanto grupo, queríamos analisar esta outra opção. Adaptando a *query* à vista materializada, esta fica da seguinte forma:

```

CREATE
MATERIALIZED VIEW avg_rating AS
SELECT title_id, avg(rating) AS avg_rating, count(rating) AS count
FROM userHistory
GROUP BY title_id;

SELECT *
FROM (SELECT t.id, left (t.primary_title, 30), ((start_year / 10) * 10):: int
AS decade, avg_r.avg_rating AS rating, rank() over (
PARTITION by ((start_year / 10) * 10):: int
ORDER BY avg_r.avg_rating DESC, t.id

```

```

    ) AS rank
FROM title t
    JOIN avg_rating avg_r
ON avg_r.title_id = t.id
    JOIN titleAkas ta ON ta.title_id = t.id
WHERE t.title_type = 'movie'
    AND ((start_year / 10) * 10):: int >= 1980
    AND ta.region IN ('US'
    , 'GB'
    , 'ES'
    , 'DE'
    , 'FR'
    , 'PT')
AND t.id IN (
    SELECT title_id
    FROM titleGenre tg
    JOIN genre g ON g.id = tg.genre_id
    WHERE g.name = 'Drama'
)
    AND avg_r.count >= 3
GROUP BY t.id, avg_r.avg_rating
ORDER BY decade, rating DESC) t_
WHERE rank <= 10;

```

E tal como esperado, esta mudança permitiu reduzir o tempo de execução da interrogação analítica para, em média, 845.37 ms.

```

QUERY PLAN
-----
Subquery Scan on t_ (cost=246619.82..246665.60 rows=359 width=59) (actual time=783.230..791.247 rows=50 loops=1)
  Filter: (t_rank <= 10)
  Rows Removed by Filter: 2859
  -> WindowAgg (cost=246619.82..246652.13 rows=1077 width=59) (actual time=783.224..791.133 rows=2109 loops=1)
    -> Sort (cost=246619.82..246622.52 rows=1077 width=39) (actual time=783.187..789.463 rows=2109 loops=1)
      Sort Key: (((t.start_year / 10) * 10)), avg_r.avg_rating DESC, t.id
      Sort Method: quicksort  Memory: 28640
      -> Group (cost=246648.69..246565.58 rows=1077 width=39) (actual time=777.574..785.589 rows=2109 loops=1)
        Group Key: t.id, avg_r.avg_rating
        -> Gather Merge (cost=246648.69..246555.71 rows=898 width=39) (actual time=777.569..784.887 rows=2109 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          -> Group (cost=245448.66..245452.03 rows=449 width=39) (actual time=755.085..755.587 rows=783 loops=3)
            Group Key: t.id, avg_r.avg_rating
            -> Sort (cost=245448.66..245449.79 rows=449 width=39) (actual time=755.056..755.155 rows=1759 loops=3)
              Sort Key: t.id, avg_r.avg_rating DESC
              Sort Method: quicksort  Memory: 22448
              Worker 0: Sort Method: quicksort  Memory: 19348
              Worker 1: Sort Method: quicksort  Memory: 18548
              -> Nested Loop (cost=57712.04..245428.88 rows=449 width=39) (actual time=251.765..753.809 rows=1759 loops=3)
                Join Filter: ((avg_r.title_id)::text = (ta.title_id)::text)
                -> Nested Loop Semi Join (cost=57711.47..242096.74 rows=181 width=59) (actual time=251.215..728.687 rows=1181 loops=3)
                  Join Filter: ((avg_r.title_id)::text = (tg.title_id)::text)
                  -> Parallel Hash Join (cost=57710.77..237387.23 rows=1825 width=49) (actual time=249.029..686.235 rows=3255 loops=3)
                    Hash Cond: ((t.id)::text = (avg_r.title_id)::text)
                    -> Parallel Bitmap Heap Scan on title t (cost=6992.86..186331.18 rows=88140 width=34) (actual time=64.661..452.344 rows=129753 loops=3)
                      Recheck Cond: ((title_type)::text = 'movie'::text)
                      Rows Removed by Index Recheck: 830665
                      Filter: (((start_year / 10) * 10) >= 1980)
                      Heap Blocks: exact=22656 lossy=11905
                      -> Bitmap Index Scan on idx_title_type (cost=0.00..6939.97 rows=634605 width=0) (actual time=49.969..49.969 rows=640813 loops=1)
                        Index Cond: (((title_type)::text = 'movie'::text))
                    -> Parallel Hash (cost=90652.53..49658.53 rows=89751 width=15) (actual time=179.665..179.666 rows=67271 loops=3)
                      Buckets: 262104  Batches: 1  Memory Usage: 1264048
                      -> Parallel Seq Scan on avg_rating avg_r (cost=0.00..49658.53 rows=84751 width=15) (actual time=23.838..150.454 rows=67271 loops=3)
                        Filter: (count >= 3)
                        Rows Removed by Filter: 1345243
                -> Nested Loop (cost=0.78..2.57 rows=1 width=10) (actual time=0.013..0.013 rows=0 loops=9766)
                  -> Index Only Scan using titlegenre_pkey on titlegenre tg (cost=0.56..1.92 rows=4 width=14) (actual time=0.010..0.010 rows=1 loops=9766)
                    Index Cond: (title_id = (t.id)::text)
                    Heap Fetches: 0
                  -> Index Scan using genre_pkey on genre g (cost=0.14..0.16 rows=1 width=4) (actual time=0.001..0.001 rows=0 loops=12384)
                    Index Cond: (id = tg.genre_id)
                    Filter: ((name)::text = 'Drama'::text)
                    Rows Removed by Filter: 1
                -> Index Scan using titleakas_pkey on titleakas ta (cost=0.56..32.55 rows=35 width=10) (actual time=0.019..0.020 rows=1 loops=3540)
                    Index Cond: ((title_id)::text = (tg.title_id)::text)
                    Filter: ((region)::text = ANY ('{US,GB,ES,DE,FR,PT}'::text[]))
                    Rows Removed by Filter: 4
Planning Time: 1.361 ms
JIT:
  Functions: 121
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 7.569 ms, Inlining 0.000 ms, Optimization 3.028 ms, Emission 68.802 ms, Total 79.400 ms
Execution Time: 794.136 ms
(56 rows)

```

Figura 3: Plano de execução da *query* 1 com a vista materializada.

2.2.2 Interrogação analítica 2

```
SELECT t.id, t.primary_title, tg.genres, te.season_number, count(*) AS views
FROM title t
JOIN titleEpisode te ON te.parent_title_id = t.id
JOIN title t2 ON t2.id = te.title_id
JOIN userHistory uh ON uh.title_id = t2.id
JOIN users u ON u.id = uh.user_id
JOIN (
    SELECT tg.title_id, array_agg(g.name) AS genres
    FROM titleGenre tg
    JOIN genre g ON g.id = tg.genre_id
    GROUP BY tg.title_id
) tg ON tg.title_id = t.id
WHERE t.title_type = 'tvSeries'
    AND uh.last_seen BETWEEN NOW() - INTERVAL '30 days' AND NOW()
    AND te.season_number IS NOT NULL
    AND u.country_code NOT IN ('US', 'GB')
GROUP BY t.id, t.primary_title, tg.genres, te.season_number
ORDER BY count(*) DESC, t.id
LIMIT 100;
```

Esta interrogação analítica tem como objetivo identificar os cem títulos de séries de TV com o maior número de visualizações nos últimos 30 dias, excluindo os utilizadores dos países com os códigos *US* (Estados Unidos) e *GB* (Reino Unido). Depois de executada, a *query* apresenta para cada *id* do título, o seu título principal, os seus géneros, a temporada e o número de visualizações.

Seguindo a estratégia utilizada na *query* anterior, começamos por utilizar o comando **EXPLAIN ANALYZE**, obtendo o plano de execução gerado pelo *PostgreSQL* e o tempo de execução da *query* original, que corresponde, em média, a 117188,172 ms.

```

QUERY PLAN
-----
Limit (cost=1217995.36..1217995.61 rows=100 width=74) (actual time=17288.470..17288.686 rows=100 loops=1)
  -> Sort (cost=1217995.36..1218155.01 rows=62861 width=74) (actual time=16874.604..16874.812 rows=100 loops=1)
    Sort Key: (count(*)) DESC, t.id
    Sort Method: top-N heapsort Memory: 41kB
    -> GroupAggregate (cost=164517.01..1215554.64 rows=62861 width=74) (actual time=1671.847..16864.562 rows=21082 loops=1)
      Group Key: t.id, (array_agg(g.name)), te.season_number
      -> Incremental Sort (cost=164517.01..1216277.42 rows=62861 width=66) (actual time=1671.012..16827.484 rows=55704 loops=1)
        Sort Key: t.id, (array_agg(g.name)), te.season_number
        Presorted Key: t.id
        Full-Sorted Groups: 1524 Sort Method: quicksort Average Memory: 29kB Peak Memory: 29kB
        Pre-sorted Groups: 416 Sort Method: quicksort Average Memory: 25kB Peak Memory: 25kB
        -> Merge Join (cost=164479.06..1211987.61 rows=62861 width=66) (actual time=1668.018..16779.822 rows=55704 loops=1)
          Merge Cond: ((te.parent_title_id)::text = (tg.title_id)::text)
          -> Nested Loop (cost=164479.46..1272146.97 rows=642 width=64) (actual time=1607.745..2087.455 rows=57695 loops=1)
            -> Nested Loop (cost=164478.17..176878.35 rows=646 width=48) (actual time=1607.698..2081.007 rows=58129 loops=1)
              -> Gather Merge (cost=164477.74..166610.94 rows=18216 width=18) (actual time=1607.570..1647.309 rows=60786 loops=1)
                Workers Planned: 2
                Workers Launched: 2
                -> Sort (cost=162477.71..163496.79 rows=7622 width=18) (actual time=1583.025..1589.345 rows=20262 loops=3)
                  Sort Key: te.parent_title_id
                  Sort Method: quicksort Memory: 249kB
                  Worker 0: Sort Method: quicksort Memory: 227kB
                  Worker 1: Sort Method: quicksort Memory: 226kB
                  -> Nested Loop (cost=0.87..162985.33 rows=7622 width=18) (actual time=129.694..1558.084 rows=20262 loops=3)
                    -> Parallel Seq Scan on userhistory uh (cost=0.00..111291.05 rows=12653 width=14) (actual time=129.575..1084.624 rows=28170 loops=3)
                      Filter: ((last_seen <= now()) AND (last_seen >= (now() - '20 days'::interval)))
                    -> Index Only Scan using title_play on title t2 (cost=0.43..3.60 rows=1 width=10) (actual time=0.012..0.012 rows=1 loops=84509)
                      Index Cond: (id = (uh.title_id)::text)
                      Heap Fetches: 5174
                  -> Index Scan using titleepisode_play on titleepisode te (cost=0.43..0.48 rows=1 width=24) (actual time=0.016..0.016 rows=1 loops=84509)
                      Index Cond: ((title_id)::text = (t2.id)::text)
                      Filter: (season_number IS NOT NULL)
                      Rows Removed by Filter: 0
                -> Index Scan using title_play on title t (cost=0.43..0.56 rows=1 width=30) (actual time=0.007..0.007 rows=1 loops=60786)
                      Index Cond: ((id)::text = (te.parent_title_id)::text)
                      Filter: ((title_type)::text = 'tvSeries'::text)
                      Rows Removed by Filter: 0
              -> Index Scan using users_play on users u (cost=0.29..0.60 rows=1 width=4) (actual time=0.005..0.005 rows=1 loops=58129)
                  Index Cond: (id = uh.user_id)
                  Filter: ((country_code)::text <> ALL (('US,GB')::text[]))
                  Rows Removed by Filter: 0
            -> GroupAggregate (cost=0.60..984211.15 rows=3991182 width=42) (actual time=0.160..12298.706 rows=9280445 loops=1)
              Group Key: tg.title_id
              -> Nested Loop (cost=0.60..858408.89 rows=15202498 width=42) (actual time=0.131..6962.126 rows=15202069 loops=1)
                -> Index Only Scan using titlegenre_play on titlegenre tg (cost=0.42..478520.78 rows=15202498 width=14) (actual time=0.087..2257.158 rows=15202069 loops=1)
                  Heap Fetches: 1520139
                -> Memoize (cost=0.16..0.18 rows=1 width=26) (actual time=0.000..0.000 rows=1 loops=15202069)
                  Cache Key: tg.genre_id
                  Cache Mode: logical
                  Hits: 15202041 Misses: 28 Evictions: 0 Overflows: 0 Memory Usage: 4kB
                  -> Index Scan using genre_play on genre g (cost=0.15..0.17 rows=1 width=26) (actual time=0.002..0.002 rows=1 loops=28)
                      Index Cond: (id = tg.genre_id)
              Rows Removed by Filter: 0
      Rows Removed by Filter: 0
Planning Time: 6.150 ms
JIT:
  Functions: 73
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 4.226 ms, Inlining 259.724 ms, Optimization 211.249 ms, Emission 221.097 ms, Total 806.298 ms
Execution Time: 17318.686 ms
(60 rows)

```

Figura 4: Plano de execução da *query* 2 original.

Após uma análise inicial do plano e da *query*, observou-se que a interrogação apresenta uma sub-consulta que agrupa os géneros associados a cada título, concluindo-se que é uma operação custosa em termos do tempo de execução. Pelo que o grupo decidiu que não era necessário realizá-la durante a execução da *query* e que a mesma passaria a aceder aos resultados já calculados. Como tal e como uma primeira tentativa, foi criada uma vista materializada para, desse modo, obter-se de uma forma mais simples o resultado da *sub-query*.

Contudo, depará-mo-nos com um problema após a criação desta, mais especificamente, a vista necessita de ser totalmente recalculada sempre que se quiser inserir um novo título. Deste modo, optou-se, novamente, pela criação de uma tabela que guardasse os resultados da *sub-query* e, assim, seria apenas necessário fazer um *INSERT* quando quiséssemos adicionar um novo título, ou seja, trata-se de uma operação menos "pesada" do que recalculiar a vista inteira.

```

CREATE TABLE titleGenresGroup AS
SELECT tg.title_id, array_agg(g.name) AS genres
FROM titleGenre tg
JOIN genre g ON g.id = tg.genre_id
GROUP BY tg.title_id;

SELECT t.id, t.primary_title, tg.genres, te.season_number, count(*) AS views
FROM title t
JOIN titleEpisode te ON te.parent_title_id = t.id
JOIN title t2 ON t2.id = te.title_id
JOIN userHistory uh ON uh.title_id = t2.id

```

```

JOIN users u ON u.id = uh.user_id
JOIN titleGenresGroup tg ON tg.title_id = t.id
WHERE t.title_type = 'tvSeries'
      AND uh.last_seen BETWEEN NOW() - INTERVAL '30 days' AND NOW()
      AND te.season_number IS NOT NULL
      AND u.country_code NOT IN ('US', 'GB')
GROUP BY t.id, t.primary_title, tg.genres, te.season_number
ORDER BY count(*) DESC, t.id
LIMIT 100;

```

A estratégia utilizada teve um impacto bastante positivo no desempenho da *query*, resultando numa redução significativa do tempo de execução, que se encontra, em média, nos 2739,395 ms.

```

----- QUERY PLAN -----

Limit (cost=309033.87..309034.12 rows=100 width=83) (actual time=2807.703..2814.735 rows=100 loops=1)
--> Sort (cost=309033.87..309034.76 rows=358 width=83) (actual time=2785.508..2792.531 rows=100 loops=1)
    Sort Key: (count(*)) DESC, t.id
    Sort Method: top-N heapsort  Memory: 41kB
    --> GroupAggregate (cost=308729.27..309020.19 rows=358 width=83) (actual time=2592.059..2784.841 rows=29390 loops=1)
        Group Key: t.id, tg.genres, te.season_number
        --> Nested Loop (cost=308729.27..309013.03 rows=358 width=75) (actual time=2592.017..2761.990 rows=51634 loops=1)
            --> Gather Merge (cost=308728.98..308771.02 rows=361 width=79) (actual time=2591.914..2619.252 rows=52024 loops=1)
                Workers Planned: 2
                Workers Launched: 2
                --> Sort (cost=307728.96..307729.33 rows=150 width=79) (actual time=2570.657..2572.285 rows=17341 loops=3)
                    Sort Key: t.id, tg.genres, te.season_number
                    Sort Method: quicksort  Memory: 3047kB
                    Worker 0: Sort Method: quicksort  Memory: 3095kB
                    Worker 1: Sort Method: quicksort  Memory: 3018kB
                    --> Parallel Hash Join (cost=158368.20..307723.53 rows=150 width=79) (actual time=1624.236..2546.186 rows=17341 loops=3)
                        Hash Cond: ((tg.title_id)::text = (te.parent_title_id)::text)
                        --> Parallel Seq Scan on titlegenresgroup tg (cost=0.00..134695.15 rows=3909215 width=51) (actual time=0.032..322.657 rows=3126915 loops=3)
                        --> Parallel Hash (cost=158366.33..158366.33 rows=150 width=48) (actual time=1619.532..1619.537 rows=17961 loops=3)
                            Buckets: 65536 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 5432kB
                            --> Nested Loop (cost=1.30..158366.33 rows=150 width=48) (actual time=12.514..1534.659 rows=17961 loops=3)
                                --> Nested Loop (cost=0.87..154902.00 rows=6180 width=18) (actual time=12.477..1308.599 rows=18775 loops=3)
                                    --> Nested Loop (cost=0.43..149948.59 rows=10246 width=24) (actual time=12.353..941.185 rows=26079 loops=3)
                                        --> Parallel Seq Scan on userhistory uh (cost=0.00..111290.06 rows=10246 width=14) (actual time=12.260..684.880 rows=26079 loops=3)
                                            Filter: ((last_seen <= now()) AND (last_seen >= (now() - '30 days'::interval)))
                                            Rows Removed by Filter: 1973852
                                        --> Index Only Scan using title_pkey on title t2 (cost=0.43..3.77 rows=1 width=10) (actual time=0.009..0.009 rows=1 loops=7823)
                                            Index Cond: (id = (uh.title_id)::text)
                                            Heap Fetches: 4777
                                    --> Index Scan using titleepisode_pkey on titleepisode te (cost=0.43..0.48 rows=1 width=24) (actual time=0.014..0.014 rows=1 loops=78237)
                                        Index Cond: ((title_id)::text = (t2.id)::text)
                                        Filter: (season_number IS NOT NULL)
                                        Rows Removed by Filter: 0
                                --> Index Scan using title_pkey on title t (cost=0.43..0.56 rows=1 width=30) (actual time=0.012..0.012 rows=1 loops=56325)
                                    Index Cond: ((id)::text = (te.parent_title_id)::text)
                                    Filter: ((title_type)::text = 'tvSeries'::text)
                                    Rows Removed by Filter: 0
                            --> Index Scan using users_pkey on users u (cost=0.29..0.67 rows=1 width=4) (actual time=0.002..0.002 rows=1 loops=52024)
                                Index Cond: (id = uh.user_id)
                                Filter: ((country_code)::text <> ALL ('US,GB')::text[]))
                                Rows Removed by Filter: 0
            Planning Time: 4.755 ms
            JIT:
                Functions: 97
                Options: Inlining false, Optimization false, Expressions true, Deforming true
                Timing: Generation 5.100 ms, Inlining 0.000 ms, Optimization 2.874 ms, Emission 56.147 ms, Total 64.121 ms
            Execution Time: 2839.850 ms
            (47 rows)

```

Figura 5: Plano de execução da *query* 2 com a tabela *titleGenresGroup*.

De seguida, o grupo reparou que a interrogação analítica filtrava a tabela *userHistory* selecionando apenas as linhas que correspondem aos filmes visualizados pelo utilizador nos últimos 30 dias. Para além desta, são também filtradas as linhas da tabela *users* cujo o código do país não pertence aos códigos dos países estabelecidos. O mesmo acontece na tabela *title* em que são apenas selecionadas as linhas que correspondem a um determinado tipo. Portanto, foram criados os índices, que se encontram abaixo, de forma a tornar a sua identificação mais rápida, uma vez que cada uma destas operações seleciona uma percentagem das linhas das tabelas.

```

create index idx_last_seen on userHistory(last_seen);
create index idx_country_code on users(country_code);
create index idx_title_type on title(title_type);

```

Ao executar novamente a *query*, comprovou-se que esta alteração teve um impacto favorável na *performance* da mesma, reduzindo o tempo de execução para 2252,145 ms, em média, como podemos ver pela figura 6.

```

QUERY PLAN
-----
Limit (cost=212179.47..212179.72 rows=180 width=83) (actual time=2275.641..2286.827 rows=180 loops=1)
--> Sort (cost=212179.47..212179.87 rows=158 width=83) (actual time=2248.958..2259.336 rows=180 loops=1)
    Sort Key: (count(*)) DESC, t.id
    Sort Method: top-N heapsort  Memory: 41kB
    --> Finalize GroupAggregate (cost=212154.08..212173.70 rows=158 width=83) (actual time=2216.767..2252.715 rows=26419 loops=1)
        Group Key: t.id, tg.genres, te.season_number
        --> Gather Merge (cost=212154.08..212170.80 rows=132 width=83) (actual time=2216.744..2241.234 rows=26419 loops=1)
            Workers Planned: 2
            Workers Launched: 2
            --> Partial GroupAggregate (cost=211154.06..211155.54 rows=66 width=83) (actual time=2192.933..2200.693 rows=8866 loops=3)
                Group Key: t.id, tg.genres, te.season_number
                --> Sort (cost=211154.06..211154.22 rows=66 width=75) (actual time=2192.864..2193.969 rows=14784 loops=3)
                    Sort Key: t.id, tg.genres, te.season_number
                    Sort Method: quicksort  Memory: 2428kB
                    Worker 0: Sort Method: quicksort  Memory: 2502kB
                    Worker 1: Sort Method: quicksort  Memory: 2501kB
                    --> Parallel Hash Join (cost=61798.83..211152.06 rows=66 width=75) (actual time=1259.140..2171.960 rows=14784 loops=3)
                        Hash Cond: ((tg.title_id)::text = (te.parent_title_id)::text)
                        --> Parallel Seq Scan on titlegenresgroup tg (cost=0.00..134693.88 rows=3909080 width=51) (actual time=0.075..348.635 rows=3126915 loops=3)
                        --> Parallel Hash (cost=61798.00..61798.00 rows=66 width=44) (actual time=1253.975..1253.984 rows=15313 loops=3)
                            Buckets: 65536 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 4600kB
                            --> Nested Loop (cost=218.48..61798.00 rows=66 width=44) (actual time=26.092..1157.708 rows=15313 loops=3)
                                --> Nested Loop (cost=218.19..61722.48 rows=66 width=40) (actual time=26.050..1105.995 rows=15442 loops=3)
                                    --> Nested Loop (cost=217.75..61673.32 rows=88 width=60) (actual time=26.000..816.897 rows=15442 loops=3)
                                        --> Parallel Hash (cost=219.22..59718.72 rows=3483 width=38) (actual time=25.929..560.572 rows=16169 loops=3)
                                            --> Parallel Bitmap Heap Scan on userhistory uh (cost=216.89..26799.13 rows=4408 width=14) (actual time=9.399..141.501 rows=22553 loops=3)
                                                Recheck Cond: ((last_seen >= (now() - '30 days'::interval)) AND (last_seen <= now()))
                                                Heap Blocks: exact=14377
                                                --> Bitmap Index Scan on idx_last_seen (cost=0.00..214.24 rows=18500 width=0) (actual time=20.198..20.198 rows=67659 loops=1)
                                                    Index Cond: ((last_seen >= (now() - '30 days'::interval)) AND (last_seen <= now()))
                                                --> Index Scan using titleepisode_pkey on titleepisode te (cost=0.43..7.47 rows=1 width=24) (actual time=0.017..0.017 rows=1 loops=67659)
                                                    Index Cond: ((title_id)::text = (uh.title_id)::text)
                                                    Filter: (season_number IS NOT NULL)
                                                    Rows Removed by Filter: 0
                                                --> Index Scan using title_pkey on title t (cost=0.43..0.56 rows=1 width=30) (actual time=0.015..0.015 rows=1 loops=48507)
                                                    Index Cond: ((id)::text = (te.parent_title_id)::text)
                                                    Filter: ((title_type)::text = 'tvSeries'::text)
                                                    Rows Removed by Filter: 0
                                                --> Index Only Scan using title_pkey on title t2 (cost=0.43..0.56 rows=1 width=10) (actual time=0.018..0.018 rows=1 loops=46327)
                                                    Index Cond: (id = (te.title_id)::text)
                                                    Heap Fetches: 46327
                                            --> Index Scan using users_pkey on users u (cost=0.29..1.14 rows=1 width=4) (actual time=0.003..0.003 rows=1 loops=46327)
                                                Index Cond: (id = uh.user_id)
                                                Filter: ((country_code)::text <> ALL ('{US,GB}'::text[]))
                                                Rows Removed by Filter: 0
Planning Time: 3.939 ms
JIT:
  Functions: 127
  Options: Inlining false, Optimization false, Expressions true, Deforming true
Timing: Generation 11.082 ms, Inlining 0.000 ms, Optimization 3.206 ms, Emission 72.319 ms, Total 86.607 ms
Execution Time: 2292.029 ms
(51 rows)

```

Figura 6: Plano de execução da *query* 2 com os índices.

Posteriormente, encontramos outro problema na interrogação que estaria a minimizar o desempenho da mesma, que correspondeu à enorme quantidade de junções entre as tabelas, visto que no total são realizadas cinco operações *JOIN*. Desta forma, decidiu-se criar uma vista materializada que agrega todas estas, já que as colunas utilizadas são, em princípio, imutáveis, dado que correspondem aos id's. Caso estas não sejam imutáveis, consideramos que seriam alteradas um número reduzido de vezes, permanecendo na maior do tempo inalteradas. E apesar do custo associado a essas alterações, existira, na mesma, uma diminuição no custo das operações, uma vez que estaríamos apenas a aceder à tabela resultante da junção das várias tabelas em vez de a criar sempre que a *query* fosse executada. Ajustando, assim, a consulta à vista materializada, obtemos a seguinte *query*:

```

CREATE MATERIALIZED VIEW results AS
SELECT t.id, t.primary_title, tg.genres, te.season_number, uh.last_seen,
u.country_code, t.title_type
FROM title t
JOIN titleEpisode te ON te.parent_title_id = t.id
JOIN title t2 ON t2.id = te.title_id
JOIN userHistory uh ON uh.title_id = t2.id
JOIN users u ON u.id = uh.user_id
JOIN titleGenresGroup tg ON tg.title_id = t.id;

```

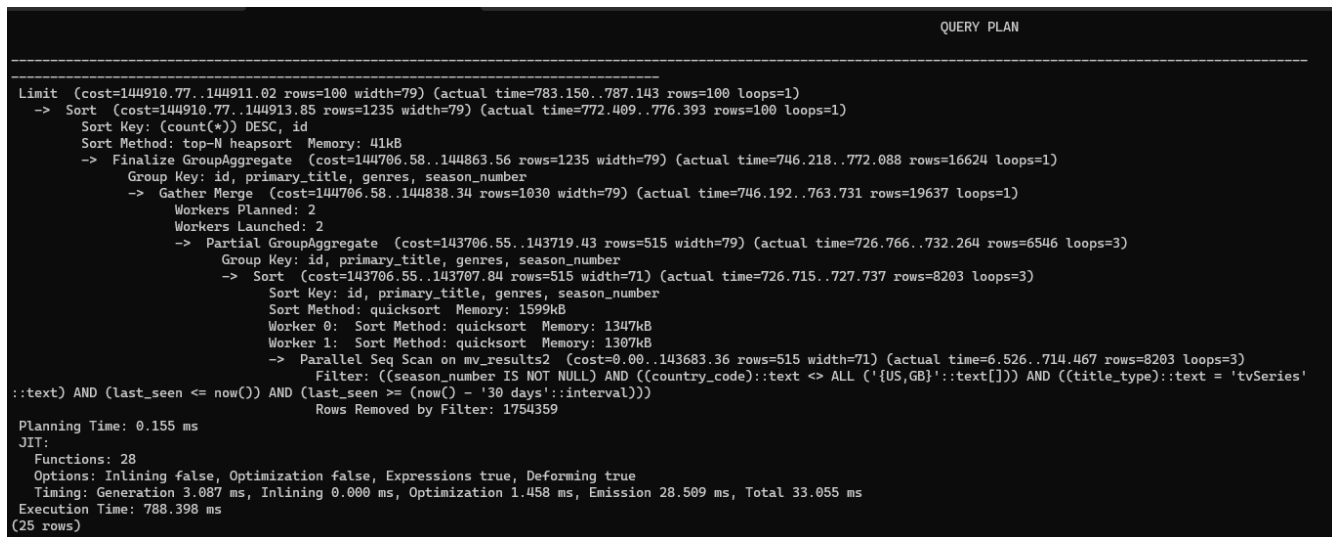


```

SELECT id, primary_title, genres, season_number, COUNT(*) AS views
FROM results
WHERE title_type = 'tvSeries'
      AND last_seen BETWEEN NOW() - INTERVAL '30 days' AND NOW()
      AND season_number IS NOT NULL
      AND country_code NOT IN ('US', 'GB')
GROUP BY id, primary_title, genres, season_number
ORDER BY views DESC, id
LIMIT 100;

```

Executando, novamente, a query, foi possível voltar a observar um impacto significativo no tempo de execução, dado que este diminuindo, em média, para 792,478 ms.



```

QUERY PLAN

Limit (cost=144910.77..144911.02 rows=100 width=79) (actual time=783.150..787.143 rows=100 loops=1)
--> Sort (cost=144910.77..144913.85 rows=1235 width=79) (actual time=772.409..776.393 rows=100 loops=1)
    Sort Key: (count(*)) DESC, id
    Sort Method: top-N heapsort  Memory: 41kB
--> Finalize GroupAggregate (cost=144706.58..144863.56 rows=1235 width=79) (actual time=746.218..772.088 rows=16624 loops=1)
    Group Key: id, primary_title, genres, season_number
--> Gather Merge (cost=144706.58..144838.34 rows=1030 width=79) (actual time=746.192..763.731 rows=19637 loops=1)
    Workers Planned: 2
    Workers Launched: 2
--> Partial GroupAggregate (cost=143706.55..143719.43 rows=515 width=79) (actual time=726.766..732.264 rows=6546 loops=3)
    Group Key: id, primary_title, genres, season_number
--> Sort (cost=143706.55..143707.84 rows=515 width=71) (actual time=726.715..727.737 rows=8203 loops=3)
    Sort Key: id, primary_title, genres, season_number
    Sort Method: quicksort  Memory: 1599kB
    Worker 0: Sort Method: quicksort  Memory: 1347kB
    Worker 1: Sort Method: quicksort  Memory: 1307kB
--> Parallel Seq Scan on mv_results2 (cost=0.00..143683.36 rows=515 width=71) (actual time=6.526..714.467 rows=8203 loops=3)
    Filter: ((season_number IS NOT NULL) AND ((country_code)::text <> ALL ('{US,GB} '::text[])) AND ((title_type)::text = 'tvSeries'
::text) AND (last_seen <= now()) AND (last_seen >= (now() - '30 days'::interval)))
    Rows Removed by Filter: 1754359
Planning Time: 0.155 ms
JIT:
  Functions: 28
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 3.087 ms, Inlining 0.000 ms, Optimization 1.458 ms, Emission 28.509 ms, Total 33.055 ms
Execution Time: 788.398 ms
(25 rows)

```

Figura 7: Plano de execução da *query* 2 com a vista materializada.

Na figura 7, relativa ao plano de execução da interrogação após a criação da vista materializada, é possível observar que esta é filtrada pelas colunas *season_number*, *country_code*, *title_type* e *last_seen*. Consequentemente, foram criados índices para cada uma destas colunas na vista materializada, que irão desempenhar o mesmo papel que os primeiros índices criados, anteriormente, para cada uma das tabelas correspondentes. Permitindo, assim, uma identificação mais rápida das mesmas, dado que é possível selecionar, de forma mais célere, a percentagem de linhas da vista.

```

CREATE INDEX idx_mv_results_last_seen ON mv_results (last_seen);
CREATE INDEX idx_mv_results_country_code ON mv_results (country_code);
CREATE INDEX idx_mv_results_title_type ON mv_results (title_type);

```

Esta modificação possibilitou a otimização da *performance* da *query*, reduzindo o tempo de execução, em média, para 225,541 ms.

```

QUERY PLAN
-----
Limit (cost=4367.94..4368.19 rows=100 width=79) (actual time=223.295..223.309 rows=100 loops=1)
-> Sort (cost=4367.94..4370.19 rows=899 width=79) (actual time=223.293..223.299 rows=100 loops=1)
    Sort Key: (count(*)) DESC, id
    Sort Method: top-N heapsort  Memory: 49kB
-> HashAggregate (cost=4324.59..4333.58 rows=899 width=79) (actual time=213.695..219.147 rows=16617 loops=1)
    Group Key: id, primary_title, genres, season_number
    Batches: 1  Memory Usage: 3865kB
-> Bitmap Heap Scan on mv_results2 (cost=24.43..4313.34 rows=900 width=71) (actual time=11.329..188.547 rows=24599 loops=1)
    Recheck Cond: ((last_seen >= (now() - '30 days'::interval)) AND (last_seen <= now()))
    Filter: ((season_number IS NOT NULL) AND ((country_code)::text <> ALL ('{US,GB}'::text[])) AND ((title_type)::text = 'tvSeries'::text))
    Rows Removed by Filter: 8916
    Heap Blocks: exact=27417
-> Bitmap Index Scan on idx_mv_results2_last_seen (cost=0.00..24.20 rows=1176 width=0) (actual time=6.099..6.099 rows=33515 loops=1)
    Index Cond: ((last_seen >= (now() - '30 days'::interval)) AND (last_seen <= now()))

Planning Time: 0.309 ms
Execution Time: 223.376 ms
(16 rows)

```

Figura 8: Plano de execução da *query* 2 com os novos índices.

Por fim, numa última tentativa de maximizar, ainda mais, a *performance* da interrogação, experimentou-se colocar grande parte da mesma numa vista materializada, deixando apenas as operações *GROUP BY*, *ORDER BY* e *LIMIT* na *query* inicial.

```

CREATE MATERIALIZED VIEW results AS
SELECT t.id, t.primary_title, tg.genres, te.season_number
FROM title t
JOIN titleEpisode te ON te.parent_title_id = t.id
JOIN title t2 ON t2.id = te.title_id
JOIN userHistory uh ON uh.title_id = t2.id
JOIN users u ON u.id = uh.user_id
JOIN titleGenresGroup tg ON tg.title_id = t.id
WHERE t.title_type = 'tvSeries'
      AND uh.last_seen BETWEEN NOW() - INTERVAL '30 days' AND NOW()
      AND te.season_number IS NOT NULL
      AND u.country_code NOT IN ('US', 'GB');

SELECT id, primary_title, genres, season_number, COUNT(*) AS views
FROM results
GROUP BY id, primary_title, genres, season_number
ORDER BY views DESC, id
LIMIT 100;

```

No entanto, embora o tempo de execução tenha reduzido significativamente, em média, para 27,230 ms, como podemos observar pela figura seguinte, esta solução compromete significativamente a flexibilidade da *query*, pelo que o grupo concluiu que não se tratava de uma boa solução, de acordo com os termos definidos inicialmente.


```

QUERY PLAN
-----
Limit  (cost=1533.57..1533.82 rows=100 width=81) (actual time=26.740..26.754 rows=100 loops=1)
-> Sort  (cost=1533.57..1567.77 rows=13680 width=81) (actual time=26.739..26.745 rows=100 loops=1)
    Sort Key: (count(*)) DESC, id
    Sort Method: top-N heapsort  Memory: 49kB
-> HashAggregate  (cost=873.93..1010.73 rows=13680 width=81) (actual time=17.445..22.594 rows=16629 loops=1)
    Group Key: id, primary_title, genres, season_number
    Batches: 1  Memory Usage: 3865kB
-> Seq Scan on mv_results  (cost=0.00..566.19 rows=24619 width=73) (actual time=0.006..1.891 rows=24619 loops=1)
Planning Time: 0.127 ms
Execution Time: 26.873 ms
(10 rows)

```

Figura 9: Plano de execução da *query* 2 com a segunda versão da vista materializada.

2.2.3 Interrogação analítica 3

```

SELECT n.id,
       n.primary_name,
       date_part('year', NOW())::int - n.birth_year AS age,
       count(*) AS roles
FROM name n
JOIN titlePrincipals tp ON tp.name_id = n.id
JOIN titlePrincipalsCharacters tpc ON tpc.title_id = tp.title_id
    AND tpc.name_id = tp.name_id
JOIN category c ON c.id = tp.category_id
JOIN title t ON t.id = tp.title_id
LEFT JOIN titleEpisode te ON te.title_id = tp.title_id
WHERE t.start_year >= date_part('year', NOW())::int - 10
    AND c.name = 'actress'
    AND n.death_year IS NULL
    AND t.title_type IN (
        'movie', 'tvSeries', 'tvMiniSeries', 'tvMovie'
    )
    AND te.title_id IS NULL
GROUP BY n.id
ORDER BY roles DESC
LIMIT 100;

```

Esta última interrogação analítica tem como propósito calcular o top das 100 atrizes que estiverem mais ativas, nos últimos dez anos. Para tal, é necessário ter, ainda, em conta que estas devem estar atualmente vivas e devem ter participado tanto em filmes como em séries. O resultado obtido é ordenado pelo número de trabalhos efetuados, sendo que este é calculado com base na participação.

Inicialmente, começou-se pela utilização do comando **EXPLAIN ANALYZE**, tal como já foi realizado anteriormente, para obter o plano de execução gerado pelo *PostgreSQL* e o tempo de execução da *query* original. Este valor correspondeu, em média, a 66354.15 ms.

```

QUERY PLAN
Limit (cost=971880.82..971880.82 rows=1 width=36) (actual time=60213.779..60388.237 rows=100 loops=1)
-> Sort (cost=971880.82..971880.82 rows=1 width=36) (actual time=59876.770..60851.219 rows=100 loops=1)
    Sort Key: (count(*)) DESC
    Sort Method: top-N heapsort  Memory: 38kB
-> GroupAggregate (cost=971880.78..971880.81 rows=1 width=36) (actual time=59571.672..60813.054 rows=183935 loops=1)
    Group Key: n.id
-> Sort (cost=971880.78..971880.78 rows=1 width=28) (actual time=59571.610..59892.320 rows=396534 loops=1)
    Sort Key: n.id
    Sort Method: external merge  Disk: 14328kB
-> Nested Loop (cost=139566.22..971880.77 rows=1 width=28) (actual time=5822.358..59238.172 rows=396534 loops=1)
-> Nested Loop (cost=139565.66..971878.01 rows=1 width=58) (actual time=5822.308..58939.298 rows=446579 loops=1)
-> Nested Loop (cost=139565.22..971872.70 rows=1 width=40) (actual time=5778.402..58689.658 rows=234316 loops=1)
-> Gather (cost=139564.66..971869.41 rows=1 width=20) (actual time=5778.194..6378.301 rows=2628365 loops=1)
    Workers Planned: 2
    Workers Launched: 2
-> Parallel Hash Anti Join (cost=138564.66..970869.31 rows=1 width=20) (actual time=5765.859..6792.552 rows=876122 loops=3)
    Hash Cond: ((tp.title_id)::text = (te.title_id)::text)
-> Hash Join (cost=1.16..782219.34 rows=1942796 width=20) (actual time=0.181..3765.278 rows=3188598 loops=3)
    Hash Cond: (tp.category_id = c.id)
-> Parallel Seq Scan on titleprincipals tp (cost=0.00..699406.50 rows=2331350 width=24) (actual time=0.040..1911.174 rows=1865021 loops=3)
-> Hash (cost=1.15..1.15 rows=1 width=4) (actual time=0.056..0.058 rows=1 loops=3)
    Buckets: 1024 Batches: 1 Memory Usage: 9kB
-> Seq Scan on category c (cost=0.00..1.15 rows=1 width=4) (actual time=0.049..0.050 rows=1 loops=3)
    Filter: ((name)::text = 'actress'::text)
    Rows Removed by Filter: 11
-> Parallel Hash (cost=84538.89..84538.89 rows=3107889 width=10) (actual time=1074.165..1074.166 rows=2486311 loops=3)
    Buckets: 131072 Batches: 128 Memory Usage: 388848
-> Parallel Seq Scan on titleepisode te (cost=0.00..84538.89 rows=3107889 width=10) (actual time=130.728..588.514 rows=2486311 loops=3)
-> Index Scan using name_pkey on name n (cost=0.56..4.38 rows=1 width=28) (actual time=0.012..0.012 rows=1 loops=2628365)
    Index Cond: ((id)::text = (tp.name_id)::text)
    Filter: (death_year IS NULL)
    Rows Removed by Filter: 0
-> Index Scan using title_pkey on title t (cost=0.43..4.23 rows=1 width=10) (actual time=0.007..0.007 rows=0 loops=234316)
    Index Cond: ((id)::text = (tp.title_id)::text)
    Filter: (((title_type)::text = ANY ('{movie,tvSeries,tvMiniSeries,tvMovie}'::text[])) AND (start_year >= ((date_part('year'::text, now()))::integer - 10)))
    Rows Removed by Filter: 1
-> Index Only Scan using titleprincipalscharacters_pkey on titleprincipalscharacters tpc (cost=0.56..2.74 rows=1 width=20) (actual time=0.007..0.007 rows=1 loops=446579)
    Index Cond: ((title_id = (tp.title_id)::text) AND (name_id = (tp.name_id)::text))
    Heap Fetches: 0
Planning Time: 5.574 ms
JIT:
Functions: 88
Options: Inlining true, Optimization true, Expressions true, Deforming true
Timing: Generation 5.381 ms, Inlining 165.697 ms, Optimization 331.883 ms, Emission 231.624 ms, Total 734.785 ms
Execution Time: 60396.555 ms
(45 rows)

```

Figura 10: Plano de execução da *query* 3 original.

Tendo sido analisado o plano e a *query*, apercebemo-nos que a mesma filtrava as linhas da coluna *start_year* da tabela *title*, de forma a ter apenas as datas superiores a um determinado valor. Mais ainda, nesta tabela, são também filtradas as linhas da coluna *title_type* cujo título deve corresponde a um(s) certo género(s). São, ainda, filtradas as linhas das tabelas *category*, *name*, *titleEpisode* de modo a obter as mesmas com os valores pretendidos. De forma semelhante, esta *query* seleciona também as colunas *id* das tabelas *titlePrincipals*, *titlePrincipalsCharacters*, *category*, *titleEpisode* e *title*. Por conseguinte, o grupo decidiu, pelas mesmas razões apresentadas anteriormente, retomar a estratégia de criação de índices para essas operações. Dessa forma, seria possível acelerar a identificação e reduzir o acesso às linhas pretendidas na *heap*.

```

CREATE INDEX idx_titlePrincipals_name_id ON titlePrincipals (name_id);
CREATE INDEX idx_titlePrincipalsCharacters_title_id_name_id ON
titlePrincipalsCharacters (title_id, name_id);
CREATE INDEX idx_category_id ON category (id);
CREATE INDEX idx_title_id ON title (id);
CREATE INDEX idx_titleEpisode_title_id ON titleEpisode (title_id);
CREATE INDEX idx_start_year ON title (start_year);
CREATE INDEX idx_title_type ON title (title_type);
CREATE INDEX idx_name ON category (name);

```

Ao executar novamente a *query*, foi possível observar um impacto positivo na *performance* da mesma, reduzindo significativamente o tempo de execução para 16293.91 ms, em média, como podemos ver pela figura 6.

```

QUERY PLAN
Limit (cost=1330739.64..1330739.64 rows=1 width=36) (actual time=16309.404..16419.823 rows=100 loops=1)
--> Sort (cost=1330739.64..1330739.64 rows=1 width=36) (actual time=15884.942..15995.352 rows=100 loops=1)
    Sort Key: (count(*)) DESC
    Sort Method: top-N heapsort Memory: 37kB
--> GroupAggregate (cost=1330739.60..1330739.63 rows=1 width=36) (actual time=15733.168..15969.739 rows=124740 loops=1)
    Group Key: n.id, (((date_part('year',::text, now()))::integer - n.birth_year))
--> Sort (cost=1330739.60..1330739.60 rows=1 width=28) (actual time=15733.651..15967.699 rows=217104 loops=1)
    Sort Key: n.id, (((date_part('year',::text, now()))::integer - n.birth_year))
    Sort Method: external merge Disk: 7856kB
--> Nested Loop (cost=10855155.76..1330739.59 rows=1 width=28) (actual time=8753.233..15553.698 rows=217104 loops=1)
--> Nested Loop (cost=10855155.28..1330736.91 rows=1 width=58) (actual time=8753.176..13584.385 rows=267165 loops=1)
--> Hash Right Join (cost=10855154.64..1330732.54 rows=1 width=30) (actual time=8753.079..10174.653 rows=268647 loops=1)
    Hash Cond: ((tp.parent_title_id)::text = (tp.title_id)::text)
    Filter: (tp.title_id IS NULL)
    Rows Removed by Filter: 5358996
--> Seq Scan on titleepisode te (cost=0.00..120809.33 rows=7458933 width=20) (actual time=0.061..1000.155 rows=7458933 loops=1)
--> Hash (cost=1081098.57..1081098.57 rows=209765 width=30) (actual time=5998.976..6109.372 rows=448662 loops=1)
    Buckets: 65536 (originally 65536) Batches: 8 (originally 4) Memory Usage: 3973kB
--> Gather (cost=219136.38..1081098.57 rows=209765 width=30) (actual time=5481.073..6001.034 rows=448662 loops=1)
    Workers Planned: 2
    Workers Launched: 2
--> Parallel Hash Join (cost=218136.38..1029122.07 rows=87402 width=30) (actual time=5454.978..5877.606 rows=109561 loops=3)
    Hash Cond: ((tp.title_id)::text = (te.id)::text)
--> Hash Join (cost=1.16..702219.02 rows=1942794 width=20) (actual time=0.245..3689.579 rows=3188598 loops=3)
    Hash Cond: (tp.category_id = c.id)
--> Parallel Seq Scan on titleprincipals tp (cost=0.00..699406.27 rows=23313527 width=24) (actual time=0.035..1827.022 rows=18658821 loops=3)
--> Hash (cost=1.15..1.15 rows=1 width=4) (actual time=0.047..0.049 rows=1 loops=3)
    Buckets: 1624 Batches: 1 Memory Usage: 94kB
--> Seq Scan on category c (cost=0.00..1.15 rows=1 width=4) (actual time=0.041..0.041 rows=1 loops=3)
    Filter: ((name)::text = 'actress'::text)
    Rows Removed by Filter: 11
--> Parallel Hash (cost=214933.43..214933.43 rows=184143 width=10) (actual time=925.259..925.261 rows=117458 loops=3)
    Buckets: 131072 Batches: 8 Memory Usage: 3104kB
--> Parallel Bitmap Heap Scan on title t (cost=1490.34..214933.43 rows=184143 width=10) (actual time=241.960..892.957 rows=117458 loops=3)
    Recheck Cond: ((title_type)::text = ANY ('{movie,tvSeries,tvMiniSeries,tvMovie}'::text[]))
    Rows Removed by Index Recheck: 1574186
    Filter: (start_year >= ((date_part('year',::text, now()))::integer - 10))
    Rows Removed by Filter: 241394
    Heap Blocks: exact=17885 lossy=30033
--> Bitmap Index Scan on idx_title_type (cost=0.00..11379.85 rows=1057248 width=0) (actual time=67.573..67.574 rows=1076558 loops=1)
    Index Cond: ((title_type)::text = ANY ('{movie,tvSeries,tvMiniSeries,tvMovie}'::text[]))
--> Index Scan using name_pkey on name n (cost=0.56..4.38 rows=1 width=28) (actual time=0.012..0.012 rows=1 loops=268647)
    Index Cond: ((id)::text = (tp.name_id)::text)
    Filter: (death_year IS NULL)
    Rows Removed by Filter: 0
--> Index Only Scan using idx_titleprincipalscharacters_title_id_name_id on titleprincipalscharacters tpc (cost=0.56..2.65 rows=1 width=20) (actual time=0.007..0.007 rows=1 loops=267165)
    Index Cond: ((title_id = (tp.title_id)::text) AND (name_id = (tp.name_id)::text))
    Heap Fetches: 0

Planning Time: 7.139 ms
JIT:
  Functions: 97
  Options: Inlining true, Optimization true, Expressions true, Deforming true
Timing: Generation 6.263 ms, Inlining 109.992 ms, Optimization 470.722 ms, Emission 320.011 ms, Total 906.989 ms
Execution Time: 16425.838 ms
(54 rows)

```

Figura 11: Plano de execução da *query* 3 com os índices.

Finalmente, voltamos a aplicar a estratégia de criação de uma tabela. Esta decisão, enquanto grupo, deveu-se ao facto de, no período de execução da interrogação, esta realizar diversas operações *JOIN*, o que minimiza o seu desempenho. Como tal, ponderamos entre a criação de uma vista materializada e de uma tabela que armazenasse a informação referente aos atores/atrizes, nomeadamente, os seus nomes, os títulos aos quais estão associados e as informações relacionadas com os mesmos. Como já foi mencionado, um problema da vista materializada corresponde à necessidade de ser recalculada sempre que pretendemos inserir uma nova informação na mesma. Deste modo, o grupo optou pela criação de uma tabela, que fornece como solução uma operação menos "pesada" do que recálculo da vista inteira. Isto é, com a tabela é apenas preciso realizar uma operação de *INSERT* quando quiséssemos adicionar algo, assim, ajustando a interrogação analítica à tabela, obtemos a seguinte *query*:

```

CREATE TABLE actors_roles AS
SELECT n.id,
       n.primary_name,
       te.title_id,
       n.birth_year,
       t.start_year,
       n.death_year,
       t.title_type,
       c.name AS category_name
FROM name n
JOIN titlePrincipals tp ON tp.name_id = n.id
JOIN titlePrincipalsCharacters tpc ON tpc.title_id = tp.title_id
AND tpc.name_id = tp.name_id
JOIN category c ON c.id = tp.category_id

```

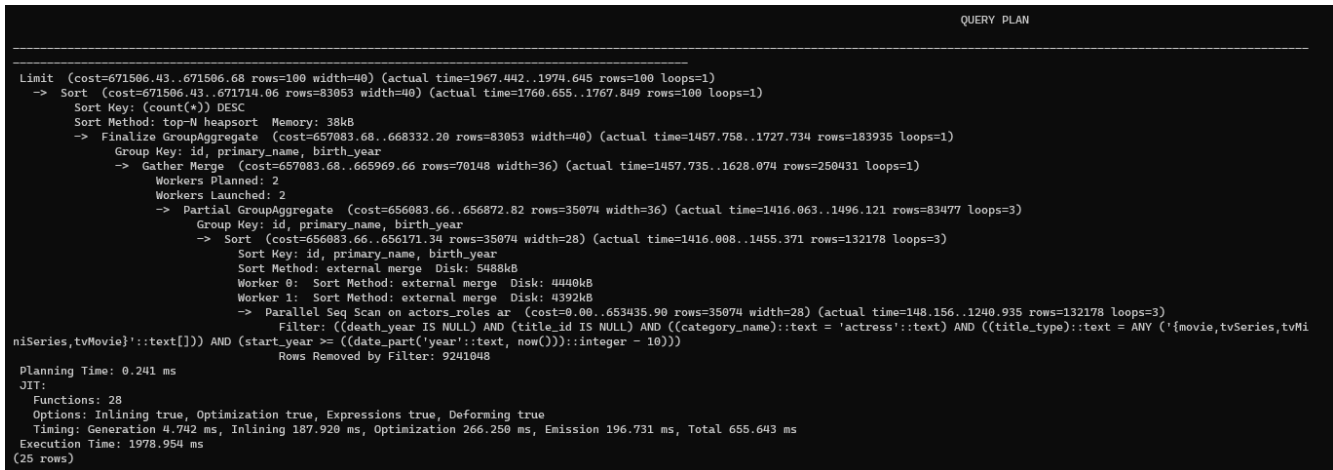
```

JOIN title t ON t.id = tp.title_id
LEFT JOIN titleEpisode te ON te.title_id = tp.title_id

SELECT id, ar.primary_name, date_part('year', NOW())::int - ar.birth_year AS age,
COUNT(*) AS roles
FROM actors_roles ar
WHERE ar.start_year >= date_part('year', NOW())::int - 10
      AND ar.category_name = 'actress'
      AND ar.death_year IS NULL
      AND ar.title_type IN ('movie', 'tvSeries', 'tvMiniSeries', 'tvMovie')
      AND ar.title_id IS NULL
GROUP BY ar.id, ar.primary_name, ar.birth_year
ORDER BY roles DESC
LIMIT 100;

```

Tal como esperado, a estratégia utilizada evidenciou um impacto bastante positivo no desempenho da *query*, uma vez que durante a execução da mesma, esta passaria apenas a aceder aos resultados calculados, resultando numa redução significativa do tempo de execução, que se encontra, em média, nos 2058.19 ms.



```

QUERY PLAN

Limit (cost=671506.43..671506.68 rows=100 width=40) (actual time=1967.442..1974.645 rows=100 loops=1)
  -> Sort (cost=671506.43..671714.06 rows=83053 width=40) (actual time=1760.655..1767.849 rows=100 loops=1)
    Sort Key: (count(*)) DESC
    Sort Method: top-N heapsort  Memory: 38kB
    -> Finalize GroupAggregate (cost=657083.68..668332.20 rows=83053 width=40) (actual time=1457.758..1727.734 rows=183935 loops=1)
      Group Key: id, primary_name, birth_year
      -> Gather Merge (cost=657083.68..665969.66 rows=70148 width=36) (actual time=1457.735..1628.074 rows=250431 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial GroupAggregate (cost=656083.66..656872.82 rows=35074 width=36) (actual time=1416.063..1496.121 rows=83477 loops=3)
          Group Key: id, primary_name, birth_year
          -> Sort (cost=656083.66..656171.34 rows=35074 width=28) (actual time=1416.008..1455.371 rows=132178 loops=3)
            Sort Key: id, primary_name, birth_year
            Sort Method: external merge  Disk: 5488kB
            Worker 0: Sort Method: external merge  Disk: 4440kB
            Worker 1: Sort Method: external merge  Disk: 4392kB
            -> Parallel Seq Scan on actors_roles ar (cost=0.00..653435.90 rows=35074 width=28) (actual time=148.156..1240.935 rows=132178 loops=3)
              Filter: ((death_year IS NULL) AND (title_id IS NULL) AND ((category_name)::text = 'actress'::text) AND ((title_type)::text = ANY ('{movie,tvSeries,tvMiniSeries,tvMovie}'::text[])) AND (start_year >= ((date_part('year'::text, now()))::integer - 10)))
              Rows Removed by Filter: 9241048
          Rows Removed by Filter: 9241048
        Planning Time: 0.241 ms
        JIT:
          Functions: 28
          Options: Inlining true, Optimization true, Expressions true, Deforming true
          Timing: Generation 4.742 ms, Inlining 187.920 ms, Optimization 266.250 ms, Emission 196.731 ms, Total 655.643 ms
        Execution Time: 1978.954 ms
        (25 rows)

```

Figura 12: Plano de execução da *query* 3 com a tabela *actors_roles*.

Denotamos que, nas interrogações analíticas 2 e 3, falta a implementação dos *triggers* para atualizar as tabelas e as vistas materializadas criadas em resposta a determinadas ações ou eventos específicos na base de dados. No caso das tabelas, os *triggers* atualizariam de forma incremental. Já no caso das vistas materializadas, estes atualizariam num período regular de tempo, por exemplo, todos os dias às 5 da manhã.

3 Otimização das Interrogações Analíticas no *Spark*

Para além dos objetivos referidos na secções anteriores, o trabalho prático apresenta mais, sendo os objetos de estudo desta secção, a exportação dos dados do *PostgreSQL* para ficheiros *Parquet* e a otimização do desempenho das interrogações analíticas usando o *Spark*.

3.1 *Setup* Automático do *Spark*

Tal como pedido no enunciado, aplicamos uma instalação virtualizada, utilizando o *Docker*, para o *cluster Spark*, recorrendo, para isso, a alguns *scripts* para o seu correto *deployment*. Desta forma, passamos a apresentar cada um destes *scripts* de modo a detalhar o seu comportamento:

- **install.sh** - Responsável pela instalação e configuração básica do *Docker*.
- **sparkStart.sh** - Trata da inicialização do *cluster Spark* e da execução do ficheiro *python* passado como argumento.
- **sparkStop.sh** - Este é responsável pela paragem e destruição do *cluster Spark*.

3.2 Exportação dos dados PostgreSQL

Como primeiro passo no desenvolvimento desta etapa, começamos por exportar os dados presentes no *PostgreSQL* para ficheiros do formato *TSV*. Para isso, recorreremos a um *script* desenvolvido pelo grupo, *spark_exportDBToTSV.sh*, que automatiza este mesmo processo ao iterar pelas tabelas presentes na base de dados e aplicar o comando para executar o respetivo *download*. Estes ficheiros, como são apenas um passo intermédio para atingir os ficheiros *Parquet* pretendidos, foram eliminados após servirem o seu propósito.

```
psql -U postgres -d abd -Atc "select table_name from information_schema.tables
where table_schema not in ('information_schema', 'pg_catalog') and
table_type = 'BASE TABLE';" | \

while read TABLENAME; do
    echo "exporting $TABLENAME"
    psql -U postgres -d abd -c "\copy (select * from $TABLENAME)
    TO '/home/abd/imdbBench/spark/sparkDB/$TABLENAME.tsv'
    WITH DELIMITER E'\t' CSV HEADER;"
done
```

3.3 Leitura dos Ficheiros TSV

Após a correta exportação dos ficheiros *TSV*, seguiu-se o primeiro passo concreto no *Spark*, que correspondeu à transformação dos ficheiros *TSV* para ficheiros *Parquet*. Para isso, recorreremos aos métodos do *Spark*, como mostra o seguinte *snippet* de código:

```
for file in tableList:
    print("Exporting "+file+" to parquet")
    tmpTable = spark.read.csv("/spark/sparkDB/"+file, header=True, inferSchema=True, sep="\t")
    tmpTable.write.parquet("/spark/parquetDB/"+file+".parquet")
```

Este excerto de código está presente no ficheiro, *spark_app_exportToParquet.py*, e após a sua execução, os ficheiros foram armazenados numa diretoria que, por sua vez, ficou integrada na diretoria */spark* e foram, ainda, colocados no *Bucket* de maneira a agilizar o acesso aos mesmos.

3.4 Otimizações

Nesta secção, apresentaremos as várias otimizações realizadas a todas as interrogações analíticas, detalhando o processo de cada uma e explicando os resultados obtidos. Estas serão executadas no ambiente *Spark* através do método `spark.sql("... ")`, de forma a facilitar o processo de desenvolvimento da *query*. No entanto, foram aplicadas certas transformações no código SQL das *queries*, visto que alguma destas tinham problemas de *syntax* dentro do interpretador do **Spark**.

3.4.1 Modificações da Interrogação Analítica 1

Tal como referido anteriormente, esta *query* é responsável pela ordenação dos top 10 filmes de todas as décadas a partir de 1980 (inclusive) de acordo com o seu *rating*. Desta forma, começamos por executar a *query* diretamente, obtendo os seguintes erros e aplicando as respetivas soluções:

- **Cast para INT** - A compilação não aceitava esta forma de executar *cast* a um valor, então substituímos pela função *CAST* do *SQL*.
- **Atributos do SELECT** - Alguns atributos presentes na operação *SELECT* não se encontravam na operação *GROUP BY*, desta forma, incluímos os mesmos.
- **ORDER BY** - Alteramos a ordem da operação *ORDER BY* sendo esta realizada no final, após a filtragem pelo *top 10*, passando a ordenar por década e *rank* crescente (uma vez que obtínhamos alguns problemas de ordenação pelo *rating* devido a possíveis arredondamentos).

Após todas estas alterações, a *query* ficou com o seguinte aspeto:

```
SELECT * FROM (
  SELECT t.id,
    left(t.primary_title, 30),
    CAST((CAST(t.start_year AS BIGINT) DIV 10) * 10 AS INT) AS decade,
    avg(uh.rating) AS rating,
    rank() over (
      PARTITION by CAST((CAST(t.start_year AS BIGINT) DIV 10) * 10 AS INT)
      ORDER BY avg(uh.rating) DESC, t.id
    ) AS rank
  FROM title t
  JOIN userHistory uh ON uh.title_id = t.id
  WHERE t.title_type = 'movie'
    AND CAST((CAST(t.start_year AS BIGINT) DIV 10) * 10 AS INT) >= 1980
    AND t.id IN (
      SELECT title_id
      FROM titleGenre tg
      JOIN genre g ON g.id = tg.genre_id
      WHERE g.name IN (
        'Drama'
      )
    )
)
```

```

    )
    AND t.id IN (
        SELECT title_id
        FROM titleAkas
        WHERE region IN (
            'US', 'GB', 'ES', 'DE', 'FR', 'PT'
        )
    )
)
GROUP BY t.id, t.primary_title, t.start_year
HAVING count(uh.rating) >= 3
) t_
WHERE rank <= 10
ORDER BY decade, rank ASC

```

3.4.2 Modificações da Interrogação Analítica 2

A *query* 2 tem como objetivo identificar o top 100 títulos de séries de TV com maior número de visualizações nos últimos 30 dias, excluindo os utilizadores dos países com códigos *US* (Estados Unidos) e *GB* (Reino Unido). No que diz respeito à sua compilação, esta *query* encontrava-se bem definida, pelo que não foi necessário alterar o seu código.

3.4.3 Modificações da Interrogação Analítica 3

Esta *query* tem como objetivo apresentar o top 100 das atrizes com o maior número de *roles* na sua carreira. Desta forma, tal como nas interrogações anteriores, começamos por executar a *query* diretamente, obtendo os seguintes erros e aplicando as respetivas soluções:

- **Cast para INT** - A compilação não aceitava esta forma de executar *cast* a um valor, então substituímos pela função *CAST* do *SQL*.
- **Atributos do SELECT** - Alguns atributos presentes na operação *SELECT* não se encontravam na operação *GROUP BY*, desta forma, incluímos os mesmos.

Assim sendo, obtivemos o seguinte código final da *query*:

```

SELECT n.id,
       n.primary_name,
       CAST(date_part('year', CURRENT_DATE()) AS INT) - n.birth_year AS age,
       count(*) AS roles
FROM name n
JOIN titlePrincipals tp ON tp.name_id = n.id
JOIN titlePrincipalsCharacters tpc ON tpc.title_id = tp.title_id
    AND tpc.name_id = tp.name_id
JOIN category c ON c.id = tp.category_id
JOIN title t ON t.id = tp.title_id
LEFT JOIN titleEpisode te ON te.title_id = tp.title_id
WHERE t.start_year >= CAST(date_part('year', CURRENT_DATE()) AS INT) - 10
    AND c.name = 'actress'
    AND n.death_year IS NULL
    AND t.title_type IN (

```



```

        'movie', 'tvSeries', 'tvMiniSeries', 'tvMovie'
    )
    AND te.title_id IS NULL
GROUP BY n.id, n.primary_name, n.birth_year
ORDER BY roles DESC
LIMIT 100;

```

3.4.4 Método de Otimização

Para alcançar a melhor otimização destas *queries*, primeiramente, modificamos as definições quer do número de *service_workers* do *Spark*, quer dos seus parâmetros individuais de memória e do número de CPU's, de forma a encontrar a melhor combinação destes parâmetros. Desta forma, alteramos os seguintes parâmetros do *Spark*:

- **Service Worker** - Com um maior número de *workers* será possível paralelizar as tarefas do *Spark*, obtendo assim melhores tempos de execução. Começamos por testar com apenas um *service worker*, passando, no seguinte teste, para 3 e, por fim, com 6.
- **Parâmetros Service Worker** - Alocando mais recursos a cada um destes *workers*, estes poderão trabalhar de forma mais rápida. Após os testes relatados acima, testamos, novamente, o caso com um *service worker*, aumentando a sua capacidade em termos de memória (16GB) e de *cores* (8). Para além disto, foi também necessário definir o valor de memória de cada executor, onde colocamos o mesmo valor que no *service worker* (16GB).
- **Shuffle Partitions** - As operações de *shuffle* são computacionalmente pesadas, sendo desejável evitá-las ou otimizá-las. Visto que este parâmetro determina o nível de paralelismo quando são executadas operações de *shuffle*, decidimos utilizá-lo com uma variedade de valores (16, 1024 e 64).
- **Dynamic Partition Pruning** - Com o objetivo de otimizar as *queries JOIN* e minimizar o tempo de leitura de dados na sua fonte, utilizamos o *Pruning* Dinâmico.
- **Storage Fraction** - Determina a fração de memória da *heap* utilizada para o *Spark*. Esta memória é tipicamente utilizada para efetuar o *caching* de dados, sendo por isso benéfica a sua utilização na otimização.
- **Coalesce Partitions** - É uma técnica de otimização que ajusta dinamicamente o número de *shuffle partitions* para as operações de *shuffle*, sendo por isso uma melhor opção que a definição estática deste valor.

Finalmente, experimentámos juntar os parâmetros mais promissores para obter a otimização final.

3.4.5 Resultados Agregados

Nesta secção, iremos apresentar os tempos de execução obtidos (em segundos) através da utilização da função **time()** de Python. É de destacar que cada *query* foi executada cinco vezes para calcular uma média do seu tempo de execução. Passamos a apresentar a tabela com os resultados obtidos na primeira fase de modificação do número de *workers* disponíveis:

Nº <i>Workers</i>	<i>Query 1</i>	<i>Query 2</i>	<i>Query 3</i>
1	84,43	116,21	346,32
3	49,68	61,14	147,02
6	40,79	50,10	118,34

Tabela 1: Testes Realizados para *workers* com 1CPU e 1GB.

Tendo obtido estes resultados, apresentamos, de seguida, a tabela de resultados principal, onde foi utilizado apenas um *worker* com acesso a mais recursos, nomeadamente os valores máximos de memória e de CPU da máquina de teste (16GB e 8CPU). Deste modo, considerando este *worker* como uma "base" de testes, fomos implementando os parâmetros referidos na secção anterior, de modo a obter os melhores tempos de execução possíveis.

Parâmetros/Definições	<i>Query 1</i>	<i>Query 2</i>	<i>Query 3</i>
Base (16G 8CPU)	15,83	24,27	74,81
<i>Shuffle.partitions</i> = 16	13,61	22,16	72,8
<i>Shuffle.partitions</i> = 1024	21,1	27,2	83,93
<i>Shuffle.partitions</i> = 64	15,96	23,51	69,37
<i>DynamicPartitioningPruning.Enable</i>	36,74	25,14	79,39
<i>memory.storageFraction</i> (0.6)	16,61	25,93	79,45
<i>coalescePartitions.enabled</i>	20,87	25,59	78,76

Tabela 2: Testes Realizados com um *worker* com 8CPUs e 16GB.

Através da análise da tabela acima, conseguimos concluir que o *worker* "base" simbolizou uma melhoria de, sensivelmente, 81% (na *query 1*), 79% (na *query 2*) e 78% (na *query 3*), relativamente ao *worker* com 1CPU e 1GB. Para além disto, observamos os efeitos dos vários parâmetros no tempo de execução das *queries*, concluindo que apenas o parâmetro ***shuffle.partition*** obteve resultados satisfatórios, isto é, apenas este parâmetro teve os tempos de execução foram melhores do que os testes "base" para cada *query*. Assim, apresentamos, para cada *query*, o melhor tempo de execução obtido com o valor correspondente de *shuffle.partition* utilizado:

- **Query 1** - Utilizando o *shuffle.partition* com valor **16**, obtivemos um tempo de execução médio de 13,61 segundos.
- **Query 2** - Utilizando o *shuffle.partition* com valor **16**, obtivemos um tempo de execução médio de 22,16 segundos.
- **Query 3** - Utilizando o *shuffle.partition* com valor **64**, obtivemos um tempo de execução médio de 69,37 segundos.

4 Otimização da Carga Transacional

Esta última secção, relativa ao desenvolvimento do projeto, tem como objetivo otimizar o desempenho da carga transacional, avaliando o impacto dos diferentes parâmetros de configurações do *PostgreSQL* na *performance* do *Benchmark*.

Para tal, o grupo concentrou-se na secção de **Write Ahead Logs** (Registros de *log* antes da escrita) da configuração, visto que contém parâmetros importantes para execução de transações sobre a base de dados. Nesta fase do trabalho prático, comparamos os resultados obtidos da realização de testes utilizados para determinar como os diferentes valores das configurações afetavam o desempenho das transações. Posteriormente, esta comparação possibilitou a análise da compatibilidade dos diversos parâmetros, determinando, quais as configurações que permitiriam uma melhor otimização.

Durante o processo, o estado inicial da base de dados era restaurado entre cada teste de modo a assegurar que os resultados obtidos não fossem enviesados. Para além disto, cada parâmetro foi, também, testado de forma isolada, isto é, apenas o parâmetro específico ao teste é modificado, mantendo-se a restante configuração com os valores predefinidos.

A secção de *Write Ahead Logs* é dividida em três subsecções distintas: *settings* (configurações), *checkpoints* e *archiving* (arquivamento). Cada uma dessas subsecções possui configurações específicas que podem ser ajustadas de forma a otimizar a *performance*.

4.1 Número de clientes ótimo

De modo a encontrar o número ótimo de clientes para qual o servidor consegue aguentar a carga, foi corrido o benchmark disponibilizado com vários números de clientes. Os resultados encontram-se na tabela abaixo:

Nº Clientes	Throughput (tx/s)	Response time (s)	Abort rate (%)
1	9.55	0.1021211520052356	0.0
5	35.95	0.13225026806397774	0.0
10	41.45	0.2016629032907117	0.0
15	37.1	0.2633694461401617	0.0
20	25.0	0.23635840971	0.0
25	30.4	0.25574735507401314	0.0
30	21.5	0.2875290146	0.0
35	16.2	0.2851196678487654	0.0
40	6.25	0.200820650824	0.0
45	0.1	0.0627226315	0.0
50	0.1	0.0627226315	0.0

Figura 13:

Com base nos resultados obtidos, podemos considerar que o número de clientes ótimo é 10. Sendo assim, todos os testes a seguir vão utilizar este valor.

4.2 Otimização dos diferentes parâmetros de configuração

De modo a otimizar o desempenho da carga transacional, foi procurada uma configuração em que se obteve os melhores resultados. Foram consultados os seguintes links para obter informações sobre os parâmetros:

- <https://www.postgresql.org/docs/current/wal-configuration.html>
- <https://www.postgresql.org/docs/current/runtime-config-wal.html>

Os vários parâmetros testados encontram-se descritos a seguir.

4.3 Settings

4.3.1 Parâmetro *fsync*

Quando esta opção está ativa, como é predefinição do *PostgreSQL*, o seu servidor verifica se as atualizações estão a escritas fisicamente escritas no disco, garantindo que a base de dados consegue recuperar para um estado consistente em caso de falha. Contudo, isto resulta numa grande penalização no que diz respeito à performance, uma vez que sempre que se realiza um *commit* de uma transação, o *PostgreSQL* necessita de esperar pelo sistema operativo, mais especificamente, que este dê *flush* do *write ahead log* para o disco.

Desativando este parâmetro, o servidor deixa de fazer estas verificações, limitando-se a escrever para o disco logo que possível, o que pode resultar num aumento significativo de performance. Contudo, é bastante perigoso, uma vez que pode haver perda ou corrupção de dados em caso de falha.

Valor	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
<i>on</i>	32.6	0.259797303648773	0.0
<i>off</i>	30.35	0.298724979339374	0.0

Tabela 3: Valores obtidos com a mudança do parâmetro *fsync*

É possível verificar que houve uma pequena melhoria em termos de tempo de execução, apesar do aumento do débito associado. Tendo sido decidido a configuração *on*.

4.3.2 Parâmetro *synchronous_commit*

O *synchronous_commit* especifica o tipo de processamento de *WAL* que deve ser realizado antes do servidor da base de dados retornar um indicador de sucesso ao cliente, regulando o nível de sincronização entre o cliente e as escritas no disco.

A desativação deste parâmetro implica a existência de um intervalo de tempo entre o momento que o cliente recebe a confirmação e o que o resultado é escrito, efetivamente, no disco.

Valor	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
<i>on</i>	33.4	0.29891920653293413	0.0
<i>off</i>	29.25	0.29119022105641024	0.0

Tabela 4: Valores obtidos com a mudança do parâmetro *synchronous_commit*.

Através da observação da tabela acima, é possível verificar que a ativação deste parâmetros não trouxe nenhuma melhoria em termos de débito e do tempo de resposta. Tendo em conta todo o panorama geral, decidimos, enquanto grupo, utilizar a configuração *on* deste parâmetro.

4.3.3 Parâmetro *WAL_sync_method*

O método associado a este parâmetro é utilizado para escrever as atualizações de *WAL* para o disco. No entanto, é de destacar que este só é relevante caso esteja ativo com o valor *fsync*.

Valor	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
<i>fdatasync</i>	28.05	0.32263579576827095	0.0
<i>fsync</i>	40.1	0.22763191649501247	0.0
<i>open_datasync</i>	38.95	0.18805371827086007	0.0
<i>open_sync</i>	36.15	0.262925570142462	0.0

Tabela 5: Valores obtidos com a mudança do parâmetro *WAL_sync_method*.

Pela observação dos diversos resultados obtidos, é possível concluirmos que o *fsync* trouxe o maior equilíbrio entre as diversas métricas, uma vez que o aumento do débito está aliado a um menor tempo de resposta, pelo que decidimos ativar este valor.

4.3.4 Parâmetro *full_page_writes*

A ativação deste parâmetro faz com que o servidor *PostgreSQL* escreva todo o conteúdo existente em cada página do disco na *WAL* durante a primeira alteração da página em questão, a seguir a um *checkpoint*. Apesar da sua desativação possibilitar uma melhor *performance*, também aumenta o risco de perda ou de corrupção dos dados, em caso de falha do sistema.

Valor	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
on	30.15	0.3516240157097844	0.0
off	36.05	0.22205695236893203	0.0

Tabela 6: Valores obtidos com a mudança do parâmetro *full_page_writes*.

Tendo em conta os valores observados e a função deste parâmetro, decidimos desligar este parâmetro.

4.3.5 Parâmetro *commit_delay*

Este corresponde ao parâmetro que define o *delay* existente antes de inicializar um *WAL flush*. O aumento deste valor pode proporcionar um aumento do débito, permitindo a realização de *commit* a um maior número de transações em cada *flush*. Existindo, assim, um compromisso entre o número de transações *committed* por *flush* e a sua latência.

Valor (s)	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
1	29.65	0.3142159119915683	0.0
30	35.3	0.24556962066430596	0.0

Tabela 7: Valores obtidos com a mudança do parâmetro *commit_delay*.

Tal como referido anteriormente, existiu uma melhoria dos valores de desempenho, assim sendo, escolhemos o valor de 30s para este parâmetro.

4.3.6 Parâmetro *commit_siblings*

Este é responsável pela definição do número mínimo de transações concorrentes ativas necessário para incorrer num *commit_delay*. É de destacar que o valor escolhido para este parâmetro foi de 5.

Tendo em conta os valores obtidos e a função deste parâmetro, decidimos definir o valor deste parâmetro a 20.

Valor	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
1	30.3	0.2962864622128713	0.0
5	36.1	0.2661937058199446	0.0
10	33.1	0.2896063003610272	0.0
20	38.2	0.25206437288481676	0.0

Tabela 8: Valores obtidos com a mudança do parâmetro *commit_siblings*.

4.4 Checkpoints

4.4.1 Parâmetro *checkpoint_timeouts*

O parâmetro *checkpoint_timeouts* define o tempo entre WAL checkpoints automáticos. Este define um limite de tempo após o qual um checkpoint é considerado falha se não for concluído dentro desse período.

Valor (s)	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
30	32.25	0.2643649285891473	0.0
60	42.0	0.25065566172619047	0.0
120	37.3	0.27070158063538874	0.0
300	29.1	0.3427788952302406	0.0

Tabela 9: Valores obtidos com a mudança do parâmetro *checkpoint_timeouts*.

Tendo em conta os valores obtidos e a função deste parâmetro, decidimos definir o valor deste parâmetro a 60s, ou seja, 1min.

4.4.2 Parâmetro *max_wal_size*

O parâmetro *max_wal_size* define o tamanho máximo aproximado que o arquivo de log sequencial pode alcançar durante *checkpoints* automáticos, sendo por *default* 1 GB. Contudo, o grupo também testou para valores superiores, para, desse modo, conseguir observar o impacto de uma WAL menos limitada na performance.

Valor (GB)	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
1	27.25	0.3147695015651376	0.0
2	35.7	0.25285307569187676	0.0
4	39.2	0.2388763372002551	0.0
8	35.7	0.25791985953641455	0.0

Tabela 10: Valores obtidos com a mudança do parâmetro *max_wal_size*.

Tendo em conta os valores obtidos e a função deste parâmetro, decidimos definir o valor deste parâmetro a 4GB.

4.4.3 Parâmetro *min_wal_size*

O parâmetro *min_wal_size* estabelece um limite de tamanho no qual, sempre que o espaço disponível no arquivo WAL (Write-Ahead Log) estiver abaixo desse limite, os arquivos WAL antigos são reciclados para uso futuro em vez de serem removidos. Neste também espera-se observar um melhor desempenho com um limite inferior maior.

Valor (MB)	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
160	32.7	0.24416541379051987	0.0
320	28.3	0.31672869510600704	0.0
640	27.85	0.2946918255745063	0.0

Tabela 11: Valores obtidos com a mudança do parâmetro *min_wal_size*.

Tendo em conta os valores obtidos e a função deste parâmetro, decidimos definir o valor deste parâmetro a 160MB.

4.4.4 Parâmetro *checkpoint_completion_target*

O parâmetro *checkpoint_completion_target* define o objetivo de conclusão do *checkpoint* como uma fração de tempo total entre *checkpoints*.

Valor	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
0.0	43.05	0.23359238987921022	0.0
0.2	34.65	0.2876284800591631	0.0
0.4	28.55	0.32052678762697023	0.0

Tabela 12: Valores obtidos com a mudança do parâmetro *checkpoint_completion_target*.

4.4.5 Parâmetro *checkpoint_warning*

O parâmetro *checkpoint_warning* define o tempo de espera após o qual uma mensagem de aviso é exibida quando um *checkpoint* está demorando mais do que o valor definido. Por outras palavras, se é excedido o valor de tempo estabelecido em relação ao intervalo de tempo entre *checkpoints* causados pelo preenchimento de segmentos de arquivos *Write-Ahead Logs*, resultará na escrita de uma mensagem no log do servidor.

Valor (s)	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
15	38.55	0.21920162404020752	0.0
30	46.95	0.20676059025239615	0.0

Tabela 13: Valores obtidos com a mudança do parâmetro *checkpoint_warning*.

Tendo em conta os valores obtidos e a função deste parâmetro, decidimos definir o valor deste parâmetro a 30s.

4.5 Archiving

4.5.1 Parâmetro *archive_mode*

O parâmetro *archive_mode* determina o modo como os dados são armazenados e acessados no sistema de armazenamento de arquivos. Por *default*, a opção de armazenamento de segmentos WAL concluídos está desativada. Quando ativada, os segmentos WAL concluídos são enviados para o armazenamento de arquivos.

Valor	<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
<i>on</i>	32.6	0.259797303648773	0.0
<i>off</i>	30.35	0.298724979339374	0.0

Tabela 14: Valores obtidos com a mudança do parâmetro *archive_mode*.

Tendo em conta os valores obtidos e a função deste parâmetro, decidimos ativar este parâmetro.

4.6 Configuração final

```
{  
  "synchronous_commit": "on",  
  "wal_sync_method": "fsync",  
  "full_page_writes": "off",  
  "commit_delay": "",  
  "commit_siblings": "5",  
  "checkpoint_timeout": "1min",  
  "max_wal_size": "4GB",  
  "min_wal_size": "160MB",  
  "checkpoint_completion_target": "0.0",  
  "checkpoint_warning": "30s",  
}
```

```
"archive_mode": "on",
```

<i>Throughput</i> (tx/s)	<i>Response Time</i> (s)	<i>Abort rate</i> (%)
33.45	0.27268158500597905	0.0

Tabela 15: Valores obtidos combinando os valores ótimos obtidos acima

Como se pode observar, apesar da utilização de todos os valores ótimos combinados, obtivemos um valor pior do que o valor base que tínhamos inicialmente. Assim sendo, o grupo decidiu não alterar os parâmetros, mantendo a configuração *default* original.

5 Conclusões e Trabalho Futuro

A realização deste trabalho prático possibilitou não só uma melhor consolidação dos conceitos lecionados nas aulas teóricas e práticas, como também possibilitou a aplicação dos mesmos num projeto de maior escala, comparativamente aos realizadas nas aulas.

Em relação à otimização das interrogações analíticas, este projeto possibilitou aprofundar o conhecimento sobre as ferramentas de otimização de *queries*, como os índices e as vistas materializadas ou pela simples criação de novas tabelas. Além disso, permitiu ainda compreender quais as melhores soluções a serem utilizadas para determinadas circunstâncias. Enquanto grupo, consideramos que os resultados obtidos, nesta primeira fase, foram bastante satisfatórios, visto que conseguimos reduzir significativamente o tempo de execução de cada uma.

Relativamente à segunda fase do projeto, ou seja, à fase de otimização no ***Spark***, consideramos que poderiam ter sido alcançados melhores resultados, caso as *queries* tivessem sido implementadas através de métodos *Spark*. Permitindo individualizar os passos de cada das *queries* e, consequentemente, permitindo a realização de otimizações específicas como, por exemplo, a alteração da ordem de execução das mesmas. Apesar disto, os resultados que obtivemos evidenciam a obtenção e o alcance de melhores tempos de execução.

Quanto à terceira fase do projeto, ou seja, à fase de otimização do desempenho da carga transacional, o grupo considerou que poderia ter alcançado resultados bastante melhores. Os testes efetuados poderiam ter sido feitos durante um intervalo de tempo maior, com um tempo de aquecimento maior e poderia ter sido restaurado o estado da base de dados inicial. Todas as variáveis foram tidas em conta, porém por falta de tempo de execução, também foram descartadas ou minimizadas.

Em suma, no que diz respeito ao trabalho desenvolvido como um todo, consideramos que, apesar dos diversos desafios com que nos deparamos durante a sua resolução, o resultado obtido é satisfatório. Existindo, contudo e de forma clara, alguns aspetos que podem ser melhorados.