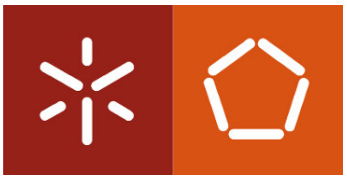

Universidade do Minho

Mestrado em Engenharia Informática

Administração de Bases de Dados

Pedro Pereira (PG47581)
Rui Moreira (PG50736)
Bernardo Saraiva (PG50259)
Miguel Santa Cruz (PG50661)
André Ferreira (PG50222)

31 de maio de 2023

Conteúdo

Capítulo 1

Introdução

O presente trabalho prático foi desenvolvido no âmbito da unidade curricular Administração de Bases de Dados, pertencente do perfil de Engenharia de Aplicações, lecionada no Mestrado de Engenharia Informática da Universidade do Minho.

O projeto consiste na configuração, otimização, e avaliação do *benchmark* que usa dados *IMDb* e contém operações transacionais e analíticas. A componente transacional simula uma pequena parte de um serviço de *streaming* de vídeo, baseado no *dataset IMDb* juntamente com informação extra relativa aos utilizadores. As interrogações analíticas simulam operações estatísticas e de *data warehousing* sobre o conjunto de dados *IMDb* + informação de utilizadores. Após esta fase inicial, começou-se a tentar otimizar o desempenho das interrogações analíticas no *PostgreSQL*, tendo em conta, principalmente, os mecanismos de redundância (e.g., índices e vistas materializadas). De seguida, exportaram-se os dados de *PostgreSQL* para ficheiros e otimizou-se o desempenho das interrogações analíticas no *Spark* e por fim, tentou-se otimizar o desempenho da carga transacional.

Neste relatório consta em detalhe os passos e as decisões tomadas pelo grupo para atingir estes objetivos.

Capítulo 2

Automatização

2.1 Deployment

Numa fase inicial, o grupo começou por automatizar a criação e configuração da máquina virtual de teste utilizando para esse efeito a ferramenta *Ansible*. Foi efetuada também, manualmente, a criação de um *bucket* para armazenar a base de dados para posterior uso.

Terminado este processo, passou as ser possível executar um *playbook ('start-playbook.yaml') Ansible* que efetua, resumidamente, as seguintes tarefas:

- lança todos os recursos do *GCP* necessários (máquina virtual, disco, *firewall*)
- instala o *Docker*
- instala e lança o *PostgreSQL*
- copia os ficheiros da pasta *imdbBench* para a máquina
- faz download dos ficheiros de dados necessários armazenados no *bucket* criado inicialmente
- copia ficheiros relativos ao *Spark* e aos *scripts* de testes para a máquina

Foi ainda criado um *playbook ('shutdown-playbook.yaml') Ansible* que termina e remove os recursos criados no *GCP*, permitindo assim facilmente poupar recursos.

2.2 Testes

Foi também automatizada a execução de todos os testes transacionais e analíticos através de *scripts bash* (B), com o intuito de obter resultados mais consistentes e em maior quantidade, permitindo ainda que estes sejam facilmente repetidos.

Estes *scripts* aplicam sequencialmente configurações em ficheiros *.test.sql* (podendo conter índices, vistas materializadas, alteração de configurações) e executa repetidamente e de forma configurável a carga a ser avaliada. Após a avaliação de uma configuração, os *scripts* aplicam a configuração correspondente *.cleanup.sql* que repõe a configuração base do *PostgreSQL*.

O *script* de carga analítica com *PostgreSQL* (B.2) permite a especificação de com que *queries* deve cada configuração deve ser testada, permitindo avaliar cada configuração apenas em relação às *queries* relevantes, bem como de com quantas repetições devem ser efetuadas as configurações a serem testadas. De salientar que entre a execução de cada *query* é executado um *script* (B.4) que reinicia o serviço *postgresql*, limpa a cache do sistema e executa o comando *vacuum analyze* de forma a repor as estatísticas da base de dados.

Já o *script* da carga transacional (B.3) permite especificar para que conjunto de número de clientes devem ser testadas as configurações. Entre cada execução da carga transacional, a base de dados é recarregada para o seu estado inicial através de um *script* (B.1), de forma a que os dados adicionados por cada teste não influenciem o resultado de outros.

Capítulo 3

Carga analítica no *PostgreSQL*

No que toca às interrogações analíticas, o grupo recorreu sempre ao comando *EXPLAIN ANALYZE* do *PostgreSQL* para analisar os planos de execução das *queries*, auxiliando-se do *website* `https://explain.dalibo.com/` [2], que permite analisar de forma mais legível, todas estas estatísticas como, por exemplo, o tempo total em cada operação do plano e a percentagem correspondente em relação ao tempo total de execução. Assim torna-se mais fácil identificar as operações mais custosas para tentar otimizá-las através da aplicação de índices e vistas materializadas.

Também se optou por estratégias como alteração do código *SQL*, sendo importante referir que houve sempre o cuidado de verificar a consistência dos resultados, isto é, os resultados das interrogações analíticas não se alteravam.

De notar ainda que o grupo realizou 3 execuções para cada teste e, por isso, os resultados apresentados (principalmente o tempo de execução) correspondem a uma média aritmética das 3 execuções.

3.1 Interrogação Analítica 1

```sql
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 seleciona os 10 filmes melhores classificados por década, do género 'Drama' e que tenham registos de titulo (" *TitleAkas*") nos países especificados ('US', 'GB', 'ES', 'DE', 'FR', 'PT').

Para começar, como foi referido em cima, foi usado o comando *EXPLAIN ANALYZE* para visualizar o plano de execução e verificar o tempo de execução da *query* original, que corresponde a 32,4 segundos, como é possível observar no Apêndice A.1.

Após introduzir este plano de execução no `https://explain.dalibo.com/` [2], e analisar algumas das estatísticas sobre este, concluiu-se que nesta *query* o *SELECT* do *title_id* do **titleAkas**, o *JOIN* do **titleGenre** e o **genre** foram os que mais impacto tiveram no tempo de processamento da *query*.

Numa primeira abordagem optou-se por criar índices para otimizar estas duas operações criando um índice composto para a tabela **titleAkas** usando as colunas *title_id* e *region*, já que são pretendidos todos os títulos cujo *id* esteja na tabela **titleAkas** com as regiões pretendidas. Com este índice, o tempo de execução melhorou para 9,3 segundos, significando uma melhoria de aproximadamente 71,2% relativamente ao tempo original.

No que toca ao *JOIN* entre a tabela **titleGenre** e a tabela **genre** também foi testado modificar a *query SQL* de forma a tornar toda a cláusula *SELECT* da tabela **titleGenre** num *JOIN*, que resultou no seguinte:

```
JOIN titleGenre TG ON tg.title_id = t.id AND tg.genre_id = 8
```

Apesar desta modificação no *SQL* ter sido testada não se comprovou eficaz na *query* produzindo aproximadamente o mesmo tempo de execução da *query* original.

Foram também considerados outros índices que possivelmente poderiam melhorar o desempenho da *query1*, como por exemplo:

- um *index* para a tabela **userHistory** na coluna *title_id*, já que na tabela apesar de existir um *index* composto *(user_id,title_id)*, correspondente à chave primária, através da análise do planeamento da execução foi possível perceber que este não era utilizado na leitura, provavelmente por causa da ordem da mesma. Com a utilização deste *index* o tempo de execução melhorou para 16,7 segundos, correspondendo a uma melhoria de performance de cerca de 50% relativamente ao tempo original.
- um *hash index* na tabela **title** na coluna *type_id* para melhorar a comparação de igualdade entre *title_type* e *'movie'*. Efetivamente era usado este *index*, no entanto a performance piorou, tendo a *query* sido executada em 40,9 segundos, piorando desta forma cerca de 25%.

3.2 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 retorna os 100 '*tvSeries*' mais vistos nos últimos 30 dias, excluindo visualizações de utilizadores dos Estados Unidos (*US*) e Reino Unido (*GB*). Para cada '*tvSeries*' fornece informação do *id*, título principal, géneros, número de temporada e número de visualizações. O tempo de execução inicial é 22,0 segundos, como é possível observar no Apêndice A.2.

A primeira otimização realizada foi obtida pela a análise da *query* onde era possível remover o *JOIN* **title_t2**, uma vez que se pode modificar a *query* para utilizar o **title_t** com o intuito de simplificar a mesma.

```
explain analyze 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 userHistory uh ON uh.title_id = te.title_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;
```

Após introduzir este plano de execução no `https://explain.dalibo.com/` [2] e analisar algumas das estatísticas sobre esta *query*, concluiu-se que o *GROUP BY* e o *JOIN* entre o **titleGenre** e o **genre** eram as cláusulas que mais demoravam a serem processados na *query*.

Então foi criado um *index* na tabela **titleGenre** na coluna *title_id* para que fosse usado no *GROUP BY*. Ao analisar o *explain analyze* verifica-se que não foi usado este *index*, não havendo assim nenhuma melhoria.

Foram também considerados outros índices que possivelmente poderiam melhorar o desempenho desta *query*, como por exemplo:

- um *index* na tabela **userHistory** na coluna *title_id* para otimizar o *JOIN* com a tabela **userHistory**. Com a utilização deste *index* o tempo de execução foi de 20,9 segundos, não havendo nenhuma melhoria.
- um *hash index* na tabela **users** na coluna *country_code* para otimizar a comparação de igualdade entre do *country_code* entre *'US'* ou *'GB'*. Ao analisar o *explain analyze* verificou-se que não foi usado este *index*, não havendo melhoria alguma.
- um *hash index* na tabela **title** na coluna *id* para otimizar o *JOIN* com a tabela **title**. Com a utilização deste *index* o tempo de execução foi para 21,3 segundos, não havendo nenhuma melhoria.
- um *index* na tabela **titleEpisode** na coluna *parent_title_id*. Pela análise do *explain analyze* constatou-se que não foi utilizado este *index*, não havendo nenhuma melhoria.

Inicialmente, o grupo descartou a utilização de vistas materializadas, uma vez que sempre que algo fosse adicionado/modificado a vista teria de ser refeita. Porém, visto que não se conseguiu obter nenhuma otimização para esta *query*, o grupo reconsiderou e procurou otimizar através das vistas materializadas.

```
CREATE MATERIALIZED VIEW all_genres_mv 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;
```

Assim, chegou à conclusão que vista materializada acima levaria a uma otimização do tempo de execução para 7,2 segundos.

Após isto, ao colocar o plano de execução resultante no `https://explain.dalibo.com/`, verificou-se que o *Parallel Seq Scan* era a tarefa mais custosa, já que para esta vista não existia nenhum *index* por predefinição. Assim, foi adicionado um *hash index* para esta vista na coluna *title_id* possibilitando passar de um *Parallel Seq Scan* para um *Index Scan*.

Esta combinação da vista com o índice resulta um tempo de execução de 5,3 segundos, uma redução de 75,9% do tempo de execução em relação à *query* original.

O grupo considera que esta vista materializada não é muito problemática, uma vez que não se espera que sejam adicionados muitos filmes todos os dias, não sendo necessários muitas recomputações desta.

3.3 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 *query* devolve as 100 atrizes mais ativas nos últimos 10 anos, com base no número de papéis em '*movie*', '*tvSeries*', '*tvMiniSeries*' e '*tvMovie*'. Para cada atriz devolve o nome principal, idade e número de participações. O tempo de execução desta *query* original é 170,7 segundos, como é possível observar no Apêndice A.3.

A primeira otimização realizada foi obtida pela a análise da *query* onde se removeu o *LEFT JOIN* da tabela **titleEpisode** e a verificação *te.title_id IS NULL*, já que esta cláusula não era usada.

```
explain analyze 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
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'
    )
GROUP BY n.id
ORDER BY roles DESC
LIMIT 100;
```

Após se ter removido esta cláusula foi testado se o output era o mesmo (sem o *LIMIT*) e constatou-se um output inalterado, relativamente à *query default*. Assim foi possível melhorar o tempo de execução da *query* para 36,2 segundos, ou seja, um diminuição no tempo de execução de 78,8% relativamente ao tempo original.

Após introduzir o plano de execução desta *query* no `https://explain.dalibo.com/` [2], em termos de custo de tempo destacou-se o *Index Only Scan* na tabela **titlePrincipalsCharacters** e o *Index Scan* na tabela **name**.

Então foi criado um *index* composto na tabela *titlePrincipalsCharacters* com *title_id* e *name_id*, já que são efetuadas comparações entre *title_id* e o *name_id* da tabela **titlePrincipalsCharacters** na cláusula do *JOIN* **titlePrincipalsCharacters**, tendo originado um tempo de execução nesta nova *query* de 31,4 segundos, o que se traduz numa melhoria de performance de 13,3% em relação à otimização prévia.

Foram também considerados outros índices que possivelmente poderiam melhorar o desempenho desta *query*, como por exemplo:

- um *index* na tabela **title** na coluna *title_type* para otimizar a comparação do *title_type* com um dos tipos especificados. Com a utilização deste *index* o tempo de execução foi de 41,3 segundos, o que significa que o tempo de execução piorou 31,5% relativamente à *query* otimizada. Isto deve-se ao facto de que, com este *index*, o *Bitmap Index Scan* com um *Parallel Bitmap Heap Scan* aumenta o tempo de execução, já que apesar de ser possível verificar o *type_id* de uma forma mais eficiente, as filtragem relativa à década ocorre em outra operação, ao contrário ao que acontece na execução da *query* sem este *index*.

3.4 Otimizações de parâmetros de configuração no *PostgreSQL*

3.4.1 *Work Memory*

Ao analisar o planeamento das diferentes *queries*, foi possível perceber que o valor *default* de *work_mem* (4 *MB*) [3] não seria suficiente para comportar determinadas operações no *buffer* em memória, podendo isto ser verificado no planeamento da *query3* (Apêndice A.3) através do aparecimento da cláusula "*Disk*: 14328kB". Para resolver este problema decidimos aumentar o tamanho da *work_mem* para que esta operação fosse comportável em memória. Desta forma, o parâmetro **work_mem** foi definido como 40 *MB* e desta forma a *query3* executou em apenas 130,2 segundos resultando numa melhoria no tempo de execução de cerca de 23.5% relativamente ao tempo original.

3.4.2 *Maximum Parallel Workers Per Gather*

Para aumentar a paralelização das operações a nível das *queries* analíticas, foi também analisado valor ótimo de *workers per gather*. O valor *default* de 2 foi substituído por 4, melhorando assim 2 das 3 *queries* analíticas.

- A *query1* foi executada em apenas 17,4 segundos, correspondendo a uma melhoria no tempo de execução de cerca de 46%.
- A *query2* foi executada em 24,5 segundos, não beneficiando desta otimização.
- A *query3* foi executada em 144,1 segundos, resultando numa melhoria no tempo de execução de cerca de 15,3%.

3.4.3 *Shared Buffers*

O parâmetro **shared_buffers** deve corresponder a 25% da *RAM* total da máquina a correr a base de dados [1]. No caso em análise como a instância onde está a correr a base de dados possui 8 *GB* de *RAM*, este parâmetro foi definido como 2 *GB*. Ao executar as *queries* analíticas esta mudança apenas se verificou significativa na *query3*, tendo-se obtido um resultado de 147,1 segundos que resulta numa melhoria de tempo de 13,8%.

3.5 Resultado Final

3.5.1 Interrogação Analítica 1

Na *query 1*, tendo em conta todos os índices e parâmetros do *Postgres* analisados, utilizou-se os seguintes parâmetros:

- *Index* composto na tabela **titleAkas** com as colunas *title_id* e *region*
- *Index* na tabela **userHistory** com a coluna *title_id*

- Alterar a *work_mem* para 40 *MB*
- Alterar o *shared_buffers* para 2 *GB*
- Alterar o *max_parallel_workers_per_gather* para 4

Com estes parâmetros em conjunto, obteve-se um tempo de execução de 6,2 segundos, uma redução de 80,8% em relação à *query* original.

3.5.2 Interrogação Analítica 2

Na *query 2*, tendo em conta todos os índices e parâmetros do *PostgreSQL*, utilizou-se os seguintes parâmetros:

- Alterar a *query* original para remover o segundo *JOIN* com a tabela **title_t2**
- *Materialized View* que contém para cada *title_id* uma lista dos géneros correspondentes.
- *Hash Index* com a coluna *title_id* na *materialized view*
- Alterar a *work_mem* para 40 *MB*
- Alterar o *shared_buffers* para 2 *GB*
- Alterar o *max_parallel_workers_per_gather* para 4

Com estes parâmetros, obteve-se um tempo de execução de 3,5 segundos, que corresponde a uma redução de 84% em relação à *query* original.

3.5.3 Interrogação Analítica 3

Na *query 3*, tendo em conta todos os índices e parâmetros do *PostgreSQL*, utilizou-se os seguintes parâmetros:

- Alterar a *query* original para remover o *LEFT JOIN* da tabela **titleEpisode** e a comparação *te.title_id IS NULL*
- *Index* composto na tabela **titlePrincipalsCharacters** com as colunas *title_id* e *name_id*
- Alterar a *work_mem* para 40 *MB*
- Alterar o *shared_buffers* para 2 *GB*
- Alterar o *max_parallel_workers_per_gather* para 4

Com estes parâmetros, obteve-se um tempo de execução de 17,1 segundos, um redução de 90% em relação à *query* original.

Capítulo 4

Carga analítica no *Spark*

4.1 Exportação dos dados

Para possibilitar a exportação de dados da base de dados *PostgreSQL* para *Spark* foi necessário exportar os dados das diferentes tabelas para *CSV* através da seguinte interrogação SQL:

```
1 COPY table TO 'tmp\table.csv' DELIMITER ',' CSV HEADER;
```

Após esta exportação, procedeu-se à conversão das tabelas para *parquet* através de um simples *script python* recorrendo à biblioteca *PySpark*, que se encontra evidenciado abaixo:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("spark://spark:7077").getOrCreate()
csv_data = spark.read.csv("/app/imdb/my_file.csv", header=True, inferSchema=True)
csv_data.write.parquet("/app/imdb/my_file.parquet")
```

Para que seja possível fazer *deployment* através do *Ansible*, à semelhança do que acontece com o *postgreSQL*, os ficheiros *parquet* foram também colocados num *bucket* da *Google Cloud Platform*.

4.2 Instalação do cluster *Spark*

Para a avaliação das interrogações analíticas com o *Spark*, este foi instalado utilizando *Docker* num *cluster* com 1 *master* e 3 *workers* com 2 *GB* de memória e limitados a 2 *cores* cada um.

4.3 Otimização do desempenho das interrogações analíticas

Numa fase inicial, as interrogações analíticas foram convertidas para a *API Dataframes* do *PySpark*, sendo o resultado de cada uma das interrogações verificado como correspondente ao resultado obtido no *postgreSQL*. Abaixo apresenta-se o código obtido para cada uma das *queries* e respetivas otimizações.

4.3.1 Otimização da interrogação analítica 1

```
def query1_dataframe(title: DataFrame,
                     userHistory: DataFrame,
                     titleGenre: DataFrame,
                     genre: DataFrame,
                     titleAkas: DataFrame
                     ) -> List[Row]:
```

```
    rank_window = Window.partitionBy(col("decade")).orderBy(col("rating").desc(), title.id)


    return title\
        .where(title.title_type == 'movie') \
        .where(title.start_year >= 1980) \
        .join(titleAkas.where(titleAkas.region.isin('US', 'GB', 'ES', 'DE', 'FR', 'PT'))\
        .select(titleAkas.title_id).distinct(), title.id == titleAkas.title_id)\
        .join(titleGenre.where(titleGenre.genre_id == 8).distinct(), titleGenre.title_id == title.id)\
        .join(userHistory, userHistory.title_id == title.id)\
        .groupBy(title.id, title.primary_title, (floor(title.start_year / 10) * 10).alias("decade")) \
        .agg(avg(userHistory.rating).alias("rating"),\
            count(userHistory.rating).alias('count')) \
        .where(col('count') >= 3) \
        .select(title.id, title.primary_title, col("decade"), col('rating'), rank().over(rank_window).alias("rank"))\
        .orderBy(col("decade"), col("rating").desc()) \
        .where(col("rank") <= 10)\
        .collect()
```

Numa primeira fase, tentou-se fazer a repartição dos dados por cada *worker* (com *repartition(3)* depois dos *JOINS*). Esta alteração não trouxe qualquer melhoria na execução da *query*.

Posteriormente, o grupo considerou ainda como otimização, a partição dos dados da tabela **titleAkas** pela coluna *region* juntamente com uma partição pelos *titleType* e *start_year* na tabela **title** que levou a um tempo de execução de 34,9s.

Ainda utilizando estas partições com um método de compressão (*com gzip*) levou a um tempo de 32,9s.

De notar que estes testes foram executados com a configuração *adaptive enabled* e por isso levou a uma melhoria de 16% (39,2s para 32,9s). Estes resultados serão resumidos mais à frente na secção 4.3.4

4.3.2 Otimização da interrogação analítica 2

```
def query2_dataframe(title: DataFrame,
                     titleEpisode: DataFrame,
                     userHistory: DataFrame,
                     users: DataFrame,
                     titleGenre: DataFrame,
                     genre: DataFrame
                     ) -> List[Row]:
    tg = titleGenre.join(genre, genre.id == titleGenre.genre_id) \
        .groupBy(titleGenre.title_id) \
        .agg(collect_list(genre.name).alias('genres')) \
        .select(titleGenre.title_id, col("genres"))

    return title.alias("t")\
        .join(titleEpisode, titleEpisode.parent_title_id == col("t.id")) \
        .join(title.alias("t2"), titleEpisode.title_id == col("t2.id")) \
        .join(userHistory, userHistory.title_id == col("t2.id")) \
        .join(users, users.id == userHistory.user_id) \
        .join(tg, tg.title_id == col("t.id")) \
        .where(col("t.title_type") == "tvSeries") \
        .where(titleEpisode.season_number.isNotNull())\
        .where(~users.country_code.isin(["US", "GB"]))\
        .where(userHistory.last_seen.between(current_timestamp() - expr("INTERVAL 30 DAYS"),current_timestamp()))\
        .groupBy(col("t.id"), col("t.primary_title"), tg.genres, titleEpisode.season_number)\
        .agg(count("*").alias("views"))\
        .orderBy(col("views").desc(), col("t.id"))\
        .limit(100)\
```

```
        .select(col("t.id"), col("t.primary_title"), tg.genres, titleEpisode.season_number, col("views"))\
        .collect()
```

Para esta interrogação analítica, o grupo experimentou realizar partições pelo *title_type* da tabela **title** e por *country_code* na tabela **users**. Contra as expectativas do grupo, estas não levaram a uma melhoria de desempenho, sendo que o tempo de execução resultante foi 60,4s.

4.3.3 Otimização da interrogação analítica 3

```
def query3_dataframe(name: DataFrame,
                     titlePrincipals: DataFrame,
                     titlePrincipalsCharacters: DataFrame,
                     category: DataFrame,
                     title: DataFrame,
                     titleEpisode: DataFrame
                     ) -> List[Row]:
    return name.join(titlePrincipals, name.id == titlePrincipals.name_id) \
        .join(titlePrincipalsCharacters, (titlePrincipalsCharacters.title_id ==
        titlePrincipals.title_id) & (titlePrincipalsCharacters.name_id == titlePrincipals.name_id)) \
        .join(category, category.id == titlePrincipals.category_id) \
        .join(title, title.id == titlePrincipals.title_id) \
        .join(titleEpisode, titleEpisode.title_id == titlePrincipals.title_id, how='left') \
        .where(title.start_year >= year(current_date()) - 10) \
        .where(category.name == 'actress') \
        .where(name.death_year.isNull()) \
        .where(title.title_type.isin(['movie', 'tvSeries', 'tvMiniSeries', 'tvMovie'])) \
        .where(titleEpisode.title_id.isNull()) \
        .groupBy(name.id, name.primary_name, name.birth_year) \
        .agg(count('*').alias('roles')) \
        .orderBy(col('roles').desc()) \
        .select(name.id, name.primary_name, (year(current_date()) - name.birth_year).alias("age"), col("roles"))\
        .limit(100) \
        .collect()
```

O grupo começou por realizar a mesma otimização realizada na *query 3* no *PostgreSQL*, ao remover o *LEFT JOIN*. Porém, tendo realizado esta modificação não se detetou nenhuma melhoria no tempo de execução.

O grupo considerou ainda, que uma possível melhoria seria criar partições pelo *death_year* na tabela **name** e pelo *name* da tabela **category**. Porém, não levou a nenhuma melhoria no tempo de execução.

4.3.4 Otimização através de configurações e testes finais

Para realizar o controlo do tempo necessário à análise dos benefícios das otimizações a implementar foi realizada a medição do tempo através da função *timeit*, inicialmente foram coletados também os dados relativamente ao tempo original para a execução das *queries* propostas.

Após o teste de várias configurações, como por exemplo *spark.sql.adaptive.coalescePartitions.enabled* e *spark.sql.optimizer.dynamicPartitionPruning.enabled*, a única que conduziu a uma otimização foi a configuração *spark.sql.adaptive.enabled* a *True*. Isto permite ao *Spark* escolher o melhor plano de execução das *queries* com estatísticas obtidas em *runtime*, melhorando assim o desempenho de cada *query* [4].

Abaixo apresenta-se uma tabela de forma a resumir os tempos de execução obtidos com as otimizações encontradas, que foram descritas anteriormente.

	Query1	Query2	Query3
Original	59,1 s	70,7 s	183,8 s
Adaptive Enabled	39,2 s	60,49 s	132,8 s
+ Partições	32,9 s	60,49 s	132,8 s

Capítulo 5

Carga transacional

A última fase deste trabalho prático consiste em otimizar o desempenho da carga transacional através de mecanismos de redundância (como por exemplo índices e vistas materializadas), paralelismo, parâmetros de configuração e/ou código *SQL/Java*. Conforme recomendado e respeitado nos outros objetivos, o grupo começou por se focar na otimização através da redundância, paralelismo, e/ou código.

5.1 Otimização através da aplicação de índices

Assim, o grupo fez uma análise a cada um dos *statements* utilizados pelas transações fornecidas pelo *benchmark*. Antes de passar à analise, é importante referir que o grupo alterou o *statement getTitleFromPopular*, que continha o intervalo de 7 dias para 31 dias. Isto porque, com 7 dias, não se obtinha nenhum titulo como resultado, uma vez que a base de dados foi carregada pela primeira vez há mais de 7 dias e por isso não tem dados dos últimos 7 dias, o que resultava na impossibilidade de executar o *benchmark*.

Após esta análise, o grupo considerou os seguintes índices:

- um *index* na tabela **title** pela coluna *title_type* - *statement getTitleFromType*
- um *index* na tabela **title** pela coluna *start_year* - *statements getTitleFromType e searchTitleByName*
- um *index* na tabela **userHistory** pela coluna *title_id* - *statement getTitleRating*
- um *index* na tabela **userHistory** pela coluna *last_seen* - *statement getTitleFromPopular*
- um *index* na tabela **title** pela expressão *to_tsvector('english', primary_title)*, utilizando GIN (Generalized Inverted Index) - *statement searchTitleByName*
- um *index* na tabela **titleAkas** composto pelas colunas *(title_id, region)* - *statement getTitleNameInRegion*

Na escolha dos índices acima mencionados, foram analisados todos os *statements* que compõe as 5 transações presentes, utilizando o comando *explain analyze* e sendo testados individualmente de forma a analisar o seu impacto na execução individual dos *statements*.

5.2 Otimização através dos parâmetros de configuração

Para otimizar utilizando os parâmetros de configuração do *PostgreSQL* foram efetuadas as seguintes alterações aos valores padrões:

- *shared_buffers* a 3.2 *GB*
- *max_wal_size* a 4 *GB*

Estes parâmetros de configuração foram utilizados uma vez que um aumento *max_wal_size* permite uma redução dos custo dos checkpoints realizados aquando do preenchimento do *WAL*, e o aumento do *shared_buffers* permite manter mais dados em memória, reduzindo a necessidade de leitura do disco.

O parâmetro *max_parallel_workers_per_gather* não foi utilizado, uma vez que cargas transacionais lidam com um grande número de clientes concorrentes, e por isso não seria benéfico utilizar mais *workers* por cada *query*. Do mesmo modo não foi alterado o parâmetro *work_mem*, uma vez que um grande número de pedidos concorrentes com valores altos nesse parâmetros poderia levar a utilizações excessivas de memória.

5.3 Resultados

Cada execução do *benchmark* foi efetuada com um *warmup* de 30 segundos e tempo de execução de 4 minutos (240 segundos). Uma vez que esta execução é contínua e o resultado obtido é resultante da execução de um grande número de *queries*, consideramos que não seria necessário a utilização de 3 repetições de forma a obter resultados mais fiáveis, sendo então apenas executado uma única vez para cada uma das combinações.

Alguns dos teste realizados apresentaram *abort rates* acima de 0% mas abaixo de 0.01% e por serem valores tão baixos foram considerados ainda assim válidos.

	1 cliente	4 clientes	8 clientes	16 clientes	24 clientes	32 clientes	48 clientes	64 clientes
Original	9.50	34.80	39.91	42.44	42.99	39.74	41.43	41.33
Com índices	20.31	96,54	160,20	250.50	249.33	253.75	257.07	232,60
Com índices + parâmetros	25.1	121.21	220.85	256.61	268.50	263.90	284.43	281.98

Tabela 5.1: Débito observado (transações por segundo)

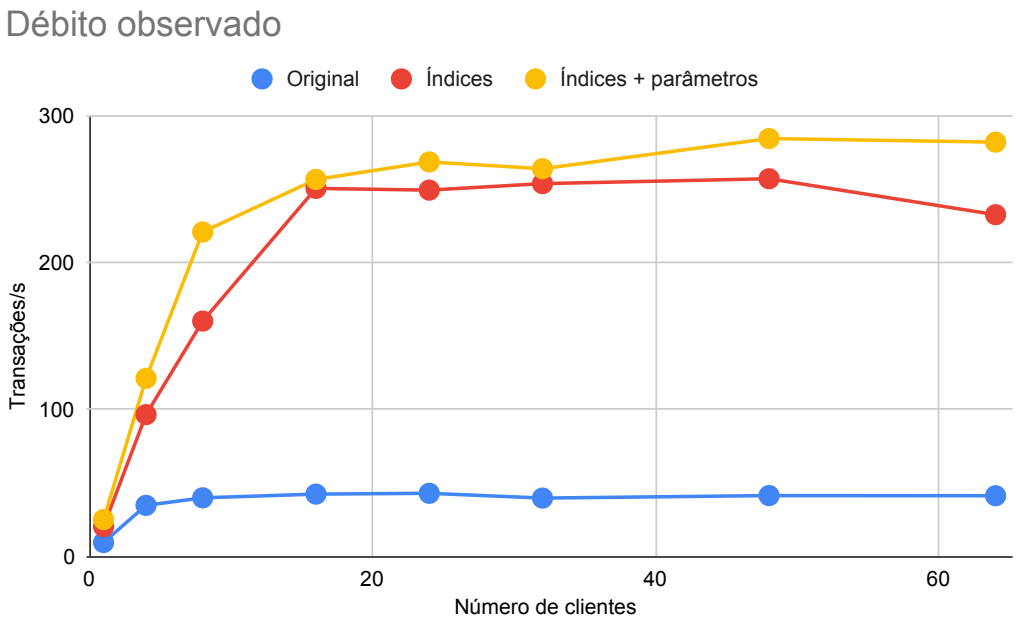

Figura 5.1: Débito observado

Podemos observar pelos resultados que o débito da configuração original estagnou por volta das 40 transações por segundo, tendo sido observado o seu maior valor de 42,99 com 24 clientes. Ambas as configurações finais otimizadas testadas conseguiram um aumento significativo do débito, atingindo melhorias de mais de 500%.

	1 cliente	4 clientes	8 clientes	16 clientes	24 clientes	32 clientes	48 clientes	64 clientes
Original	106	114	205	377	557	793	1091	1416
Com índices	49	42	50	64	96	126	185	262
Com índices + parâmetros	40	33	36	61	88	121	167	227

Tabela 5.2: Latência média observada (milissegundos)

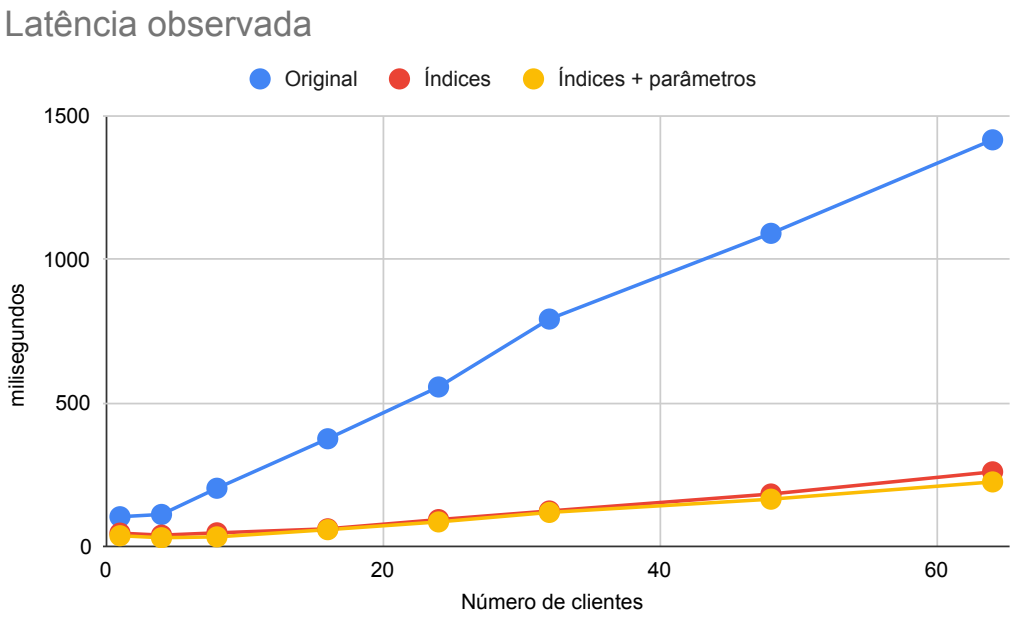

Figura 5.2: Latência média observada

Apesar do débito observado na configuração original se manter relativamente estável entre os 8 e os 64 clientes, foi observado um aumento substancial na latência média, atingindo os 1.42 segundos com 64 clientes, enquanto que com 8 clientes este valor foi de 0.20 segundos. Por contraste, com a configuração ótima testada (com índices e parâmetros), foi observada uma latência média de 0.23 segundos com 64 clientes, correspondendo a uma redução de cerca de 84%.

Em suma, podemos concluir que foi atingido um *speed up* do débito em relação à configuração original do teste transacional entre 2.6x (com 1 cliente) e 6.9x (com 48 clientes).

Capítulo 6

Conclusão

Com a execução deste trabalho prático foi possível aplicar os conhecimentos adquiridos na unidade curricular na otimização de *queries* e de operações transacionais sobre a base de dados.

Através da experimentação com diversos métodos de otimização, como por exemplo, índices e entre outros e através da modificação dos parâmetros padrão de configuração do *PostgreSQL* foi possível começar a observar variações significativas nos tempos de execução e no débitos máximos obtidos. Olhando para os ganhos de performance que foram obtidos nas otimização das diversas *queries* podemos concluir que houve uma melhoria significativa no tempo de execução das mesmas. Relativamente à otimização da carga transacional foi possível observar que o débito obtido aumentou também significativamente quando comparado com os valores de débito originais.

O grupo considera que de uma forma geral o trabalho desenvolvido cumpre os requisitos que foram propostos e que os ganhos de performance foram significativos. No entanto, consideramos que o trabalho desenvolvido em relação à otimização da carga analítica com *Spark* não foi satisfatório, e merecia mais atenção de forma a que fosse melhorado.

Como trabalho futuro o grupo considera importante referir que os valores de desempenho obtidos na parte de otimização da carga analítica com *Spark* poderiam certamente ser melhorados através de outras abordagens não consideradas ou de uma melhor aplicação das abordagens consideradas.

Apêndice A

Queries

A.1 Query 1

A.1.1 Plano de execução default

```
Subquery Scan on t_  (cost=421916.13..422108.57 rows=1509 width=86) (actual time=31907.248..31957.787 rows=50 loops=1)
  Filter: (t_.rank <= 10)
  Rows Removed by Filter: 1953
  ->  WindowAgg  (cost=421916.13..422051.97 rows=4528 width=86) (actual time=31907.243..31957.679 rows=2003 loops=1)
        ->  Sort  (cost=421916.13..421927.45 rows=4528 width=66) (actual time=31907.197..31956.106 rows=2003 loops=1)
              Sort Key: (((t.start_year / 10) * 10)), (avg(uh.rating)) DESC, t.id
              Sort Method: quicksort  Memory: 228kB
              ->  Finalize GroupAggregate  (cost=419910.05..421641.17 rows=4528 width=66) (actual time=31848.461..31952.276 rows=2003 loops=1)
                    Group Key: t.id
                    Filter: (count(uh.rating) >= 3)
                    Rows Removed by Filter: 40986
                    ->  Gather Merge  (cost=419910.05..421329.86 rows=11320 width=74) (actual time=31848.401..31921.328 rows=45163 loops=1)
                          Workers Planned: 2
                          Workers Launched: 2
                          ->  Partial GroupAggregate  (cost=418910.03..419023.23 rows=5660 width=74) (actual time=31825.877..31837.942 rows=15054 loops=3)
                                Group Key: t.id
                                ->  Sort  (cost=418910.03..418924.18 rows=5660 width=38) (actual time=31825.822..31828.369 rows=20226 loops=3)
                                      Sort Key: t.id
                                      Sort Method: quicksort  Memory: 2377kB
                                      Worker 0:  Sort Method: quicksort  Memory: 2449kB
                                      Worker 1:  Sort Method: quicksort  Memory: 2405kB
                                      ->  Nested Loop  (cost=202151.39..418557.22 rows=5660 width=38) (actual time=4279.790..31788.831 rows=20226 loops=3)
                                            ->  Nested Loop Semi Join  (cost=202150.83..407045.76 rows=8654 width=58) (actual time=4272.502..26717.752 rows=58327 loops=3)
                                                  ->  Parallel Hash Join  (cost=202150.27..313892.93 rows=53656 width=48) (actual time=4270.262..4965.938 rows=96250 loops=3)
                                                        Hash Cond: ((uh.title_id)::text = (t.id)::text)
                                                        ->  Parallel Seq Scan on userhistory uh  (cost=0.00..80070.05 rows=2500805 width=14) (actual time=0.070..1498.848 rows=2000631 loops=3)
                                                        ->  Parallel Hash  (cost=200367.67..200367.67 rows=87728 width=34) (actual time=2311.827..2311.829 rows=129726 loops=3)
                                                              Buckets: 65536  Batches: 8  Memory Usage: 3840kB
                                                              ->  Parallel Seq Scan on title t  (cost=0.00..200367.67 rows=87728 width=34) (actual time=49.343..2259.907 rows=129726 loops=3)
                                                                    Filter: (((title_type)::text = 'movie'::text) AND (((start_year / 10) * 10) >= 1980))
                                                                    Rows Removed by Filter: 3143815
                                                  ->  Index Scan using titleakas_pkey on titleakas  (cost=0.56..34.98 rows=34 width=10) (actual time=0.225..0.225 rows=1 loops=288751)
                                                        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
                                            ->  Index Only Scan using titlegenre_pkey on titlegenre tg  (cost=0.56..1.33 rows=1 width=10) (actual time=0.086..0.086 rows=0 loops=174982)
                                                  Index Cond: ((title_id = (uh.title_id)::text) AND (genre_id = 8))
                                                  Heap Fetches: 0
Planning Time: 15.315 ms
JIT:
  Functions: 90
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 6.631 ms, Inlining 0.000 ms, Optimization 5.110 ms, Emission 135.955 ms, Total 147.697 ms
Execution Time: 32311.108 ms
```

Node Type	Count	Time
Index Scan	2	32s 983ms 69%
Index Only Scan	1	7s 641ms 16%
Seq Scan	2	5s 383ms 11%
Hash Join	1	1s 388ms 3%
Nested Loop Semi Join	2	119ms 0%
Gather Merge	1	97.3ms 0%
Nested Loop	1	72.1ms 0%
Hash	1	65.3ms 0%
Sort	2	62.7ms 0%
Finalize GroupAggregate	1	30.9ms 0%
Partial GroupAggregate	1	12.2ms 0%
WindowAgg	1	1.57ms 0%
Subquery Scan	1	0.114ms 0%

Index	Count	Time
titleakas_pkey	1	32s 821ms 69%
titlegenre_pkey	1	7s 641ms 16%
genre_pkey	1	161ms 0%

Figura A.1: Estatísticas sobre o plano de execução da query 1 default

A.2 Query 2

```
Limit  (cost=1133655.37..1133655.62 rows=100 width=74) (actual time=23282.204..23282.366 rows=100 loops=1)
  ->  Sort  (cost=1133655.37..1133671.96 rows=6634 width=74) (actual time=22680.179..22680.333 rows=100 loops=1)
        Sort Key: (count(*)) DESC, t.id
        Sort Method: top-N heapsort  Memory: 41kB
        ->  Finalize GroupAggregate  (cost=118873.10..1133401.82 rows=6634 width=74) (actual time=7701.657..22678.214 rows=5809 loops=1)
              Group Key: t.id, (array_agg(g.name)), te.season_number
              ->  Gather Merge  (cost=118873.10..1133280.20 rows=5528 width=74) (actual time=7701.601..22674.862 rows=6455 loops=1)
                    Workers Planned: 2
                    Workers Launched: 2
                    ->  Partial GroupAggregate  (cost=117873.08..1131642.11 rows=2764 width=74) (actual time=7678.161..22092.808 rows=2152 loops=3)
                          Group Key: t.id, (array_agg(g.name)), te.season_number
                          ->  Incremental Sort  (cost=117873.08..1131586.83 rows=2764 width=66) (actual time=7678.120..22091.240 rows=2376 loops=3)
                                Sort Key: t.id, (array_agg(g.name)), te.season_number
                                Presorted Key: t.id
                                Full-sort Groups: 75  Sort Method: quicksort  Average Memory: 29kB  Peak Memory: 29kB
                                Worker 0:  Full-sort Groups: 72  Sort Method: quicksort  Average Memory: 29kB  Peak Memory: 29kB
                                Worker 1:  Full-sort Groups: 75  Sort Method: quicksort  Average Memory: 29kB  Peak Memory: 29kB
                                ->  Merge Join  (cost=117506.23..1131462.45 rows=2764 width=66) (actual time=7653.723..22086.876 rows=2376 loops=3)
                                      Merge Cond: ((te.parent_title_id)::text = (tg.title_id)::text)
                                      ->  Sort  (cost=117505.52..117505.57 rows=19 width=44) (actual time=7582.675..7584.777 rows=2461 loops=3)
                                            Sort Key: t.id
                                            Sort Method: quicksort  Memory: 322kB
                                            Worker 0:  Sort Method: quicksort  Memory: 314kB
                                            Worker 1:  Sort Method: quicksort  Memory: 323kB
                                            ->  Nested Loop  (cost=1.59..117505.12 rows=19 width=44) (actual time=304.143..7575.705 rows=2461 loops=3)
                                                  ->  Nested Loop  (cost=1.30..117448.28 rows=19 width=48) (actual time=304.083..7537.205 rows=2484 loops=3)
                                                        ->  Nested Loop  (cost=0.87..117044.66 rows=720 width=18) (actual time=302.592..6026.096 rows=2593 loops=3)
                                                              ->  Nested Loop  (cost=0.43..116467.87 rows=1193 width=24) (actual time=300.614..3828.624 rows=3611 loops=3)
                                                                    ->  Parallel Seq Scan on userhistory uh  (cost=0.00..111329.32 rows=1193 width=14) (actual time=299.370..2286.584 rows=3611 loops=3)
                                                                          Filter: ((last_seen <= now()) AND (last_seen >= (now() - '30 days'::interval)))
                                                                          Rows Removed by Filter: 1997019
                                                                    ->  Index Only Scan using title_pkey on title t2  (cost=0.43..4.31 rows=1 width=10) (actual time=0.424..0.424 rows=1 loops=10834)
                                                                          Index Cond: (id = (uh.title_id)::text)
                                                                          Heap Fetches: 0
                                                              ->  Index Scan using titleepisode_pkey on titleepisode te  (cost=0.43..0.48 rows=1 width=24) (actual time=0.605..0.605 rows=1 loops=10834)
                                                                    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=0.579..0.579 rows=1 loops=7778)
                                                              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..2.99 rows=1 width=4) (actual time=0.012..0.012 rows=1 loops=7451)
                                                        Index Cond: (id = uh.user_id)
                                                        Filter: ((country_code)::text <> ALL ('{US,GB}'::text[]))
                                                        Rows Removed by Filter: 0
                                      ->  GroupAggregate  (cost=0.71..962198.93 rows=4138392 width=42) (actual time=1.665..13509.389 rows=9374750 loops=3)
                                            Group Key: tg.title_id
                                            ->  Nested Loop  (cost=0.71..834474.73 rows=15198860 width=17) (actual time=1.633..7173.108 rows=15193346 loops=3)
                                                  ->  Index Only Scan using titlegenre_pkey on titlegenre tg  (cost=0.56..462079.46 rows=15198860 width=14) (actual time=1.560..2422.672 rows=15193346 loops=3)
                                                        Heap Fetches: 0
                                                  ->  Memoize  (cost=0.15..0.17 rows=1 width=11) (actual time=0.000..0.000 rows=1 loops=45580039)
                                                        Cache Key: tg.genre_id
                                                        Cache Mode: logical
                                                        Hits: 15197777  Misses: 28  Evictions: 0  Overflows: 0  Memory Usage: 4kB
                                                        Worker 0:  Hits: 15184479  Misses: 28  Evictions: 0  Overflows: 0  Memory Usage: 4kB
                                                        Worker 1:  Hits: 15197699  Misses: 28  Evictions: 0  Overflows: 0  Memory Usage: 4kB
                                                        ->  Index Scan using genre_pkey on genre g  (cost=0.14..0.16 rows=1 width=11) (actual time=0.003..0.003 rows=1 loops=84)
                                                              Index Cond: (id = tg.genre_id)
Planning Time: 19.381 ms
JIT:
  Functions: 148
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 8.658 ms, Inlining 288.276 ms, Optimization 676.786 ms, Emission 533.860 ms, Total 1507.579 ms
Execution Time: 23714.505 ms
```

Node Type	Count	Time ↓	
GroupAggregate	1	6s 336ms	27%
Nested Loop	5	4s 792ms	21%
Index Only Scan	2	3s 954ms	17%
Index Scan	4	3s 716ms	16%
Seq Scan	1	2s 287ms	10%
Merge Join	1	993ms	4%
Limit	1	602ms	3%
Gather Merge	1	582ms	2%
Sort	2	11.2ms	0%
Incremental Sort	1	4.36ms	0%
Finalize GroupAggregate	1	3.35ms	0%
Partial GroupAggregate	1	1.57ms	0%
Memoize	1	0ms	0%

Index	Count	Time ↓	
title_pkey	2	3s 32.4ms	13%
titlegenre_pkey	1	2s 423ms	10%
titleepisode_pkey	1	2s 185ms	9%
users_pkey	1	29.8ms	0%
genre_pkey	1	0.084ms	0%

Figura A.2: Estatísticas sobre o plano de execução da query 1 default

A.3 Query 3

```
Limit  (cost=971357.26..971357.27 rows=1 width=36) (actual time=170621.866..170767.202 rows=100 loops=1)
  ->  Sort  (cost=971357.26..971357.27 rows=1 width=36) (actual time=170108.402..170253.730 rows=100 loops=1)
        Sort Key: (count(*)) DESC
        Sort Method: top-N heapsort  Memory: 38kB
        ->  GroupAggregate  (cost=971357.22..971357.25 rows=1 width=36) (actual time=169814.467..170215.613 rows=183837 loops=1)
              Group Key: n.id
              ->  Sort  (cost=971357.22..971357.23 rows=1 width=28) (actual time=169814.405..170092.546 rows=396348 loops=1)
                    Sort Key: n.id
                    Sort Method: external merge  Disk: 14328kB
                    ->  Nested Loop  (cost=139523.35..971357.21 rows=1 width=28) (actual time=12317.528..169349.155 rows=396348 loops=1)
                          ->  Nested Loop  (cost=139522.79..971354.46 rows=1 width=58) (actual time=12316.049..124452.712 rows=446348 loops=1)
                                ->  Nested Loop  (cost=139522.36..971350.23 rows=1 width=48) (actual time=11108.373..76646.533 rows=2342630 loops=1)
                                      ->  Gather  (cost=139521.80..971345.86 rows=1 width=20) (actual time=11087.980..11761.373 rows=2627555 loops=1)
                                            Workers Planned: 2
                                            Workers Launched: 2
                                            ->  Parallel Hash Anti Join  (cost=138521.80..970345.76 rows=1 width=20) (actual time=11063.462..12087.508 rows=875852 loops=3)
                                                  Hash Cond: ((tp.title_id)::text = (te.title_id)::text)
                                                  ->  Hash Join  (cost=1.16..781764.71 rows=1941667 width=20) (actual time=1.025..7777.790 rows=3185892 loops=3)
                                                        Hash Cond: (tp.category_id = c.id)
                                                        ->  Parallel Seq Scan on titleprincipals tp  (cost=0.00..699000.00 rows=23300000 width=24) (actual time=0.884..5767.459 rows=18639969 loops=3)
                                                        ->  Hash  (cost=1.15..1.15 rows=1 width=4) (actual time=0.057..0.060 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=0.051..0.052 rows=1 loops=3)
                                                                    Filter: ((name)::text = 'actress'::text)
                                                                    Rows Removed by Filter: 11
                                                  ->  Parallel Hash  (cost=84512.06..84512.06 rows=3107006 width=10) (actual time=2176.035..2176.036 rows=2485475 loops=3)
                                                        Buckets: 131072  Batches: 128  Memory Usage: 3808kB
                                                        ->  Parallel Seq Scan on titleepisode te  (cost=0.00..84512.06 rows=3107006 width=10) (actual time=131.695..1529.085 rows=2485475 loops=3)
                                      ->  Index Scan using name_pkey on name n  (cost=0.56..4.37 rows=1 width=28) (actual time=0.024..0.024 rows=1 loops=2627555)
                                            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..4.22 rows=1 width=10) (actual time=0.020..0.020 rows=0 loops=2342630)
                                      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=0.099..0.100 rows=1 loops=446348)
                                Index Cond: ((title_id = (tp.title_id)::text) AND (name_id = (tp.name_id)::text))
                                Heap Fetches: 0
Planning Time: 27.089 ms
JIT:
  Functions: 80
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 5.219 ms, Inlining 261.205 ms, Optimization 367.289 ms, Emission 280.329 ms, Total 914.043 ms
Execution Time: 171169.547 ms
```

Node Type	Count	Time ↓	
Index Scan	2	1m 49s	64%
Index Only Scan	1	44s 635ms	26%
Seq Scan	3	7s 297ms	4%
Hash Join	2	4s 144ms	2%
Nested Loop	3	2s 713ms	2%
Sort	2	782ms	0%
Hash	2	647ms	0%
Limit	1	513ms	0%
GroupAggregate	1	123ms	0%
Gather	1	0ms	0%

Index	Count	Time ↓	
name_pkey	1	1m 3s	37%
title_pkey	1	46s 853ms	27%
titleprincipalscharacters_pkey	1	44s 635ms	26%

Figura A.3: Estatísticas sobre o plano de execução da query 3 default

Apêndice B

Scripts

B.1 Reload base de dados

```
DB_NAME=${1:-dbdump}
DUMP_FILE_FOLDER=${2:-/home}

echo "Dropping DB" && sudo -u postgres psql -U postgres -c "drop database $DB_NAME;"
echo "Creating DB" && sudo -u postgres psql -U postgres -c "create database $DB_NAME;"
echo "Restoring DB" && sudo -u postgres pg_restore -j 8 -d $DB_NAME -U postgres -Fc $DUMP_FILE_FOLDER/dump_file
```

B.2 Testes analíticos

```
#!/bin/bash

DATABASE_NAME=dbdump
RESULTS_FOLDER=results
NUMBER_OF_RUNS=${1:-3}

mkdir $RESULTS_FOLDER

echo "Starting at $(date +%T)"

for filename in tests/*.test.sql; do
    echo "Running $filename"

    IFS=" " read -a queries <<< $(head -n 1 $filename)

    echo "Running preparation script" && sudo sh -c "sudo -u postgres psql -U postgres -d $DATABASE_NAME -a -f $filename"


    for query in "${queries[@]:1}"; do # skip comment
        for queryfile in queries/$query*; do # for all queries that match
            for ((i=1; i<=$NUMBER_OF_RUNS; i++)); do
                /bin/bash clear_cache.sh $DATABASE_NAME
                echo "$(date +%T) - $(basename "$filename" .test.sql) - query $(basename "$queryfile" .sql) - Run $i "
                RESULTS_FILE_NAME="$RESULTS_FOLDER/$(basename "$filename" .test.sql)-$(basename "$queryfile" .sql)-$i.txt"

                echo "Running query" && sudo sh -c "sudo -u postgres psql -U postgres -d $DATABASE_NAME -a -f $queryfile > $RESULTS_FILE_NAME"
                echo "Took $(cat $RESULTS_FILE_NAME | awk '/Execution Time:/ {print $3 $4}')"
            done
        done
    done

    # echo "tests/£(basename "£filename" .test.sql).cleanup.sql"
    sudo echo "Running cleanup script" && sh -c "sudo -u postgres psql -U postgres -d $DATABASE_NAME -a -f tests/$(basename "$filename" .test.sql).cleanup.sql"
done

echo "Finished at $(date +%T)"
```

B.3 Testes transacionais

```
#!/bin/bash

CLIENTS=(1 4 8 16 24 32 48 64)
BENCHMARK_FOLDER=/home/imdbBench/transactional/
JAR_FOLDER=$BENCHMARK_FOLDER/target
TESTS_FOLDER=transactional-tests
RESULTS_FOLDER=transactional-results
DATABASE_NAME=dbdump
DB_DUMP_FOLDER=/home

mkdir $RESULTS_FOLDER
```

```
echo "Starting at $(date +%T)"

mvn -f $BENCHMARK_FOLDER/pom.xml clean package #build benchmark jar

echo "Running preparation script" && sudo sh -c "sudo -u postgres psql -U postgres -d $DATABASE_NAME -a -c \"ALTER USER postgres WITH PASSWORD 'postgres'\";"

for filename in $TESTS_FOLDER/*.test.sql; do
    echo "Running $filename"

    IFS=" " read -a queries <<< $(head -n 1 $filename) # load queries from first line of test file

    for c in "${CLIENTS[@]}"; do
        /bin/bash reload.sh $DATABASE_NAME $DB_DUMP_FOLDER
        echo "Running preparation script" && sudo sh -c "sudo -u postgres psql -U postgres -d $DATABASE_NAME -a -f $filename"
        /bin/bash clear_cache.sh $DATABASE_NAME
        echo "$(date +%T) - $(basename "$filename" .test.sql) - $c Clients "
        RESULTS_FILE_NAME="$RESULTS_FOLDER/$(basename "$filename" .test.sql)-$c.txt"

        # run benchmark
        java -jar $JAR_FOLDER/transactional-1.0-SNAPSHOT.jar -d jdbc:postgresql://localhost:5432/$DATABASE_NAME -U postgres -P postgres -W 30 -R 240 -c $c > $RESULTS_FILE_NAME

        # results
        echo "Took $(cat $RESULTS_FILE_NAME | awk -v ORS=";" '/(throughput)|(response)/  {print $NF}')"
    done

    sudo echo "Running cleanup script" && sh -c "sudo -u postgres psql -U postgres -d $DATABASE_NAME -a -f $TESTS_FOLDER/$(basename "$filename" .test.sql).cleanup.sql"
done

/bin/bash reload.sh $DATABASE_NAME $DB_DUMP_FOLDER # reload database

echo "Finished at $(date +%T)"
```

B.4 Clear cache

```
DB_NAME=${1:-dbdump}

echo "Stopping postgres service" && systemctl stop postgresql && \
echo "Clearing system cache" && sh -c "echo 3 > /proc/sys/vm/drop_caches" && \
echo "Restarting postgres service" && systemctl start postgresql && \
echo "Postgres vacuum" && sh -c "sudo -u postgres psql -U postgres -d $DB_NAME -c \"vacuum analyze;\""
```

Bibliografia

[1] postgresqlco.nf. (n.d.). PostgresqlCO.NF: PostgreSQL configuration for humans. [online] Available at: `https://postgresqlco.nf/tuning-guide.` [Accessed 30 May 2023]

[2] explain.dalibo.com. (n.d.). explain.dalibo.com. [online] Available at: `https://explain.dalibo.com/` [Accessed 30 May 2023].

[3] PostgreSQL Documentation. (2023). 20.4. Resource Consumption. [online] Available at: https://www.postgresql.org/docs/current/runtime-config-resource.html [Accessed 30 May 2023].

[4] spark.apache.org. (n.d.). PySpark Documentation — PySpark 3.1.1 documentation. [online] Available at: `https://spark.apache.org/docs/latest/api/python/`[Accessed 30 May 2023].