

UNIVERSIDADE DO MINHO

MESTRADO EM ENGENHARIA INFORMÁTICA
ADMINISTRAÇÃO DE BASE DE DADOS

IMDb benchmark



Diana Rodrigues
pg50320



Mariana Rodrigues
pg50622



Mariana Amorim
pg50623



Miguel Fernandes
pg50654



Conteúdo

1	Introdução	2
2	Otimizações das Interrogações Analíticas	3
2.1	Interrogação 1	3
2.2	Interrogação 2	6
2.3	Interrogação 3	10
3	Otimizações das Interrogações Analíticas no Spark	13
3.1	Interrogação 1	14
3.2	Interrogação 2	17
3.3	Interrogação 3	19
3.4	Considerações finais	22
4	Otimizações nas Interrogações Transacionais	23
4.1	Índices e Vistas	23
4.1.1	<i>addNewTitleToList</i>	23
4.1.2	<i>getWatchListInformation</i>	25
4.1.3	<i>viewTitle</i>	27
4.1.4	<i>rateTitle</i>	29
4.1.5	<i>searchTitles</i>	30
4.2	Melhorias obtidas	30
4.3	Parâmetros de configuração	31
4.3.1	<i>shared_buffers</i> e <i>work_mem</i>	31
4.3.2	<i>Write Ahead Log</i>	34
4.3.3	<i>Lock Management</i>	35
4.4	Melhorias obtidas	36
5	Conclusão	37
6	Anexos	38
6.1	Script <i>start_spark</i>	38
6.2	Script <i>shared_buffers</i>	38
6.3	Script <i>work_mem</i>	39
6.4	Script para <i>max_wal_size</i> e <i>min_wal_size</i>	40
6.5	Script para <i>wal_buffers</i> e <i>wal_writer_delay</i>	40
6.6	Script para <i>Lock Management</i>	42



1 Introdução

Este trabalho prático tinha como objetivo configurar, otimizar e avaliar o *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. Fornecendo operações para:

- Adicionar um título à lista "para ver" de um utilizador.
- Obter a lista "para ver" de um utilizador.
- Assinalar a visualização de um título por um utilizador.
- Adicionar a avaliação de um cliente a um título.
- Pesquisar títulos pelo nome.

Operações essas que foram alvo de otimizações de desempenho.

Por sua vez, as interrogações analíticas simulam operações estatísticas e de *data warehousing* sobre o conjunto de dados IMDb e sobre a informação adicional sobre os utilizadores. Pretendia-se, neste trabalho, otimizar o desempenho destas interrogações tanto no PostgreSQL como no Spark.



2 Otimizações das Interrogações Analíticas

Todos os resultados obtidos nesta secção não consideram a existência de *cache*, isto é, antes de executar cada *query* foi feito o *restart* do *PostgreSQL* e sempre que foi criado um índice ou vista foi corrido o comando *vacuum analyze*, de forma a garantir que as estatísticas estavam atualizadas.

2.1 Interrogação 1

O primeiro passo para a otimização da *query* passou pela compreensão dos objetivos da mesma. Após uma breve análise, o grupo chegou à conclusão que a *query* em causa retorna um *top 10* de filmes por cada década com base no seu *rating*. Para além disso, são considerados aspectos específicos dos filmes como o género e regiões. Neste ponto, sem qualquer otimização, esta *query* demorava cerca de 5 segundos (5 711.373 ms).

Para o *top* calculado são apenas considerados os filmes com mais de 3 avaliações, assim sendo, sempre que a *query* é executada é calculado o número de avaliações dos filmes. Tendo isso em conta, pensámos que poderia ser útil persistir esse número atualizado numa nova coluna da tabela *title*.

```
1 ALTER TABLE title add COLUMN rating_count int;
2
3 UPDATE title SET
4 rating_count = (SELECT COUNT(*) FROM UserHistory
5 WHERE title_id = title.id
6 AND rating IS NOT NULL);
```

Outra alternativa considerada pelo grupo foi a criação de uma vista materializada que permitisse persistir os mesmos dados. Esta abordagem permitiria uma manutenção da vista mais facilitada do que a manutenção da tabela *title*, uma vez que, para as vistas, é possível introduzir *triggers* que permitiriam que qualquer alteração na tabela *userhistory*, onde está a informação sobre *rating*, se refletisse automaticamente na vista.

Contudo, no momento de atualização da vista, todas as suas entradas são recalculadas. Assim, torna-se importante refletir sobre a recorrência dos eventos que despoletam estas atualizações. Neste caso específico, sempre que uma nova entrada fosse adicionada à tabela *userhistory*, todos os números de *ratings* seriam recalculados. O grupo considerou que a adição de entradas na tabela *userhistory* seria recorrente pelo que a criação de uma vista materializada não seria adequada.

Para que a *query* utilizasse esta nova coluna tivemos de a adaptar. Para tal, tivemos de retirar a linha onde era calculado o número de avaliações dos títulos:

```
1 HAVING count(uh.rating) >= 3
```

Para além disso, adicionamos a linha para filtrarmos os títulos com mais de 3 avaliações (linha 20):



```
1   ...
2   JOIN userHistory uh ON uh.title_id = t.id
3   WHERE t.title_type = 'movie'
4     AND ((start_year / 10) * 10)::int >= 1980
5   AND t.id IN (
6     SELECT title_id
7     FROM titleGenre tg
8     JOIN genre g ON g.id = tg.genre_id
9     WHERE g.name IN (
10       'Drama'
11     )
12   )
13   AND t.id IN (
14     SELECT title_id
15     FROM titleAkas
16     WHERE region IN (
17       'US', 'GB', 'ES', 'DE', 'FR', 'PT'
18     )
19   )
20   AND t.rating_count >= 3
21 GROUP BY t.id
22 ...
```

Para além disso, como o objetivo era filtrar os filmes pelo intervalo de número de avaliações deles (`t.rating_count >= 3`, neste caso), podemos utilizar um índice *btree* que facilita este tipo de pesquisas.

```
1 CREATE INDEX on title USING btree (rating_count);
```

Após estas alterações, o tempo de execução diminui consideravelmente para cerca de 2 segundos (2 080.435 ms). Ainda pela análise da *query*, tendo em conta o número de vezes em que é utilizada a coluna `start_year` da tabela `title`, o grupo entendeu que a pesquisa com base nesta coluna ficaria facilitada se fosse criado um índice onde esta estivesse incluída. Para tirar ainda mais partido desta otimização, decidimos associar ao `start_year` o `title_type`.

```
1 CREATE INDEX ON title USING btree (start_year DESC, title_type);
```

De seguida, de modo a percebemos em que parte da *query* existia mais potencial para otimização, decidimos verificar o estado do plano de execução da interrogação:



Figura 1: Plano de execução.

Analisando-o, inferimos que poderia ser vantajoso criar um índice na tabela `userhistory` pelo identificador título de modo a diminuir o *hotspot* identificado no plano:

```
1 CREATE INDEX ON userhistory USING HASH (title_id);
```

Esta alteração fez com que o tempo de execução diminuisse para sensivelmente 0.7 segundos (692.515 ms) e com que o plano de execução passasse a ser o seguinte:

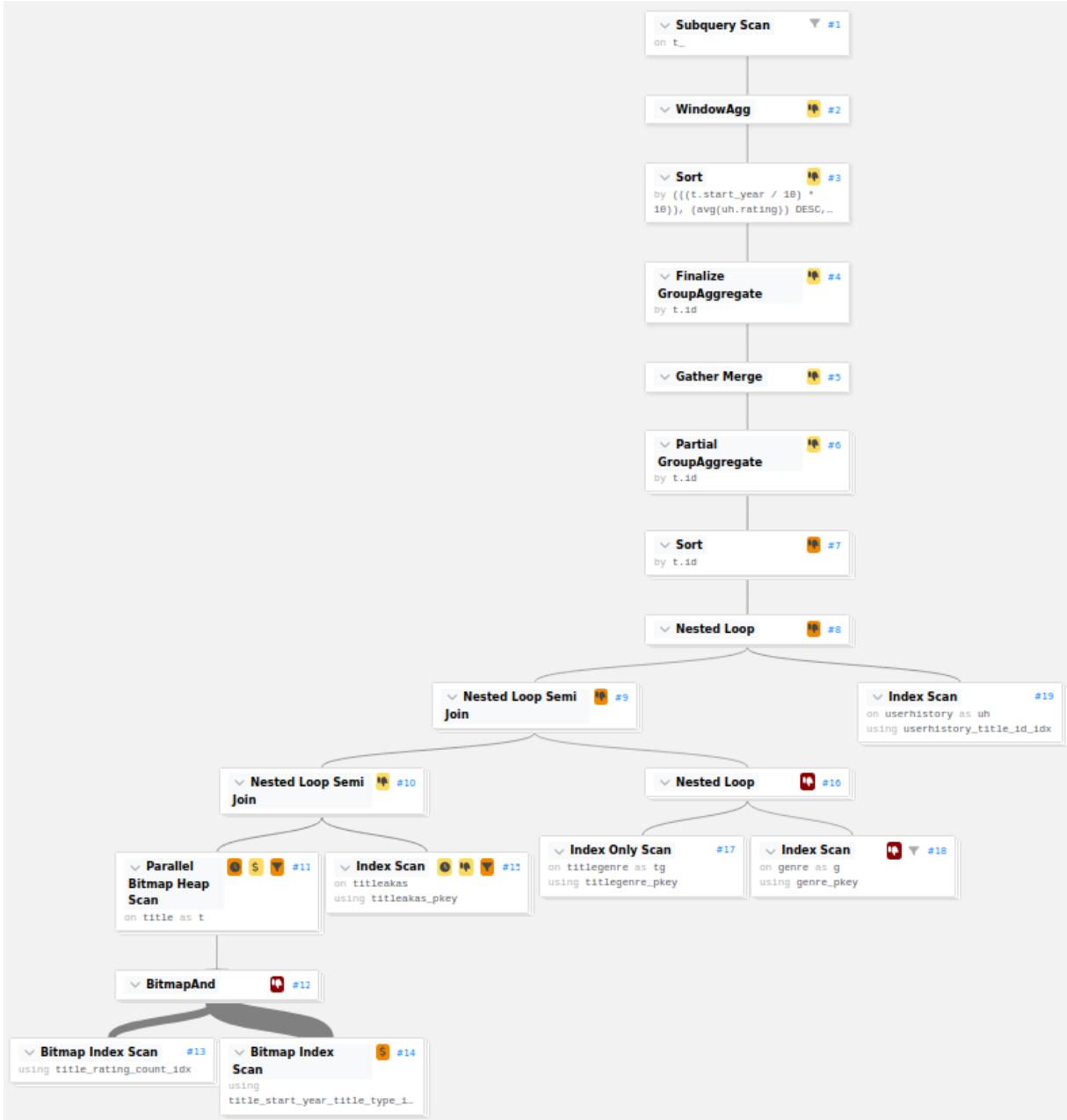


Figura 2: Plano de execução.

Assim, com as otimizações apresentadas conseguimos melhorar o tempo de execução de 5 segundos para 0.7 segundos, o que representa um ganho de 86%.

2.2 Interrogação 2

Seguindo a mesma estratégia utilizada para a primeira *query*, o grupo começou por analisar o retorno da *query 2*. Assim sendo, chegou-se à conclusão que retorna o *top 100* de séries televisivas com base no seu número de visualizações, e ainda com restrições de tempo relativo à visualização, temporada da série e os códigos associados aos países dos utilizadores associados às *views*. Sem qualquer otimização, esta interrogação demorava quase 22 segundos (21 620.257 ms) a executar.

Ao analisar o plano de execução desta *query* percebemos que existia uma parte consideravelmente mais custosa que todo o restante plano:

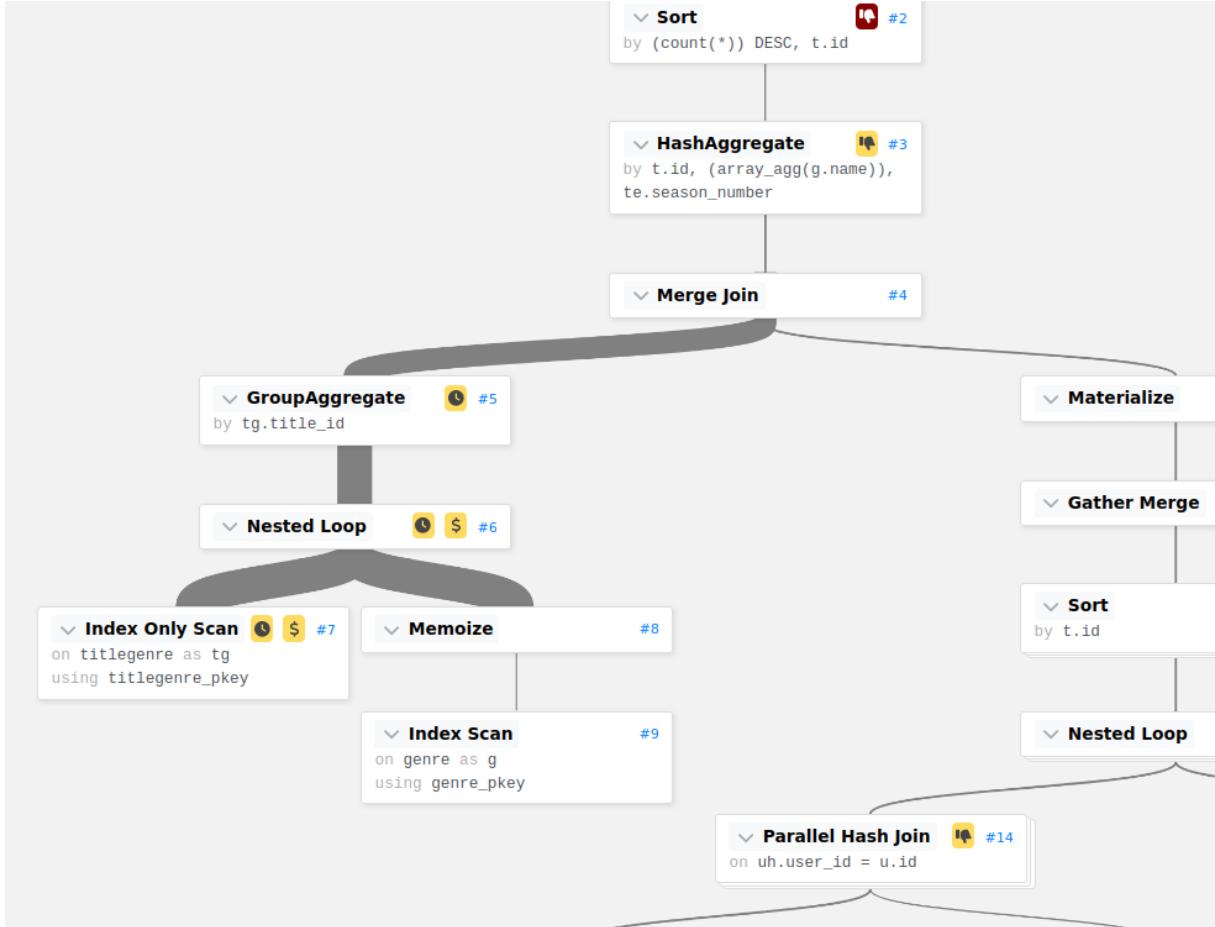


Figura 3: Plano de execução.

Percebemos que essa parte corresponde à seguinte parte da interrogação onde, para cada título, se verifica quais os nomes dos géneros em que o mesmo se integra:

```

1 JOIN (
2   SELECT tg.title_id, array_agg(g.name) AS genres
3   FROM titleGenre tg
4   JOIN genre g ON g.id = tg.genre_id
5   GROUP BY tg.title_id
6 ) tg ON tg.title_id = t.id
  
```

Consideramos a possibilidade de criar uma vista materializada onde a cada título estariam associados os géneros do mesmo. No entanto, tendo em conta que podem ser adicionados muitos filmes diariamente e cada uma dessas adições iria significar que seria feito um *refresh* da vista materializada, percebemos que esta estratégia não seria muito adequada.

Assim, pensamos em criar uma tabela com esse conteúdo pois, uma vez que não é considerada a possibilidade de adicionar ou remover os géneros associados a um título depois de adicionado,



as alterações a esta tabela seriam só a criação de uma nova entrada aquando da adição de um novo título. Assim, criamos a tabela da seguinte forma:

```
1 CREATE TABLE genreagg AS SELECT * FROM (
2     SELECT tg.title_id, array_agg(g.name) AS genres
3         FROM titleGenre tg
4     JOIN genre g ON g.id = tg.genre_id
5         GROUP BY tg.title_id) AS genreagg;
```

De seguida, foi necessário atualizar a *query* para que utilizasse a tabela criada:

```
1 SELECT t.id, t.primary_title, gagg.genres, te.season_number, count(*) AS views
2   FROM title t
3   JOIN titleEpisode te ON te.parent_title_id = t.id
4   JOIN title t2 ON t2.id = te.title_id
5   JOIN userHistory uh ON uh.title_id = t2.id
6   JOIN users u ON u.id = uh.user_id
7   JOIN genreagg gagg ON gagg.title_id = t.id
8 WHERE t.title_type = 'tvSeries'
9     AND uh.last_seen BETWEEN NOW() - INTERVAL '30 days' AND NOW()
10    AND te.season_number IS NOT NULL
11    AND u.country_code NOT IN ('US', 'GB')
12 GROUP BY t.id, t.primary_title, gagg.genres, te.season_number
13 ORDER BY count(*) DESC, t.id
14 LIMIT 100;
```

Com estas alterações conseguimos diminuir o tempo de execução da *query* para, aproximadamente, 9 segundos (8 936.928 ms) e, analisando o plano de execução, percebemos a existência de um grande custo associado ao *scan* sequencial da tabela criada:

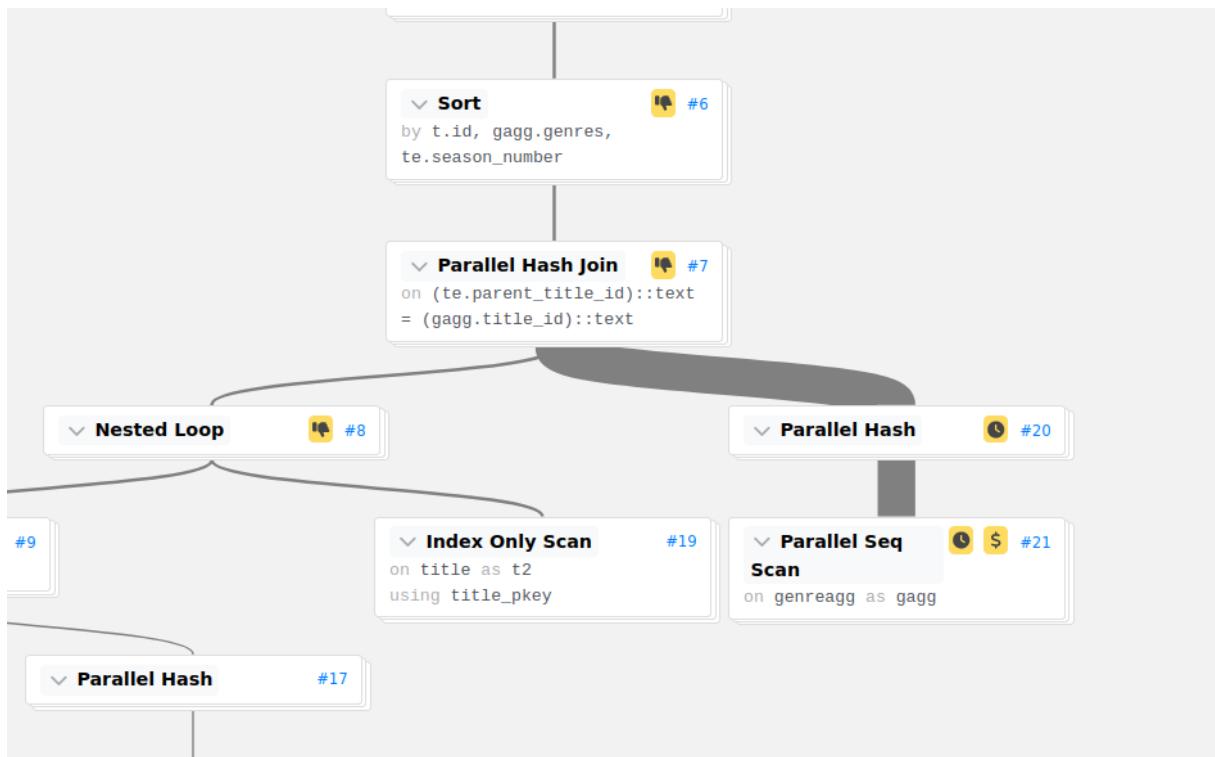


Figura 4: Plano de execução.

Deste modo, como a procura é feita pelo `title_id`, decidimos criar um índice para a facilitar:

```
1 CREATE INDEX ON genreagg USING HASH(title_id);
```

Após a criação deste índice pudemos verificar que o tempo de execução havia baixado para cerca de 6 segundos (5 785.702 ms).

Como referido inicialmente, esta interrogação filtra as visualizações dos títulos tendo em conta a data em que ocorreram, no caso específico desta *query* são consideradas apenas as visualizações dos últimos 30 dias. Assim, pareceu-nos que poderíamos tirar partido do uso de uma *btree* que facilita as pesquisas por intervalos, neste caso seria apenas necessário percorrer as folhas da árvore que correspondem a datas dos últimos 30 dias.

```
1 CREATE INDEX ON userhistory USING BTREE(last_seen DESC, title_id);
```

A criação deste índice permitiu obter um tempo de execução de cerca de 5 segundos (5 361.087 ms).

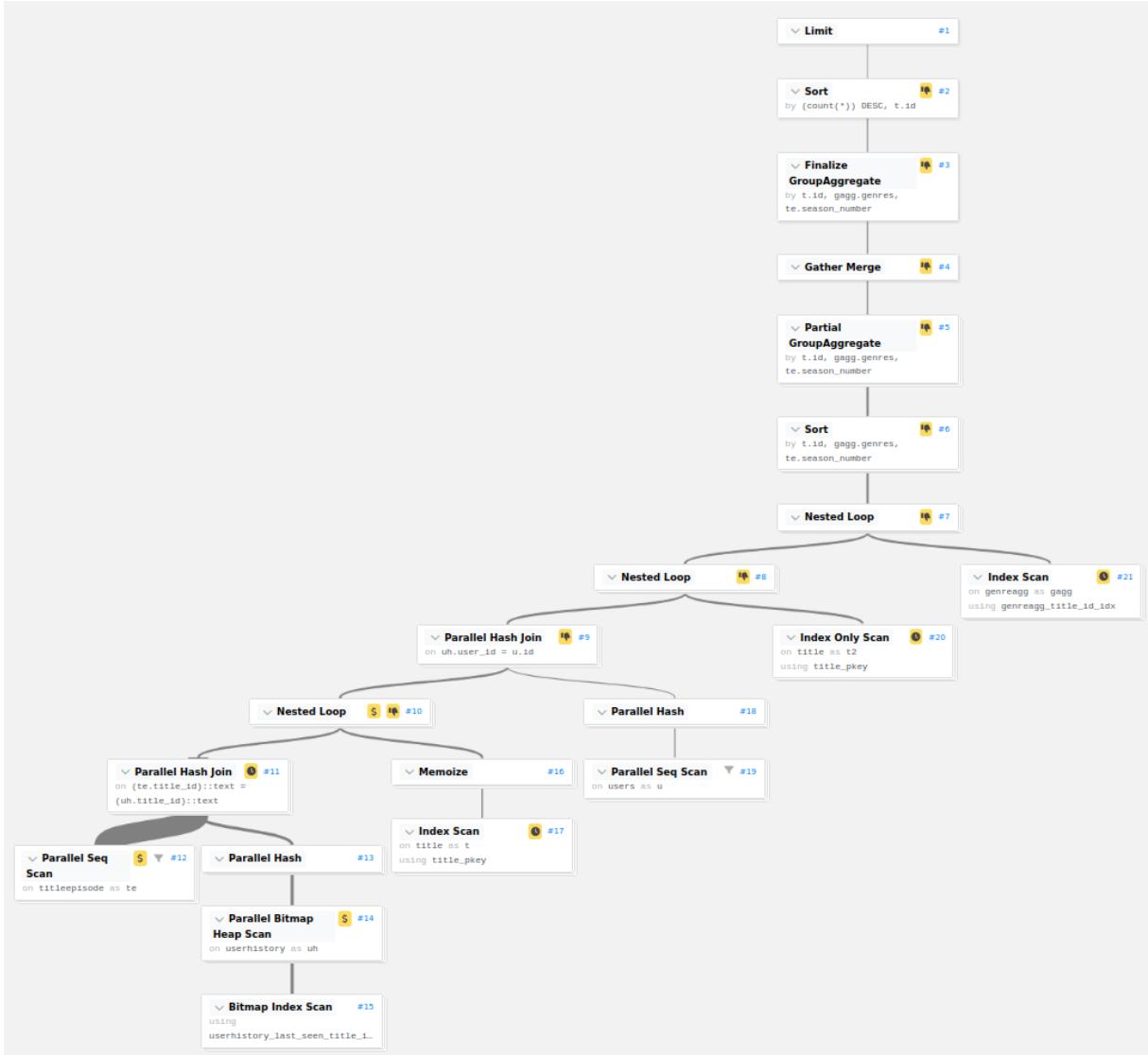


Figura 5: Plano de execução.

Consideradas todas as otimizações feitas, foi nos possível diminuir o tempo de 22 segundos para 5 segundos, o que se traduz num ganho de 77%.

2.3 Interrogação 3

De forma análoga às interrogações anteriores começamos por compreender que estatística era calculada por esta *query*. Neste caso, ela é utilizada para devolver informações sobre o *top* 100 de atrizes que ainda no ativo (vivas) e que fizeram títulos de uns determinados tipos nos últimos 10 anos. Inicialmente, o seu tempo de execução rondava os 91 segundos (90 950.453 ms).

Apercebemo-nos que sempre que esta *query* era executada incluía no *join* pessoas que não estavam no ativo que eram retiradas apenas no *where*. Assim, pensamos que podíamos beneficiar de ter uma vista materializada com as pessoas que já não estão no ativo:

```
1 CREATE MATERIALIZED VIEW deadpeople AS
2   SELECT id FROM name WHERE death_year IS NOT NULL;
```



Para além disso, fizemos as alterações necessárias à interrogação para que a mesma recorra à vista materializada. Assim retiramos a linha que verifica que a pessoa ainda estava no ativo:

```
1 AND n.death_year IS NULL
```

E substituímos por um *left join* combinado com um *where dp.id* que exclui as pessoas que não estão no ativo:

```
1 ...
2 LEFT JOIN titleEpisode te ON te.title_id = tp.title_id
3 LEFT JOIN deadpeople dp ON dp.id = n.id
4 WHERE t.start_year >= date_part('year', NOW())::int - 10
5     AND c.name = 'actress'
6     AND n.death_year IS NULL
7     AND t.title_type IN (
8         'movie', 'tvSeries', 'tvMiniSeries', 'tvMovie'
9     )
10    AND te.title_id IS NULL
11    AND dp.death_year IS NULL
12 GROUP BY n.id
13 ...
```

Esta vista levou o tempo de execução a diminuir para 30 segundos (29 499.337 ms) e originou o seguinte plano de execução.

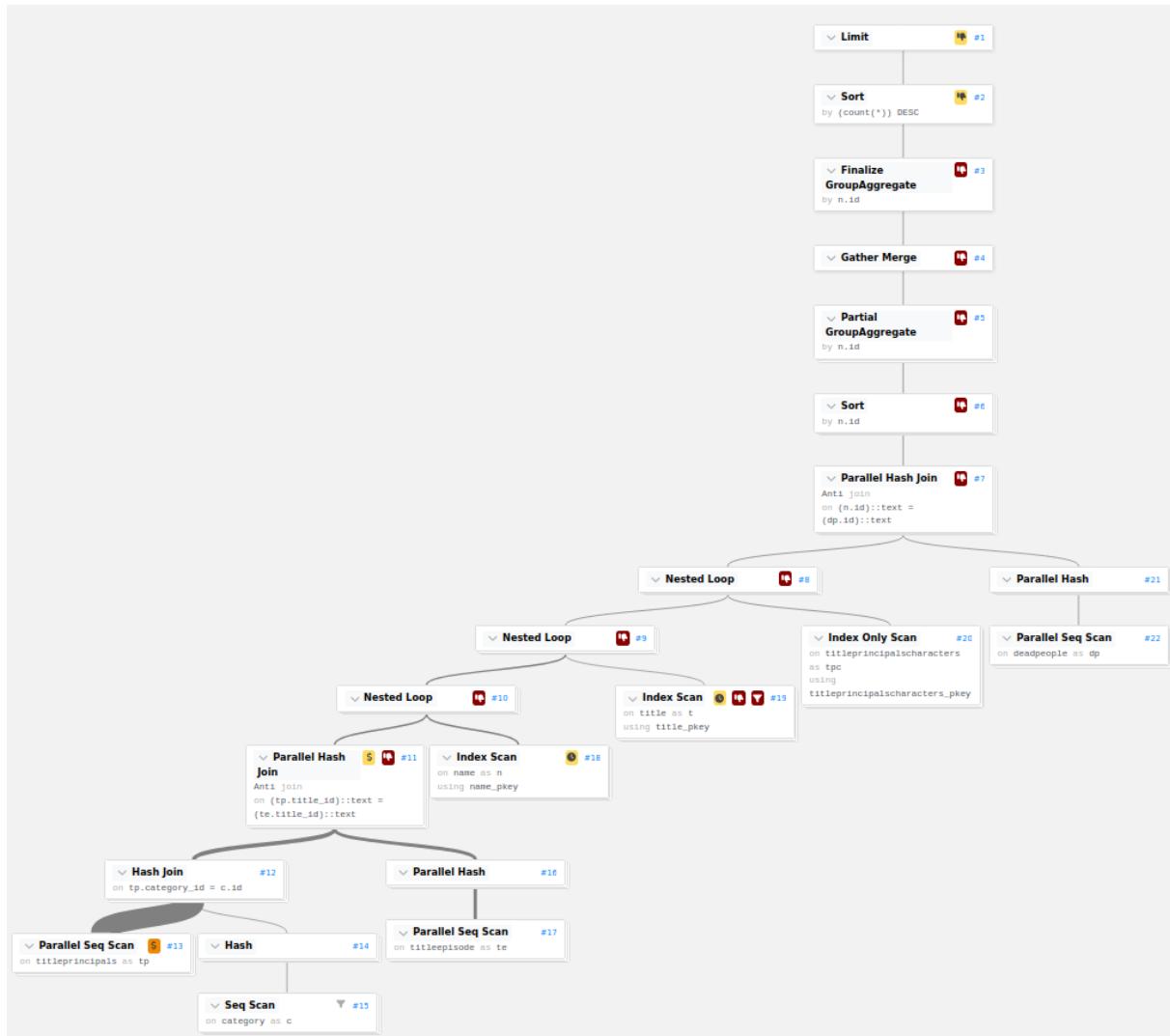


Figura 6: Plano de execução.

Assim sendo, conseguimos melhorar o tempo de execução de 91 segundos para 30 segundos, o que representa um ganho de 68%.



3 Otimizações das Interrogações Analíticas no Spark

De modo a conseguirmos realizar otimizações das interrogações analíticas no Spark, começámos por configurar e estabelecer uma sessão de Spark para interagir com um *cluster* Spark remoto. Isto permite a execução eficiente e escalável de operações e processamento distribuído de dados. A configuração inclui definições de endereço do *master* Spark, arquivo JAR para integração com o Google Cloud Storage e credenciais da conta de serviço do Google Cloud.

Na configuração do Spark é de salientar que foram ativadas três configurações:

- ***spark.eventLog.enable***: que ativa o registro de eventos, permitindo a monitorização e análise detalhada do desempenho das aplicações.
- ***spark.sql.cbo.enable***: que habilita o otimizador de *joins*. Melhora o desempenho das consultas ao determinar a melhor estratégia de *join* com base em estatísticas e custos.
- ***spark.sql.adaptive.enabled***: que ativa a execução adaptativa e, através disto, o Spark pode ajustar dinamicamente o plano de execução de consultas com base nas informações coletadas durante a execução.

```
1 spark = SparkSession.builder []
2   .master("spark://spark:7077") []
3   .config("spark.jars", "/app/gcs-connector-hadoop3-2.2.12.jar") []
4   .config("spark.driver.extraClassPath",
5     "/app/gcs-connector-hadoop3-2.2.12.jar") []
6   .config("spark.eventLog.enabled", "true") []
7   .config("spark.eventLog.dir", "/tmp/spark-events") []
8   .config("spark.sql.cbo.enable", "true") []
9   .config("spark.sql.adaptive.enabled", "true") []
10  .config("spark.executor.memory", "1g") []
11  .getOrCreate()
12
13  # google cloud service account credentials file
14  spark._jsc.hadoopConfiguration().set(
15    "google.cloud.auth.service.account.json.keyfile",
16    "/app/credentials.json")
```

Em primeiro lugar todas as tabelas do *PostgreSQL* foram convertidas para *csv*. Depois com o auxílio do *PySpark* estes ficheiros *csv* foram convertidos para *Parquet* e adicionados a um *Bucket* na *Google Cloud*, de maneira que estes apenas sejam gerados uma vez.

De modo a evitar a repetição e melhorar a legibilidade, o código abaixo foi criado de forma a ser possível ler ficheiros Parquet. Estes contém dados de diferentes categorias, gêneros, nomes de títulos, informações de histórico do utilizador, entre outros, que serão utilizados nas análises subsequentes.

```
1 # bucket name
2 BUCKET_NAME = 'imdbbench'
3
4 # read parquet files
```



```
5 category = spark.read.parquet(f"gs://{BUCKET_NAME}/category.parquet")
6 genre = spark.read.parquet(f"gs://{BUCKET_NAME}/genre.parquet")
7 name = spark.read.parquet(f"gs://{BUCKET_NAME}/name.parquet")
8 titleAkas = spark.read.parquet(f"gs://{BUCKET_NAME}/titleakas.parquet")
9 titleEpisode = spark.read.parquet(f"gs://{BUCKET_NAME}/titleepisode.parquet")
10 titleGenre = spark.read.parquet(f"gs://{BUCKET_NAME}/titlegenre.parquet")
11 titlePrincipals = spark.read.parquet(
12     f"gs://{BUCKET_NAME}/titleprincipals.parquet")
13 titlePrincipalsCharacters = spark.read.parquet(
14     f"gs://{BUCKET_NAME}/titleprincipalscharacters.parquet")
15 userHistory = spark.read.parquet(f"gs://{BUCKET_NAME}/userhistory.parquet")
16 users = spark.read.parquet(f"gs://{BUCKET_NAME}/users.parquet")
17 title = spark.read.csv(
18     f"gs://{BUCKET_NAME}/title.csv", header=True, inferSchema=True)
```

Todos os testes realizados nesta secção utilizaram três *workers*. De modo a agilizar o processo de iniciação do Spark e da criação dos *containers* na máquina virtual foi criado o *script* 6.1 "*start_spark*".

3.1 Interrogação 1

Primeiramente, começámos por converter a *query* original para o formato do *spark* sem considerarmos possíveis otimizações. Tendo ficado assim:

```
1 rank_window = Window.partitionBy(col("decade"))
2     .orderBy(col("rating_avg").desc(), title["id"])
3
4 query1 = title [
5     .join(userHistory, userHistory["title_id"] == title["id"])
6     .where(title["title_type"] == "movie")
7     .where((floor(title["start_year"] / 10) * 10).cast("int") >= 1980)
8     .join(titleGenre.join(genre, genre["id"] == titleGenre["genre_id"]))
9     .where((genre["name"]).isin(["Drama"]))
10    .select(titleGenre["title_id"]), title["id"] ==
11        titleGenre["title_id"], "inner")
12        .join(titleAkas[titleAkas["region"].isin("US", "GB", "ES", "DE", "FR",
13            "PT")])
14        .select(titleAkas["title_id"]), title["id"] ==
15        titleAkas["title_id"], "inner")
16        .groupBy(title["id"], title["primary_title"], (floor(title["start_year"] /
17            10) * 10)
18            .cast("int"))
19        .agg(title["id"],
20            title["primary_title"],
21            (floor(title["start_year"] / 10)
22            * 10).cast("int").alias("decade"),
23            avg(userHistory["rating"].cast("double")).alias("rating_avg"),
24            count(userHistory["rating"]).alias("rating_count"))
25        )
26        .where(col("rating_count") >= 3)
```



```
23     .withColumn("rank", rank().over(rank_window)) []
24     .orderBy(col("decade"), col("rating_avg").desc()) []
25     .select(title["id"],
26             title["primary_title"].substr(1, 30).alias("left"),
27             col("decade"),
28             col("rating_avg"),
29             col("rank")) []
30     .where(col("rank") <= 10)
31
32 query1.show(100, False)
```

Após a conversão, realizámos algumas otimizações para reduzirmos a quantidade de dados processados e melhorarmos o desempenho da consulta.

De seguida, aplicámos os filtros "where" antes dos *joins*, limitando assim a quantidade de dados a ser processados nas operações subsequentes. Uma outra otimização que fizemos foi alterar a ordem pelo qual os "*joins*" estavam a ser feitos, a fim de conseguirmos realizar os *joins* com tabelas potencialmente menores, uma vez que as junções anteriores já aplicaram filtros.

```
1 rank_window = Window.partitionBy(col("decade")) []
2     .orderBy(col("rating_avg").desc(), title["id"])
3
4 query1 = title []
5     .where(title["title_type"] == "movie") []
6     .where((floor(title["start_year"] / 10) * 10).cast("int") >= 1980) []
7     .join(titleGenre.join(genre, genre["id"] == titleGenre["genre_id"]))
8         .where((genre["name"]).isin(["Drama"]))
9         .select(titleGenre["title_id"]), title["id"] ==
10            titleGenre["title_id"], "inner") []
11     .join(titleAkas[titleAkas["region"].isin("US", "GB", "ES", "DE", "FR",
12    "PT")])
13         .select(titleAkas["title_id"]), title["id"] ==
14            titleAkas["title_id"], "inner") []
15         .join(userHistory, userHistory["title_id"] == title["id"]) []
16         .groupBy(title["id"], title["primary_title"], (floor(title["start_year"] /
17            10) * 10)
18             .cast("int"))
19             .agg(title["id"],
20                 title["primary_title"],
21                 (floor(title["start_year"] / 10)
22                 * 10).cast("int").alias("decade"),
23                 avg(userHistory["rating"].cast("double")).alias("rating_avg"),
24                 count(userHistory["rating"]).alias("rating_count")
25             ) []
26             .where(col("rating_count") >= 3) []
27             .withColumn("rank", rank().over(rank_window)) []
28             .orderBy(col("decade"), col("rating_avg").desc()) []
29             .select(title["id"],
30                     title["primary_title"].substr(1, 30).alias("left"),
```



```
27         col("decade"),
28         col("rating_avg"),
29         col("rank")) \n
30     .where(col("rank") <= 10)
31
```

Com isto, conseguimos observar uma melhoria na execução da *query* em questão (48.406 segundos para 31.590 segundos).

Uma outra otimização realizada foi a substituição das cláusulas "*where*" por "*filter*". A cláusula "*filter*" foi utilizada pois é um método otimizado do *Spark* que oferece um desempenho superior em comparação com o uso de "*where*".

Por último, reorganizamos a cláusula "*orderBy*" para que fosse aplicada após o último "*where*". Esta modificação visa reduzir a quantidade de dados processados antes da ordenação, pois, ao posicionarmos o "*orderBy*" por último, garantimos que apenas os registos relevantes vão ser ordenados, evitando o processamento desnecessário de dados não utilizados posteriormente.

Com todas as otimizações aplicadas, a consulta final no *Spark* apresentou um desempenho superior, passando para 29.516 segundos. Esta ficou da seguinte forma:

```
1  filteredTitle = title \n
2  .filter(((floor(title["start_year"] / 10) * 10).cast("int") >= 1980) &
   ↵  (title["title_type"] == "movie")) \n
3
4  query1 = filteredTitle \n
5      .join(titleAkas[titleAkas["region"].isin("US", "GB", "ES", "DE", "FR",
   ↵  "PT")])
6          .select(titleAkas["title_id"]), filteredTitle["id"] ==
   ↵  titleAkas["title_id"], "inner") \n
7      .join(titleGenre.join(genre, genre["id"] == titleGenre["genre_id"]))
8          .where((genre["name"]).isin(["Drama"]))
9          .select(titleGenre["title_id"]), filteredTitle["id"] ==
   ↵  titleGenre["title_id"], "inner") \n
10     .join(userHistory, userHistory["title_id"] == filteredTitle["id"]) \n
11     .groupBy(filteredTitle["id"], filteredTitle["primary_title"],
   ↵  (floor(filteredTitle["start_year"] / 10) * 10).cast("int")) \n
12     .agg(filteredTitle["id"],
   ↵  filteredTitle["primary_title"],
13      (floor(filteredTitle["start_year"] / 10)
   ↵  * 10).cast("int").alias("decade"),
14      avg(userHistory["rating"].cast("double")).alias("rating_avg"),
15      count(userHistory["rating"]).alias("rating_count")
   ↵ ) \n
16      .where(col("rating_count") >= 3) \n
17      .withColumn("rank", rank().over(rank_window)) \n
18      .select(filteredTitle["id"],
   ↵  filteredTitle["primary_title"].substr(1, 30).alias("left"),
   ↵  col("decade"),
```



```
24         col("rating_avg"),  
25         col("rank"))  
26     .where(col("rank") <= 10)  
27     .orderBy(col("decade"), col("rating_avg").desc())
```

A partir do *Spark UI* foi possível comparar a carga de trabalho de cada *worker*. O resultado obtido encontra-se na figura 7 e mostra que todos os *workers* têm a mesma carga de trabalho, isto é, os *task time* são semelhantes. Todos os *workers* realizarem a mesma carga de trabalho é importante para eficiência da *query*.

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs
0	172.18.0.5:45667	Active	0	0.0 B / 413.9 MiB	0.0 B	1	0	0	12	12	1.2 min (3 s)	455.7 MiB	59.4 MiB	56.6 MiB	stdout stderr
driver	9c9067f0ac4e:39953	Active	0	0.0 B / 434.4 MiB	0.0 B	0	0	0	0	0	1.8 min (0.0 ms)	0.0 B	0.0 B	0.0 B	
1	172.18.0.2:44403	Active	0	0.0 B / 413.9 MiB	0.0 B	1	0	0	12	12	1.3 min (3 s)	465.4 MiB	59.4 MiB	62.1 MiB	stdout stderr
2	172.18.0.3:34335	Active	0	0.0 B / 413.9 MiB	0.0 B	1	0	0	16	16	1.3 min (3 s)	572.6 MiB	61.9 MiB	61.9 MiB	stdout stderr

Figura 7: Balanceamento da carga de trabalho dos *workers* na *query* 1

3.2 Interrogação 2

Primeiramente, como feito na *query* anterior, começámos por converter a *query* original para o formato do *spark*. Tendo obtido o tempo de 73.687 segundos.

```
1  
2     title.alias("t")  
3     .join(titleEpisode.alias("te"), col("t.id") == col("te.parent_title_id"))  
4     .join(title.alias("t2"), col("t2.id") == col("te.title_id"))  
5     .join(userHistory.alias("uh"), col("uh.title_id") == col("t2.id"))  
6     .join(users.alias("u"), col("u.id") == col("uh.user_id"))  
7     .join((titleGenre.alias("tg").join(genre.alias("g"), col("g.id") ==  
    ↪ col("tg.genre_id")))  
8         .select(col("tg.title_id"), col("g.name"))  
9         .groupBy(col("tg.title_id"))  
10  
    ↪ .agg(collect_list(col("g.name")).alias("genres")).alias("tg"),  
11        col("t.id") == col("tg.title_id"))  
12     .where(col("t.title_type") == "tvSeries")  
13     .where(col("uh.last_seen").between(date_sub(current_timestamp(), 30),  
    ↪ current_timestamp()))  
14     .filter(col("te.season_number").isNotNull())  
15     .filter(~col("u.country_code").isin("US", "GB"))  
16     .select(col("t.id"), col("t.primary_title"), col("tg.genres"),  
    ↪ col("te.season_number"))  
17     .groupBy(col("t.id"), col("t.primary_title"), col("tg.genres"),  
    ↪ col("te.season_number"))  
18     .agg(count("*").alias("views"))  
19     .orderBy(col("views").desc(), col("t.id"))  
20     .limit(100)
```



```
21 .show()
```

Após a conversão, começámos por tentar reduzir a quantidade de dados a serem processados, realizando os *filter* antes dos *joins*, conseguindo com isto limitar a quantidade de dados a ser processados nas operações subsequentes.

```
1 title_filtered = title [
2     .where(title["title_type"] == "tvSeries") [
3         .select(title["id"], title["primary_title"]) [
4
5 userHistory_filtered = userHistory [
6     .where(userHistory["last_seen"].between(date_sub(current_timestamp(), 30),
7     ↪ current_timestamp()))) [
8         .select(userHistory["user_id"], userHistory["title_id"])
9
10 titleEpisode_filtered = titleEpisode [
11     .filter(titleEpisode["season_number"].isNotNull()) [
12         .select(titleEpisode["title_id"],
13             ↪ titleEpisode["parent_title_id"], titleEpisode["season_number"])
14
15 users_filtered = users [
16     .filter(~users["country_code"].isin("US", "GB")) [
17         .select(users["id"])
18
19 genre_filtered = genre [
20     .filter(genre["name"].isin("Drama")) [
21         .select(genre["id"], genre["name"])
22
23 titleGenre_filtered = titleGenre [
24     .join(genre_filtered, genre_filtered["id"] == titleGenre["genre_id"])) [
25         .groupBy(titleGenre["title_id"])) [
26             .agg(collect_list(genre_filtered["name"]).alias("genres")) [
27                 .select(titleGenre["title_id"], col("genres"))
28
29 title_filtered.alias("t") [
30     .join(titleEpisode_filtered.alias("te"), col("te.parent_title_id") ==
31     ↪ col("t.id")) [
32         .join(title.alias("t2"), col("t2.id") == col("te.title_id")) [
33             .join(userHistory_filtered.alias("uh"), col("uh.title_id") ==
34             ↪ col("t2.id")) [
35                 .join(users_filtered.alias("u"), col("u.id") == col("uh.user_id")) [
36                     .join(titleGenre_filtered.alias("tg"), col("tg.title_id") == col("t.id"))
37
38             .groupBy(col("t.id"), col("t.primary_title"), col("tg.genres"),
39             ↪ col("te.season_number")) [
40                 .agg(count("*").alias("views"))
41
42                 .orderBy(col("views").desc(), title_filtered["id"]))
43
44             .show()
```



Através disto, fomos capazes de reduzir de 73.680 segundos para 53.427 segundos.

Por último decidimos alterar a ordem do *join* do *dataframe* "title" com o "userHistory_filtered", colocando este por último na medida em que o *dataframe* "title" não se encontra previamente filtrado.

```
1 users_filtered.alias("u") \n
2     .join(userHistory_filtered.alias("uh"), col("u.id") == col("uh.user_id")) \n
3         \n
4             .join(titleEpisode_filtered.alias("te"), col("te.title_id") == \n
5                 col("uh.title_id")) \n
6                 .join(title_filtered.alias("t"), col("te.parent_title_id") == col("t.id")) \n
7                     \n
8                         .join(titleGenre_filtered.alias("tg"), col("tg.title_id") == col("t.id")) \n
9                             \n
10                                .join(title.alias("t2"), col("t2.id") == col("uh.title_id")) \n
11                                    .groupBy(col("t.id"), col("t.primary_title"), col("tg.genres"), \n
12                                        col("te.season_number")) \n
13                                            .agg(count("*").alias("views")) \n
14                                                .orderBy(col("views").desc(), "t.id") \n
15                                                    .show()
```

Tal como na query anterior verificou-se que todos os *workers* têm a mesma carga de trabalho (ver figura 8).

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs
0	172.18.0.5:35823	Active	0	0.0 B / 413.9 MiB	0.0 B	1	0	0	22	22	1.4 min (3 s)	728.3 MiB	76.7 MiB	63.6 MiB	stdout stderr
driver	9c9067f0ac4e:41721	Active	0	0.0 B / 434.4 MiB	0.0 B	0	0	0	0	0	1.9 min (0.0 ms)	0.0 B	0.0 B	0.0 B	
1	172.18.0.2:46463	Active	0	0.0 B / 413.9 MiB	0.0 B	1	0	0	20	20	1.3 min (2 s)	833.1 MiB	63.5 MiB	76 MiB	stdout stderr
2	172.18.0.3:45949	Active	0	0.0 B / 413.9 MiB	0.0 B	1	0	0	17	17	1.3 min (2 s)	606.1 MiB	78.7 MiB	79.3 MiB	stdout stderr

Figura 8: Balanceamento da carga de trabalho dos *workers* na *query* 2

Conseguimos, assim, melhorar esta *query* de 73.680 s para 45.507 s.

3.3 Interrogação 3

Repetindo o processo anterior, começámos por converter a *query* para o formato do *spark*, tendo obtido 173.480 segundos.

```
1 ten_years = 365 * 10
2
3 name.alias("n") \
4     .join(titlePrincipals.alias("tp"), col("tp.name_id") == col("n.id")) \
5     .join(titlePrincipalsCharacters.alias("tpc"), (col("tpc.title_id") ==
6         col("tp.title_id")) & (col("tpc.name_id") == col("tp.name_id"))) \
7             .join(category.alias("c"), col("c.id") == col("tp.category_id")) \
8                 .join(title.alias("t"), col("t.id") == col("tp.title_id")) \n
```



```
8     .join(titleEpisode.alias("te"), col("te.title_id") == col("tp.title_id"),
  ↵ "left") [
  ↵     .where(col("t.start_year") >= year(date_sub(current_timestamp(),
  ↵ ten_years))) [
  ↵     .where(col("c.name") == "actress") [
  ↵     .filter(col("n.death_year").isNull()) [
  ↵     .filter(col("t.title_type").isin("movie", "tvSeries", "tvMiniSeries",
  ↵ "tvMovie")) [
  ↵     .filter(col("te.title_id").isNull()) [
  ↵     .select(col("n.id"), col("n.primary_name"), (year(current_timestamp()) -
  ↵ col("n.birth_year")).alias("age")) [
  ↵     .groupBy(col("n.id"), col("n.primary_name"), col("age")) [
  ↵     .agg(count("*").alias("roles")) [
  ↵     .orderBy(col("roles").desc()) [
  ↵     .limit(100) [
  ↵     .show()
```

De seguida, alterou-se a execução dos filtros "*filter*" e "*where*" para antes dos *joins* de modo a reduzir a quantidade de dados a serem processados. Além disso, fez-se uma reordenação dos *joins*.

```
1 title_filtered = title [
  ↵     .where(title["start_year"] >= year(date_sub(current_timestamp(),
  ↵ ten_years))) [
  ↵     .filter(title["title_type"].isin("movie", "tvSeries", "tvMiniSeries",
  ↵ "tvMovie")) [
  ↵     .select(title["id"])

  ↵
  ↵ category_filtered = category [
  ↵     .where(category["name"] == "actress") [
  ↵     .select(category["id"])

  ↵
  ↵ name_filtered = name [
  ↵     .filter(name["death_year"].isNull()) [
  ↵     .select(name["id"], name["primary_name"], (year(current_timestamp()) -
  ↵ name["birth_year"]).alias("age"))

  ↵
  ↵ title_filtered.alias("t") [
  ↵     .join(titlePrincipals.alias("tp"), col("tp.title_id") == col("t.id")) [
  ↵     .join(category_filtered.alias("c"), col("c.id") == col("tp.category_id")) [
  ↵     [
  ↵         .join(name_filtered.alias("n"), col("n.id") == col("tp.name_id")) [
  ↵         .join(titleEpisode.alias("te"), col("te.title_id") == col("tp.title_id"),
  ↵ "left") [
  ↵             .filter(col("te.title_id").isNull()) [
  ↵             .join(titlePrincipalsCharacters.alias("tpc"), (col("tpc.title_id") ==
  ↵ col("tp.title_id")) & (col("tpc.name_id") == col("tp.name_id"))) [
  ↵             .groupBy(col("n.id"), col("n.primary_name"), col("age")) [
  ↵             .agg(count("*").alias("roles")) ]
```



```
23     .orderBy(col("roles").desc()) \n
24     .limit(100) \n
25     .show()
```

Com estas otimizações foi possível obter 111.88 segundos, conseguindo com isto melhorar 36%.

Numa última análise, decidiu-se alterar todos os *wheres* para *filters* de modo a manter a consistência no código e utilizar o método *join* com uma lista de condições em vez de termos múltiplos &. Esta abordagem melhora a legibilidade do código e facilita a sua manutenção, especialmente quando temos várias condições de *join*.

Fora isso, evitámos realizar o método *groupBy* antes do método *agg*. Esta mudança ajuda a otimizar a consulta, pois o *agg* já realiza a agregação dos dados e a operação de *groupBy* é aplicada apenas aos resultados finais. Isto pode resultar num desempenho mais eficiente, especialmente quando lidamos com grandes volumes de dados.

```
1  title_filtered = title \n
2      .filter(title["start_year"] >= year(date_sub(current_timestamp(), \n
3          ten_years))) \n
4      .filter(title["title_type"].isin("movie", "tvSeries", "tvMiniSeries", \n
5          "tvMovie")) \n
6      .select(title["id"])
7
8  category_filtered = category \n
9      .filter(category["name"] == "actress") \n
10     .select(category["id"])
11
12 name_filtered = name \n
13     .filter(name["death_year"].isNull()) \n
14     .select(name["id"], name["primary_name"], (year(current_timestamp()) - \n
15         name["birth_year"]).alias("age"))
16
17 title_filtered.alias("t") \n
18     .join(titlePrincipals.alias("tp"), col("tp.title_id") == col("t.id")) \n
19     .join(category_filtered.alias("c"), col("c.id") == col("tp.category_id")) \n
20
21     .join(name_filtered.alias("n"), col("n.id") == col("tp.name_id")) \n
22     .join(titleEpisode.alias("te"), col("te.title_id") == col("tp.title_id"), \n
23         "left") \n
24     .filter(col("te.title_id").isNull()) \n
25     .join(titlePrincipalsCharacters.alias("tpc"), \n
26         [col("tp.title_id") == col("tpc.title_id"), \n
27             col("tp.name_id") == col("tpc.name_id")]) \n
28     .groupBy(col("n.id"), col("n.primary_name"), col("age")) \n
29     .agg(count("*").alias("roles")) \n
30     .orderBy(col("roles").desc()) \n
31     .limit(100) \n
32     .show()
```



Mais uma vez, com o recurso do *Spark UI* verificou-se que todos as *queries* têm a mesma *workload* (ver figura 9).

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs
0	172.18.0.5:42883	Active	0	0.0 B / 413.9 MiB	0.0 B	1	0	0	26	26	2.4 min (4 s)	686.4 MiB	459.7 MiB	609.2 MiB	stdout stderr
driver	9c9067f0ac4e:40687	Active	0	0.0 B / 434.4 MiB	0.0 B	0	0	0	0	0	3.0 min (0.0 ms)	0.0 B	0.0 B	0.0 B	
1	172.18.0.2:33951	Active	0	0.0 B / 413.9 MiB	0.0 B	1	0	0	28	28	2.5 min (4 s)	784.1 MiB	672 MiB	537.3 MiB	stdout stderr
2	172.18.0.3:35745	Active	0	0.0 B / 413.9 MiB	0.0 B	1	0	0	25	25	2.4 min (4 s)	721.9 MiB	630.5 MiB	615.7 MiB	stdout stderr

Figura 9: Balanceamento da carga de trabalho dos *workers* na *query 3*

Alcançando, com isto, uma melhoria de $\approx 41\%$, passando de 173.48 segundos para 102.29 segundos.

3.4 Considerações finais

Como consideração final, uma melhoria que poderia ter sido explorada seria o uso do particionamento por colunas com o formato de ficheiro Parquet, utilizando a função ‘*partitionBy*’. O particionamento por colunas permite dividir os dados em diretorias com base nos valores de uma determinada coluna, o que poderia ter melhorado significativamente o desempenho das consultas ao restringir a leitura apenas a essas diretorias relevantes. O formato Parquet é altamente eficiente em termos de compactação e leitura seletiva de colunas, o que resulta em tempos de resposta mais rápidos e menor uso de recursos. Ao combinar estas técnicas, é possível otimizar ainda mais o processamento e análise dos dados.



4 Otimizações nas Interrogações Transacionais

4.1 Índices e Vistas

Tal como foi feito na Otimização das Interrogações Analíticas (secção 2) todos os resultados apresentados não consideram existência de *cache* e foi utilizado o comando *vacuum analyze* para manter as estatísticas atualizadas.

4.1.1 addNewTitleToList

A operação *addNewTitleToList* permite adicionar um título (filme, episódio, vídeo, etc.) à lista de títulos “para ver” de um utilizador. Um título pode ser adicionado de três diferentes formas:

- com base no género preferido do utilizador (*FromPreference*).
- dado tipo, adicionar um título recente (*FromType*).
- adicionar um título com base na sua popularidade na semana anterior (*FromPopular*).

getTitleFromPreference

A partir do plano de execução gerado com os parâmetros `ug.user_id = '500'` e `t_.user_id = '100000'` consegue encontrar-se dois *Seq Scan* em que podem ser explorados índices.

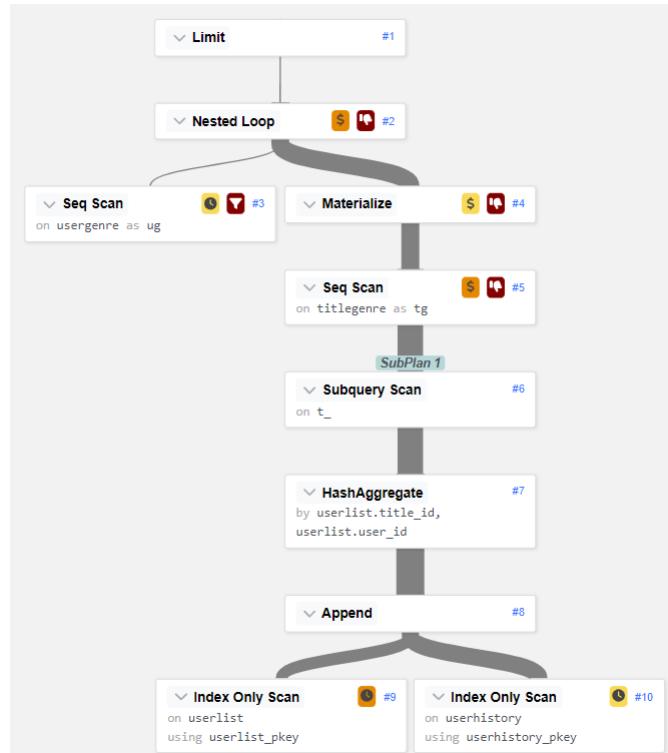


Figura 10: Plano de execução da query *getTitleFromPreference*

Na tabela `titleGenre` é apenas composta por 2 colunas, em que o `genre_id` apenas tem 28 valores distintos para mais de 9 milhões distintos no `title_id`, por essa razão criar um índice pelo `genre_id` irá ser pouco eficiente, dado que vão existir muitas colisões. A tabela `userGenre`



também só tem 2 valores, porém a discrepância entre valores distintos é menor, por isso foi criado o índice hash(user_id) que melhorou o tempo de execução de 7.636 ms para 0.629 ms.

getTitleFromType

De forma a analisar o plano gerado pela *query*, foram utilizados como valores title_type = 'movie' e user_id = '100000', obtendo o plano visível na figura 11. Neste plano é notável qual a operação mais custosa, neste caso o *Seq Scan* em *title*. O *title* está a ser filtrado pelas colunas start_year e title_type, sendo uma otimização possível um índice composto com estas duas variáveis.

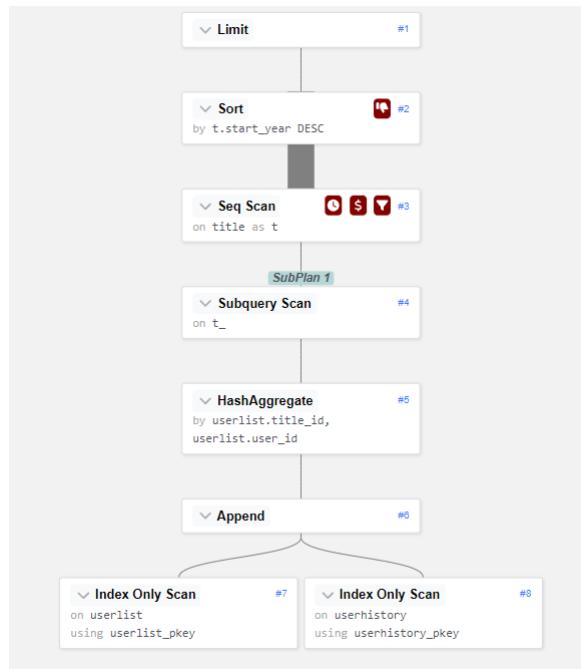


Figura 11: Plano de execução da *query* *getTitleFromType*

Contando o número de valores distintos das duas colunas obtemos que title_type apresenta 11 valores distintos enquanto que o start_year tem 152. Por essa razão criou-se um índice composto com btree (start_year, title_type). Com esta otimização o tempo de execução passou de 2680.516 ms para 364.273 ms.

getTitleFromPopular

O parâmetro utilizado para obter o plano da figura 12 foi t_.user_id = '100000'. O *Seq Scan* no *userHistory* foi para filtrar com base na coluna last_seen.

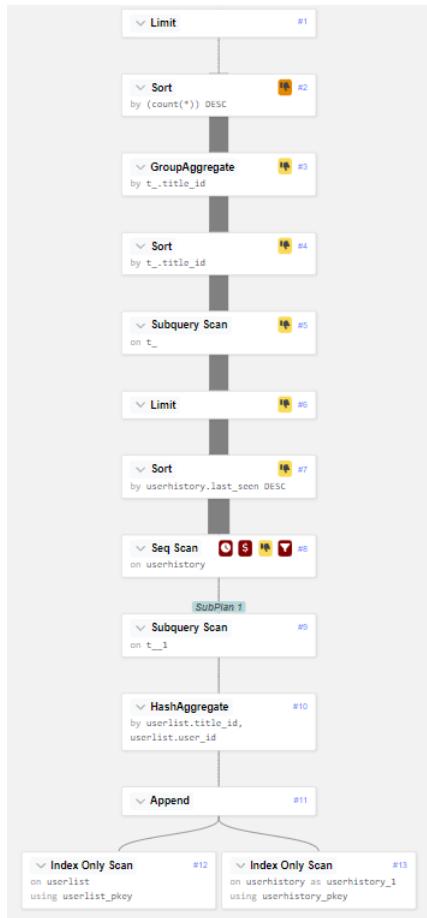


Figura 12: Plano de execução da query *getTitleFromPopular*

Sendo que a coluna *last_seen* é filtrada com base num intervalo foi criado o índice btree (*last_seen*) que melhorou o tempo de execução de 3623.778 ms para 291.904 ms. Além de ser filtrado pelo *last_seen*, também é feita uma filtragem com base no *title_id*. Existem mais valores distintos em *last_seen* do que em *title_id*, logo o índice composto criado foi btree (*last_seen*, *title_id*). Com esta última optimização o tempo de execução foi de 15.678 ms.

4.1.2 getWatchListInformation

A operação *getWatchListInformation* serve para obter informações sobre os títulos presentes na lista "para ver" de um utilizador.

Assim, esta operação começa por obter os identificadores dos títulos presentes na lista "para ver" (*getUserWatchList*).

Percorrendo essa lista, para cada título, extrai a informação do mesmo, como o nome, o tipo, a duração e o ano de lançamento (*getTitleInfo*). De seguida, obtém a ficha técnica (*getTitleMainCastAndCrew*).

Por fim, sabendo em que região habita o utilizador (*getUserRegion*), obtém o nome do título utilizado nessa região (*getTitleNameInRegion*).

getUserWatchList



O plano de execução da figura 13 foi gerado com o parâmetro `user_id = '100000'`. Ao observar o plano podemos concluir que, para obter a lista dos títulos na lista de um utilizador ordenadas pela data de criação, é feito um *Sort* imediatamente antes do *Index Scan* que filtra os resultados para o utilizador em causa.

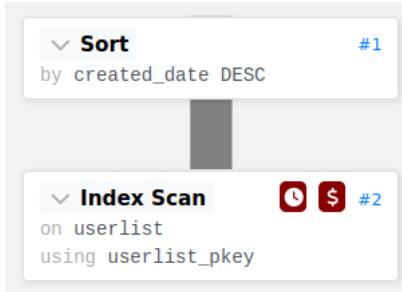


Figura 13: Plano de execução da *query getuserwatchlist*

Esta ordenação pela data de criação torna-se desnecessária se for criado um índice composto btree(`user_id, created_date`). Com a criação deste índice evitou-se o custo associado ao *Sort* e o tempo de execução passou de 1.888 ms para 0.416 ms.

getTitleInfo

Pela análise do plano de execução gerado, concluímos que apenas era usado um *Index Scan* sobre o título, utilizando o índice btree(`id`). Como não há influência de mais nenhum atributo, a única hipótese de melhoria seria a utilização de um índice hash, uma vez que se trata de uma pesquisa para um valor apenas. Embora o novo índice tenha sido utilizado, esta melhoria não foi consideravelmente impactante.

getTitleMainCastAndCrew

O plano de execução da figura 14 foi gerado com o parâmetro `p.title_id = 'tt0041024'`. Com base no plano é notável que a operação mais demorada é o *Index Scan* na tabela `name`.

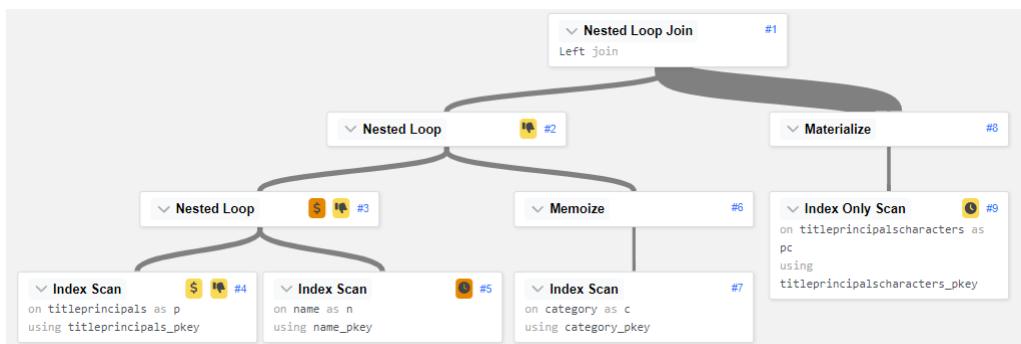


Figura 14: Plano de execução da *query gettitlemaincastandcrew*

A tabela `name` utiliza o índice btree(`name_id`), que é criado automaticamente dado que é a chave primária. Como os resultados deste índice não apresentam os melhores resultados, foi criado um índice utilizando *hash* e os resultados foram significativamente melhores, isto é, o tempo de execução passou de 52.761 ms para 29.030 ms.



getUserRegion

Pela análise do plano de execução, inicialmente, era usado um *Index Scan* com recurso a um índice btree(id). Tratando-se de uma pesquisa específica por um id, decidimos criar um índice hash(id). Obtivemos melhorias no tempo de execução, mas não muito significativas.

getTitleNameInRegion

Para a análise esta *query* começámos por analisar os valores distintos de *title_id* (cerca de 7 milhões) e *region* (cerca de 250) na tabela *tittleAkas*. Concluímos que não seria vantajoso criar um índice composto que associa-se esses dois atributos, uma vez que o número de regiões diferentes por identificador de título é muito pequeno. Logo, o *overhead* criado pelo índice não compensa.

4.1.3 viewTitle

Esta operação adiciona informação de visualização de um título ao “histórico” de um utilizador, como a data e a duração. Caso um utilizador veja um título na sua totalidade, este é removido da lista “para ver”.

Para isso, começa por executar a *query addTitleToHistory* que adiciona o título ao “histórico” do utilizador tendo em conta até onde foi visto.

E, de seguida, de acordo com a duração da visualização, verifica se o utilizador viu o título até ao fim. Se tal se confirmar é executada a *query removeTitleFromWatchList* que remove o título da lista “para ver” do utilizador.

addTitleToHistory

Geramos o seguinte plano de execução utilizando como parâmetros *user_id = '100000'* e *least = 20*:

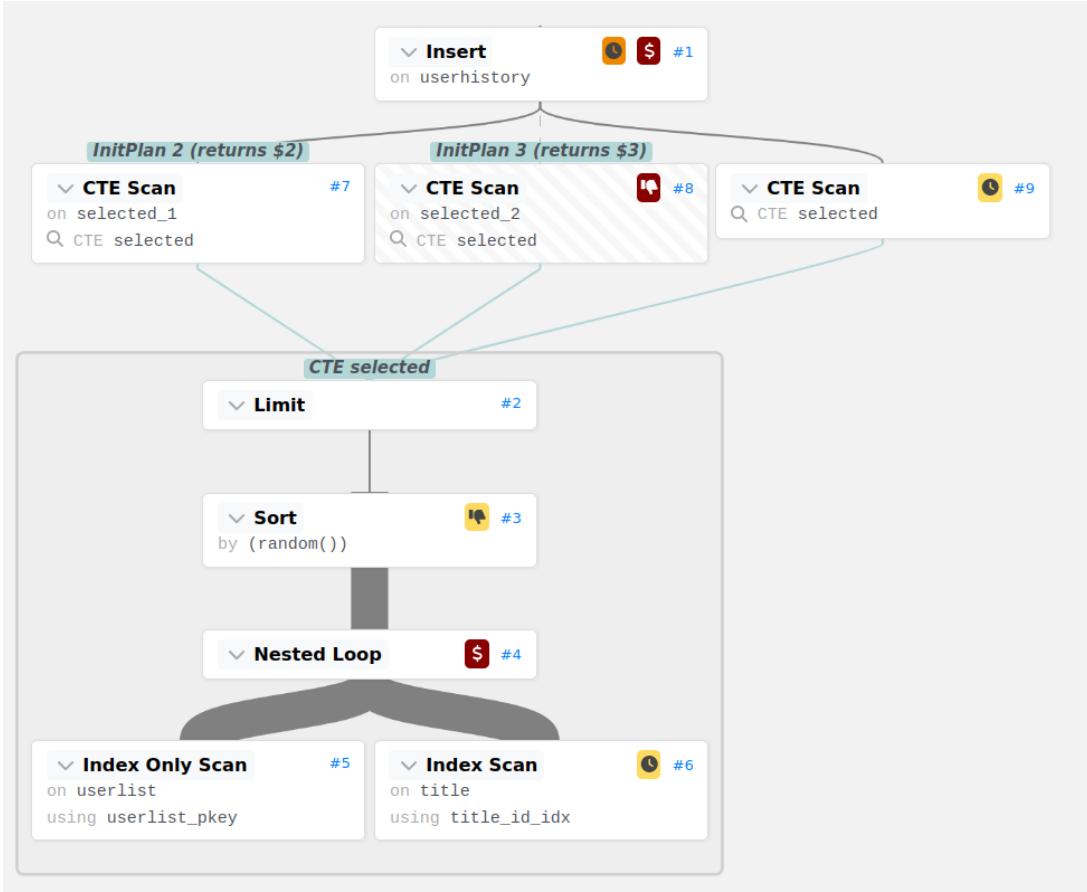


Figura 15: Plano de execução da query *addTitleToHistory*

Como é possível verificar, já é utilizado um índice anteriormente criado no campo id da tabela *title*. Para além disso, detetamos um custo elevado associado à pesquisa na tabela *userlist* e, por isso, decidimos então criar um índice utilizando *hash* que melhorou o tempo de 7.480 ms para 5.741 ms.

removeTitleFromWatchList

Para percebermos o tempo que esta query demoraria, executamos um *select* de um entrada onde *user_id='29120'* e *title_id='tt0505891'*, que recorria ao índice associado à chave primária composta por (*user_id*, *title_id*). A execução desta query demorava 1.781 ms.

No entanto, ao calcular o número de valores distintos de *user_id* e *title_id*, percebemos que existiam significativamente mais valores diferentes de *title_id*:

title_id_distinct_count	user_id_distinct_count
3 725 553	100 000

Tendo em conta este resultado, concluímos que seria mais vantajoso ter um índice composto pela ordem inversa: (*title_id*, *user_id*). Recorrendo a este índice obtivemos um tempo de execução 0.126 ms.



4.1.4 rateTitle

Este operação serve para adicionar ao histórico de um utilizador uma classificação relativa a um título que o mesmo visualizou e devolver a classificação global do título. Para tal começa por utilizar a *query rateTitleFromHistory* para atribuir a classificação do utilizador ao filme (substituindo possíveis avaliações que o cliente já tivesse feito a este mesmo título). De seguida, de forma a devolver a classificação global utiliza a *query getTitleRating*.

rateTitleFromHistory

Começámos por obter o plano de execução utilizando como parâmetros `user_id = 100000` e `rating = 5`, como é possível verificar observando esse plano de execução, é utilizado um índice que já composto em `(title_id, user_id)` já existente:

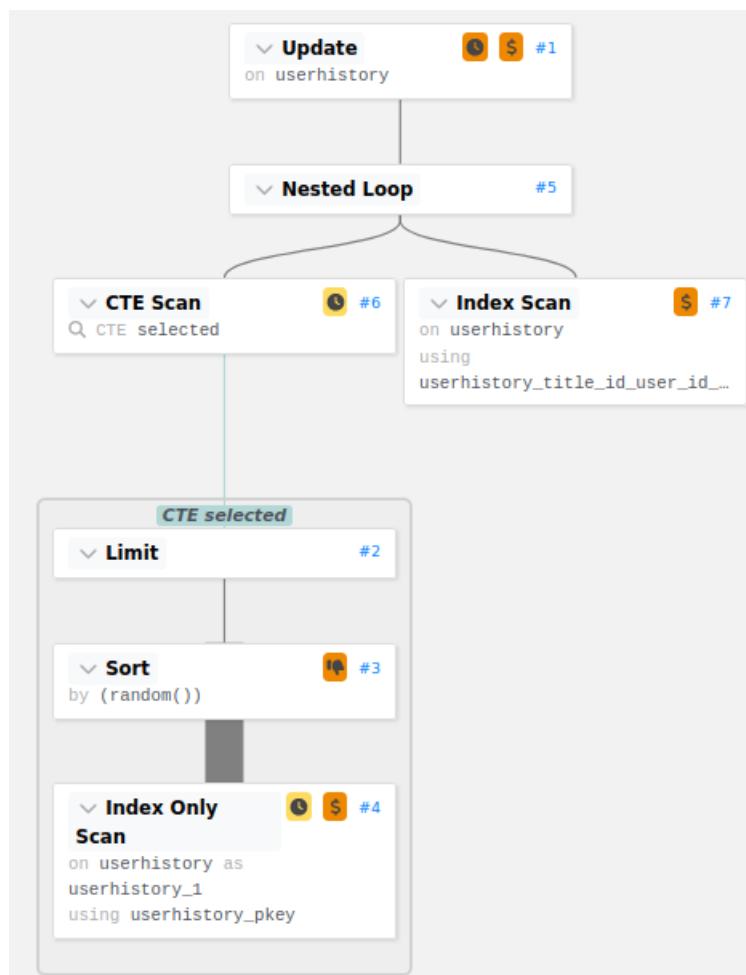


Figura 16: Plano de execução da *query rateTitleFromHistory*

Deste modo, o tempo de resposta era de 3.666 ms

getTitleRating



Para esta *query* não foram considerados novos índices, uma vez que a pesquisa dependia apenas do identificador do título. O índice já criado btree(user_id,title_id) já cumpre os requisitos para uma pesquisa otimizada.

4.1.5 searchTitles

Esta operação tal como o nome indica procura um filme pelo nome utilizando, para isso, a *query* *searchTitleByName*.

searchTitleByName

Esta interrogação utiliza um índice anteriormente criado no start_year baseado numa *btree* e demora 4109.535 ms usando como parâmetro 'north'.

4.2 Melhorias obtidas

Antes de fazer qualquer otimização, como criação de índices e vistas o resultado do *Benchmark* com um *warm-up* de 30 segundos, 300 segundos de tempo de execução e 1 cliente:

- **AddNewTitleToList** = 0.236s
- **GetWatchListInformation** = 0.231s
- **ViewTitle** = 0.015s
- **RateTitle** = 0.279s
- **SearchTitles** = 0.137s
- **throughput** = 7.41txn/s
- **response time** = 0.135s
- **abort rate** = 0%

Após as otimizações referidas anteriormente, o mesmo teste obteve os seguintes resultados:

- **AddNewTitleToList** = 0.018s
- **GetWatchListInformation** = 0.211s
- **ViewTitle** = 0.011s
- **RateTitle** = 0.007s
- **SearchTitles** = 0.113s
- **throughput** = 13.27txn/s
- **response time** = 0.075s
- **abort rate** = 0%

De uma forma geral, os tempo de execução de cada transação, bem como o tempo de resposta diminuíram e o *throughput* aumentou ligeiramente.

4.3 Parâmetros de configuração

Os parâmetros de configuração apresentam um papel crucial no desempenho da carga transacional. Dessa forma esta secção mostra qual foi o processo efetuado para o *tunning* dos parâmetros de configuração do *PostgreSQL*.

De maneira a testar diferentes parâmetros e fazer *tunning* destes foram feitos vários *bash scripts* que automatizam o processo de testagem e apresentam resultados para as diferentes tentativas.

Tendo como objetivo na parte transacional alcançar o maior *throughput* possível, em primeiro lugar foi descoberto qual o número de clientes no *Benchmark* que atinge maior *throughput*. Com base nos resultados apresentados na figura 17, o máximo *throughput* alcançado foi para 16 clientes.

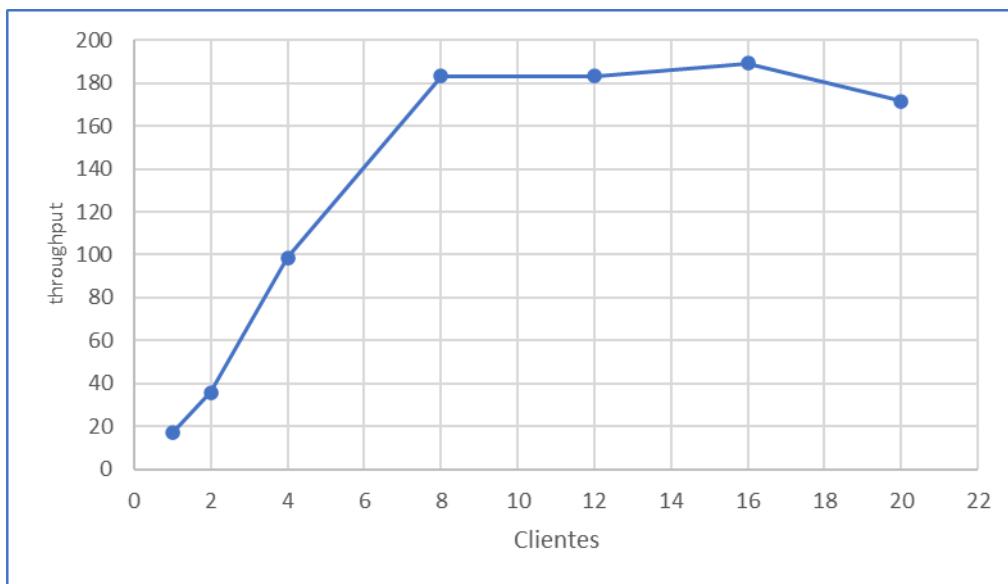


Figura 17: *Throughput* por cliente

4.3.1 *shared_buffers* e *work_mem*

Os primeiros parâmetros a fazer *tunning* foram o *shared_buffers* e o *work_mem*. Tal como referido anteriormente, foram criados *bash scripts* que permitiram obter os resultados para diferentes valores. Todos os *scripts* utilizados encontram-se nos anexos(secção 6).

Para o parâmetros *shared_buffers* teve-se em consideração a documentação do *PostgreSQL* que refere que um bom valor de começo é 25% da memória do sistema e que esta não deve ultrapassar os 40%. Sendo que a memória da VM utilizada é 16GB, 25% corresponde a 4GB e 40% a 6,4GB, por essa razão foi testado o intervalo entre 1 GB e 6GB.

A figura 18 mostra o *throughput* obtido para os diferentes valores de *shared_buffers*, verificando-se uma clara melhoria entre as 2GB e os 3GB, tendo depois estabilizado daí para frente, atingindo o máximo com 4GB.

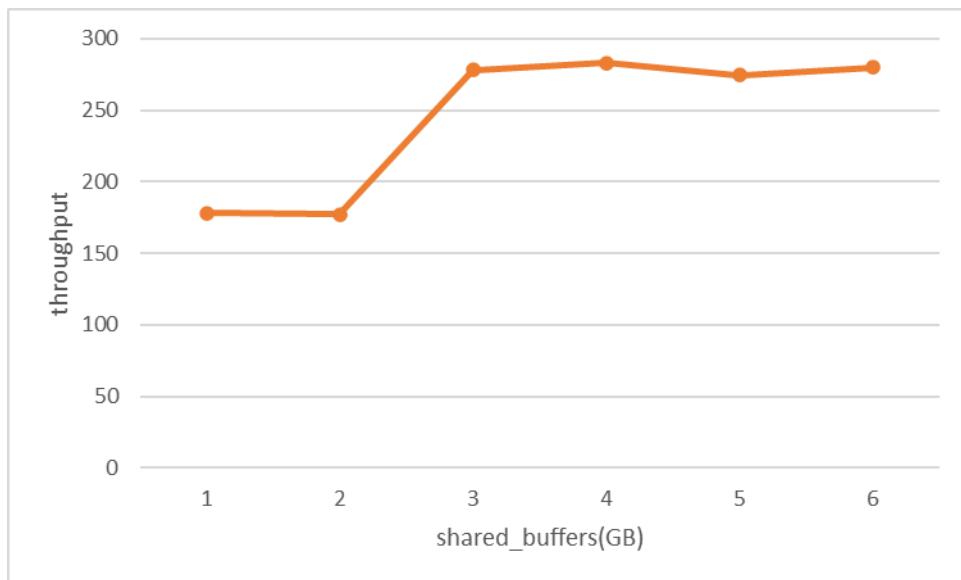


Figura 18: *Throughput* por *shared_buffers*

A figura 19 demonstra a variação do tempo de execução das *queries* analíticas com base no *shared_buffers*. As *queries* 1 e 2 não apresentam grande variação com os diferentes valores, porém a *query* 3 apresenta melhores resultados com maiores valores de *shared_buffers*.

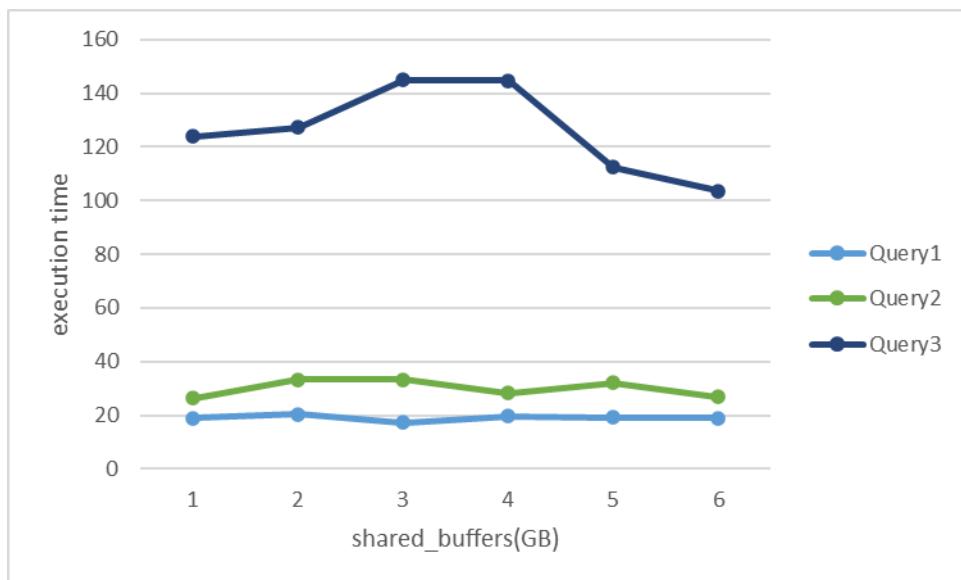


Figura 19: Tempo de execução por *shared_buffers* para cada *query*

Como o objetivo é maximizar o *throughput* da parte transacional, o valor escolhido para *shared_buffers* foi 4GB, dado que este valor corresponde ao máximo de *throughput* obtido.

Fixado o valor do *shared_buffers* foi utilizada a mesma estratégia para definir o valor do *work_mem*. Para definir um valor inicial para este parâmetro foi necessário alterar algumas configurações do *PostgreSQL*, mais concretamente ativar o *trace_sort*, alterar o *client_min_messages* para *log* e também alterar o *log_temp_files* para 64MB. Estas alterações foram feitas com o intuito de verificar se alguma *query* transacional estava a escrever para ficheiros temporários, dado

que o tamanho desses ficheiros é um bom ponto de partida para o *work_mem*.

Analizando o plano de execução das *queries* mais "pesadas" do *Benchmark*, verificou-se que em nenhuma delas houve a necessidade de criar ficheiros temporários. Por essa razão não foi considerado nenhum ponto de partida para o parâmetro *work_mem* e apenas foram testados diferentes valores de maneira a analisar o seu impacto.

O facto de não terem sido criados ficheiros temporários antecede que o *work_mem* não terá um impacto muito grande na parte transacional, como é visível na figura 20 em que a diferença entre o ponto mínimo e máximo é de cerca de 10 unidades.

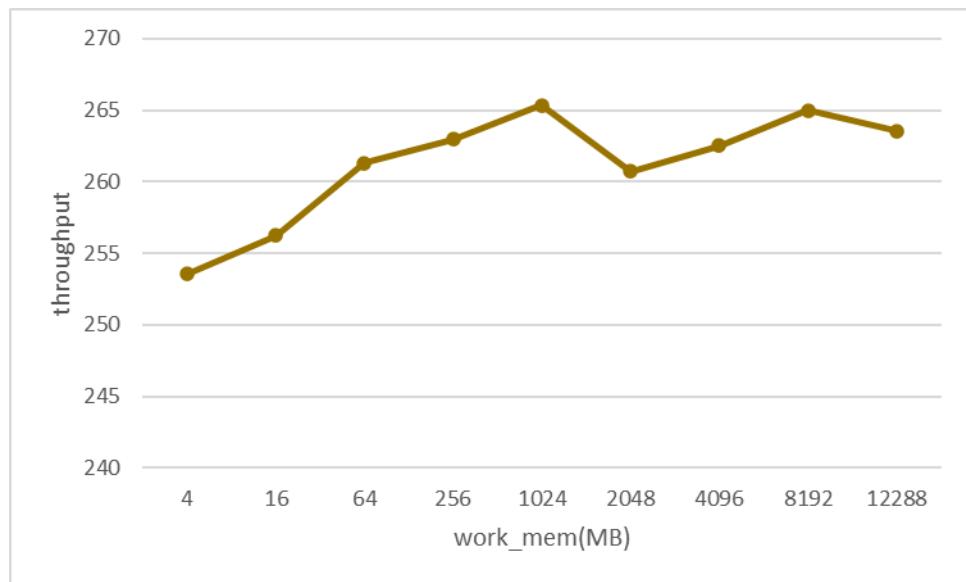


Figura 20: *Throughput* por *work_mem*

Ao analisar o impacto do *work_mem* no tempo de execução das *queries* (ver figura 21) verifica-se que o maior impacto é na *query 3*, sendo que nas outras 2 não existe grande variações.

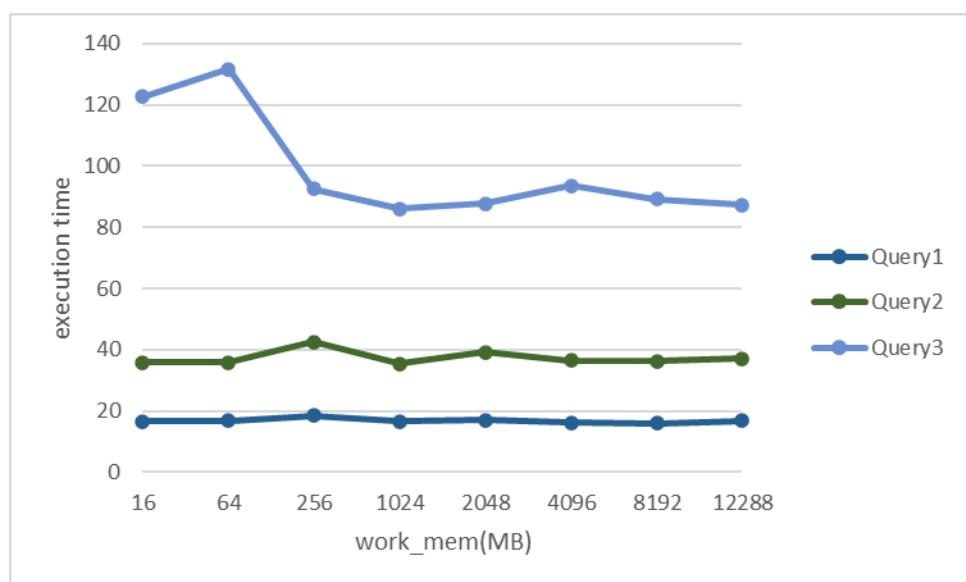


Figura 21: Tempo de execução por *work_mem* para cada *query*



Com base nestes dois gráficos, o melhor valor para o *work_mem* é 1024MB, uma vez que representa o máximo de *throughput* atingido e por outro lado o mínimo, ou valores perto do mínimo, para os tempos de execução de cada *query*.

4.3.2 Write Ahead Log

Os parâmetros de configuração do *Write Ahead Log*(WAL) são importantes para garantir a durabilidade dos dados na base de dados. Procurou-se otimizar apenas os parâmetros que permitiam melhorar o desempenho da carga transacional sem colocar em perigo os dados, isto é, evitando *data corruption*.

O *checkpoint_timeout* define o tempo máximo entre *checkouts*, por defeito é de 300 segundos(5 minutos), sendo que o tempo utilizado para correr o *Benchmark* é 500 segundos(8 minutos). Este valor foi aumentado para 10 minutos com o intuito de diminuir o número de *checkouts*, que poderá ter resultado num impacto positivo no desempenho e um impacto negativo no tempo necessário para recuperar de um *crash*.

Além de aumentar o *checkpoint_timeout* também foram testados diferentes valores de *max_wal_size* e *min_wal_size* com o propósito de diminuir o número de *checkouts*.

Por defeito o *max_wal_size* é 1GB e o *min_wal_size* 80MB, os valores testados bem como o *throughput* obtido está representado na seguinte tabela:

<i>max_wal_size</i> (GB)	<i>textit{min_wal_size}</i> (GB)	<i>throughput</i>
4	1	267.75
8	2	268.994
16	4	264.804

Os resultados obtidos não são esclarecedores, visto que o *throughput* obtido é sensivelmente o mesmo para cada tentativa. Uma das possíveis explicações para isto poderá ser que o tempo utilizado para correr o *Benchmark*(8 minutos) não seja suficiente para encher o WAL.

Foram considerados ainda outros parâmetros, como o *wal_buffers* que representa a quantidade de memória partilhada utilizada pelo WAL que ainda não foi escrita para disco. Segundo a documentação do *PostgreSQL* o tamanho utilizado neste parâmetro pode ter impacto no desempenho e o melhor valor para o *tunning* é -1 que representa cerca de $\frac{1}{32}$ (cerca de 3%) do *shared_buffers*.

Como o valor do *shared_buffers* é 4GB, 3% corresponde a 128MB, por essa razão esse foi o valor base e foram testados valores de *wal_buffers* maiores e menores que 128. A figura 22 mostra o *throughput* para diferentes valores de *wal_buffers*.

A partir dos resultados obtidos conclui-se que o *tunning* do *wal_buffers* não permitiu obter resultados melhores, visto que os *throughput* obtidos são todos semelhantes e abaixo de valores obtidos anteriormente.

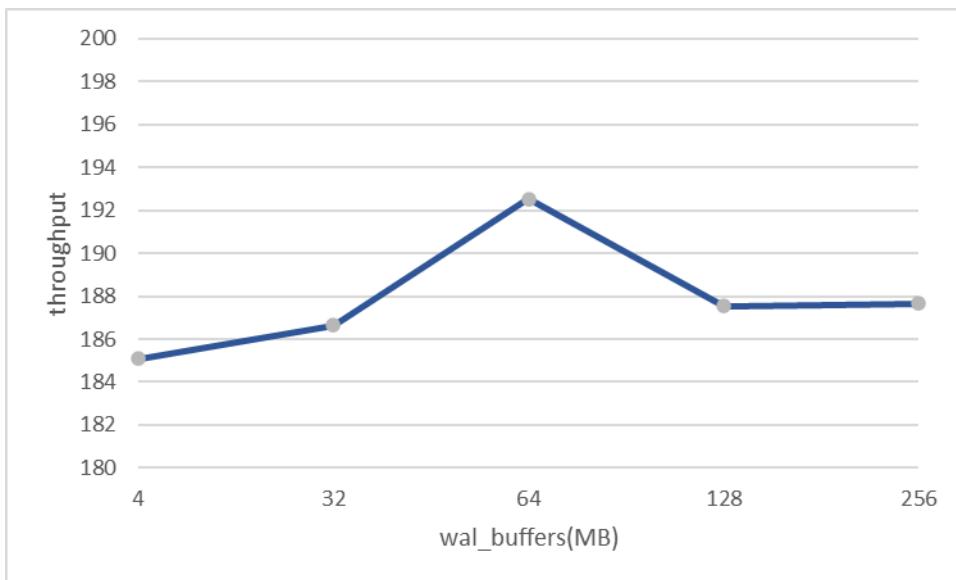


Figura 22: *Throughput* por *wal_buffers*

Da mesma forma que o *wal_buffers* o parâmetro *wal_writer_delay* foi testado com diferentes valores. Os resultados são apresentados na figura 23. Mais uma vez os resultados obtidos mostram que o *tunning* deste parâmetro não permitiu obter resultados melhores.

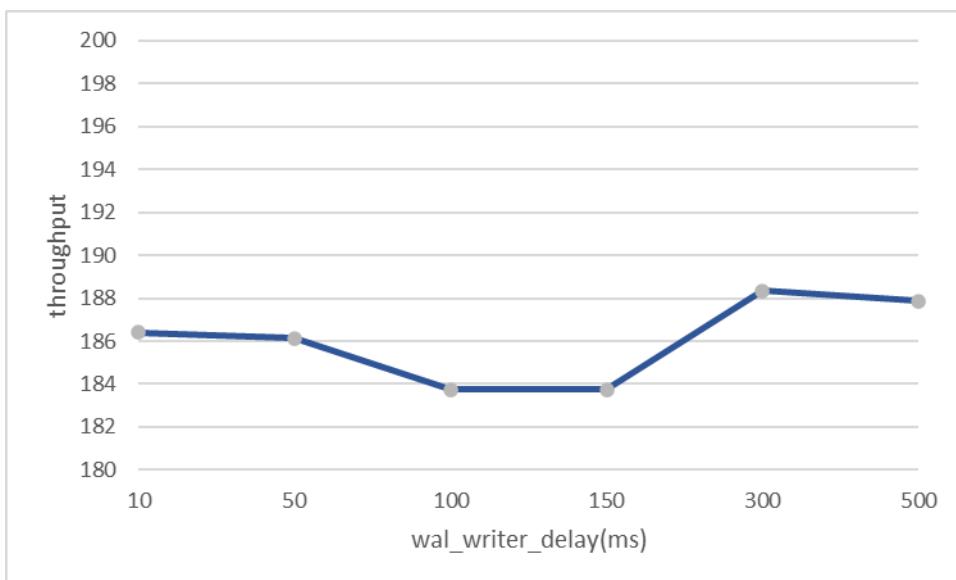


Figura 23: *Throughput* por *wal_writer_delay*

4.3.3 Lock Management

Quanto às configurações do *Lock Management*, segundo a documentação do *PostgreSQL*:

- **deadlock_timeout**: O valor por defeito, 1 segundo, é o menor valor que se pode ter na prática e para servidores com uma carga elevada é ideal utilizar valores superiores.
- **max_locks_per_transaction**: O valor por defeito, 64, e que só deve ser alterado caso as transações utilizem muitas tabelas diferentes, que não é o caso.

Por essa razão só foi feito o *tunning* do parâmetro *deadlock_timeout*. Tal como em *tunnings* anteriores, os resultados obtidos (ver figura 24) não permitiram obter melhorias significativas.

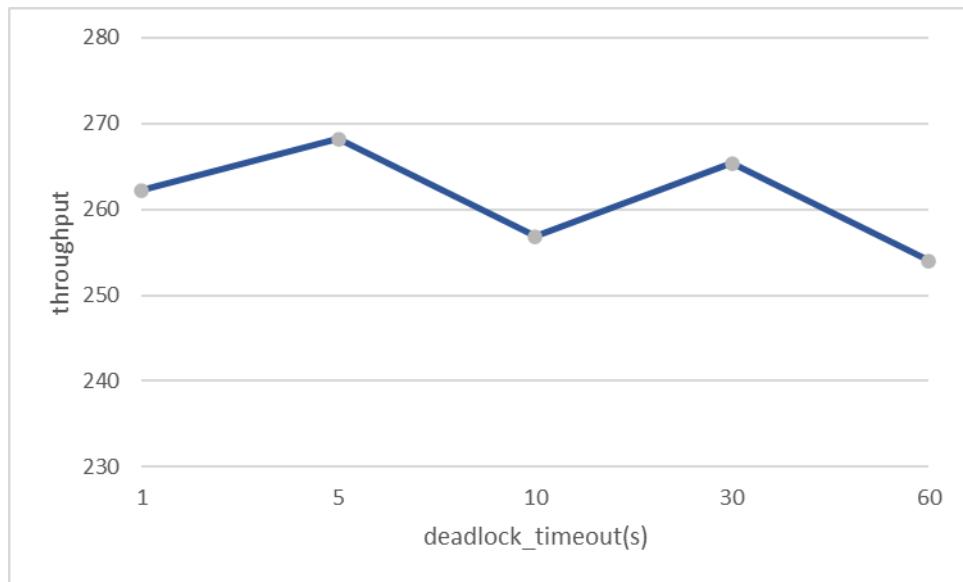


Figura 24: *Throughput* por *deadlock_timeout*

4.4 Melhorias obtidas

Das diferentes configurações testadas, os parâmetros que acabaram por ser alterados e que trouxeram melhorias ao *throughput* foram o *shared_buffers* e *work_mem*. Com um *shared_buffers* de 4GB e um *work_mem* de 1024MB o resultado *Benchmark* com 16 clientes, o máximo de *throughput* atingido foi de 283.11.



5 Conclusão

No que diz respeito à otimização das interrogações analíticas, consideramos que podíamos ter obtido melhores resultados. O grupo entendeu que não ficou claro a que tipo de interrogações, para além das apresentadas, a base de dados poderia ser sujeita. Tendo isto em conta, tornou-se complicado perceber se as decisões tomadas foram demasiado específicas neste contexto ou se teriam uma influência positiva noutras interrogações possíveis.

Já em relação à otimização das interrogações analíticas no Spark, foram observadas melhorias nos resultados das diferentes interrogações. Os resultados obtidos são considerados bons, porém, o grupo reconhece que poderíamos ter explorado mais as técnicas mencionadas anteriormente, o que teria permitido alcançar resultados ainda melhores.

Na configuração dos parâmetros da parte transacional, o grupo considera que o *tunning* dos parâmetros *shared_buffers* e *work_mem* efetivamente trouxeram resultados melhores no que toca ao *throughput*, porém não conseguiu obter qualquer melhoria com outros parâmetros de configuração (*Write Ahead Log* e *Locking Management*). Seria ideal, possivelmente considerar outros parâmetros que não foram tidos em conta e desenvolver mais *scripts* que permitissem avaliar o desempenho destes.



6 Anexos

6.1 Script *start_spark*

```
1 #!/usr/bin/env sh
2 sudo apt-get install ca-certificates curl gnupg
3 sudo install -m 0755 -d /etc/apt/keyrings
4 curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor
5   ↳ -o /etc/apt/keyrings/docker.gpg
6 sudo chmod a+r /etc/apt/keyrings/docker.gpg
7 echo \
8   "deb [arch=$(dpkg --print-architecture)]"
9     ↳ signed-by=/etc/apt/keyrings/docker.gpg
10    ↳ https://download.docker.com/linux/ubuntu \
11      "$(. /etc/os-release && echo \"$VERSION_CODENAME\")" stable" | \
12        sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
13 sudo apt-get update
14 sudo apt-get install docker-ce docker-ce-cli containerd.io
15   ↳ docker-buildx-plugin docker-compose-plugin
16 sudo apt install docker-compose
17 sudo gpasswd -a $USER docker
18 newgrp docker
19 docker-compose -p spark up -d --scale spark-worker=3
20 docker exec spark_spark_1 start-history-server.sh
21 docker exec spark_spark_1 python3 main.py
```

6.2 Script *shared_buffers*

```
1#!/bin/bash
2shared_buffers=(1 2 3 4 5 6 7)
3for s in "${shared_buffers[@]}"; do
4  echo "-----Running $s shared_buffers"
5  ↳ -----
6  export PGPASSWORD=postgres
7  psql -U postgres -h localhost -c "ALTER SYSTEM SET shared_buffers TO '$s GB';"
8  psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
9  sudo systemctl restart postgresql
10 java -jar target/transactional-1.0-SNAPSHOT.jar -d
11   ↳ jdbc:postgresql://localhost:5432/imdb -U postgres -P postgres -W 30 -R 500
12   ↳ -c 16
13 start=$(date +%s.%N)
14 psql -U postgres -h localhost -d imdb -a -f ../analytical/1.sql
15 end=$(date +%s.%N)
16 runtime=$(echo "$end - $start" | bc)
17 echo "1.sql: $runtime"
18 start=$(date +%s.%N)
```



```
19 psql -U postgres -h localhost -d imdb -a -f ./analytical/2.sql
20 end=$(date +%s.%N)
21 runtime=$(echo "$end - $start" | bc)
22 echo "2.sql: $runtime"
23
24 start=$(date +%s.%N)
25 psql -U postgres -h localhost -d imdb -a -f ./analytical/3.sql
26 end=$(date +%s.%N)
27 runtime=$(echo "$end - $start" | bc)
28 echo "3.sql: $runtime"
29 done
```

6.3 Script *work_mem*

```
1 #!/bin/bash
2 export PGPASSWORD=postgres
3 psql -U postgres -h localhost -c "ALTER SYSTEM SET shared_buffers TO '4 GB';"
4 psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
5 work_mem=(4 16 64 256 1024 2048 4096 8192 12288)
6 for s in "${work_mem[@]}"; do
7     echo "-----Running $s work_mem -----"
8     export PGPASSWORD=postgres
9     psql -U postgres -h localhost -c "ALTER SYSTEM SET work_mem TO '$s MB';"
10    psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
11    sudo systemctl restart postgresql
12
13    java -jar target/transactional-1.0-SNAPSHOT.jar -d
14        ↳ jdbc:postgresql://localhost:5432/imdb -U postgres -P postgres -W 30 -R 500
15        ↳ -c 16
16
17    start=$(date +%s.%N)
18    psql -U postgres -h localhost -d imdb -a -f ./analytical/1.sql
19    end=$(date +%s.%N)
20    runtime=$(echo "$end - $start" | bc)
21    echo "Runtime was $runtime"
22
23    start=$(date +%s.%N)
24    psql -U postgres -h localhost -d imdb -a -f ./analytical/2.sql
25    end=$(date +%s.%N)
26    runtime=$(echo "$end - $start" | bc)
27    echo "2.sql: $runtime"
28
29    start=$(date +%s.%N)
30    psql -U postgres -h localhost -d imdb -a -f ./analytical/3.sql
31    end=$(date +%s.%N)
32    runtime=$(echo "$end - $start" | bc)
33    echo "3.sql: $runtime"
34
35 done
```



6.4 Script para *max_wal_size* e *min_wal_size*

```
1 #!/bin/bash
2 export PGPASSWORD=postgres
3 psql -U postgres -h localhost -c "ALTER SYSTEM SET shared_buffers TO '4 GB';"
4 psql -U postgres -h localhost -c "ALTER SYSTEM SET work_mem TO '1024 MB';"
5 psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
6
7 psql -U postgres -h localhost -c "ALTER SYSTEM SET checkpoint_timeout TO
8   ↵ '10min';"
9 psql -U postgres -h localhost -c "ALTER SYSTEM SET max_wal_size TO '4GB';"
10 psql -U postgres -h localhost -c "ALTER SYSTEM SET min_wal_size TO '1GB';"
11 psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
12
13 echo "Running with max_wal_size = 4GB, min_wal_size = 1GB"
14 java -jar target/transactional-1.0-SNAPSHOT.jar -d
15   ↵ jdbc:postgresql://localhost:5432/imdb -U postgres -P postgres -W 30 -R 500
16   ↵ -c 16
17
18 sudo systemctl restart postgresql
19 echo "Running with max_wal_size = 8GB, min_wal_size = 2GB"
20 psql -U postgres -h localhost -c "ALTER SYSTEM SET max_wal_size TO '8GB';"
21 psql -U postgres -h localhost -c "ALTER SYSTEM SET min_wal_size TO '2GB';"
22 psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
23
24 java -jar target/transactional-1.0-SNAPSHOT.jar -d
25   ↵ jdbc:postgresql://localhost:5432/imdb -U postgres -P postgres -W 30 -R 500
26   ↵ -c 16
27
28 sudo systemctl restart postgresql
29 echo "Running with max_wal_size = 16GB, min_wal_size = 4GB"
30 psql -U postgres -h localhost -c "ALTER SYSTEM SET max_wal_size TO '16GB';"
31 psql -U postgres -h localhost -c "ALTER SYSTEM SET min_wal_size TO '4GB';"
32 psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
33
34 java -jar target/transactional-1.0-SNAPSHOT.jar -d
35   ↵ jdbc:postgresql://localhost:5432/imdb -U postgres -P postgres -W 30 -R 500
36   ↵ -c 16
```

6.5 Script para *wal_buffers* e *wal_writer_delay*

```
1 #!/bin/bash
2 export PGPASSWORD=postgres
3 psql -U postgres -h localhost -c "ALTER SYSTEM SET shared_buffers TO '4 GB';"
4 psql -U postgres -h localhost -c "ALTER SYSTEM SET work_mem TO '1024 MB';"
5
6 psql -U postgres -h localhost -c "ALTER SYSTEM SET checkpoint_timeout TO
7   ↵ '10min';"
8 psql -U postgres -h localhost -c "ALTER SYSTEM SET max_wal_size TO '8GB';"
```



```
8 psql -U postgres -h localhost -c "ALTER SYSTEM SET min_wal_size TO '2GB';"
9 psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
10 sudo systemctl restart postgresql
11
12 echo "Running wal buffers = -1"
13 psql -U postgres -h localhost -c "ALTER SYSTEM SET wal_buffers TO '-1';"
14 psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
15 sudo systemctl restart postgresql
16
17 java -jar target/transactional-1.0-SNAPSHOT.jar -d
    ↳ jdbc:postgresql://localhost:5432/imdb -U postgres -P postgres -W 30 -R 500
    ↳ -c 16
18
19 wal_buffers=(4 32 64 128 256)
20 for w in "${wal_buffers[@]}"; do
21     echo "-----Running $w wal_buffers -----"
22     export PGPASSWORD=postgres
23     psql -U postgres -h localhost -c "ALTER SYSTEM SET wal_buffers TO '$w MB';"
24     psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
25     sudo systemctl restart postgresql
26
27     java -jar target/transactional-1.0-SNAPSHOT.jar -d
        ↳ jdbc:postgresql://localhost:5432/imdb -U postgres -P postgres -W 30 -R 500
        ↳ -c 16
28 done
29
30 psql -U postgres -h localhost -c "ALTER SYSTEM SET wal_buffers TO '-1';"
31 psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
32 sudo systemctl restart postgresql
33
34 wal_writer_delay=(10 50 100 300 500)
35 for w in "${wal_writer_delay[@]}"; do
36     echo "-----Running $w wal_writer_delay
    ↳ -----
37     export PGPASSWORD=postgres
38     psql -U postgres -h localhost -c "ALTER SYSTEM SET wal_writer_delay TO '$w
    ↳ ms';"
39     psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
40     sudo systemctl restart postgresql
41
42     java -jar target/transactional-1.0-SNAPSHOT.jar -d
        ↳ jdbc:postgresql://localhost:5432/imdb -U postgres -P postgres -W 30 -R 500
        ↳ -c 16
43
44 done
```



6.6 Script para *Lock Management*

```
1 #!/bin/bash
2 export PGPASSWORD=postgres
3 psql -U postgres -h localhost -c "ALTER SYSTEM SET shared_buffers TO '4 GB';"
4 psql -U postgres -h localhost -c "ALTER SYSTEM SET work_mem TO '1024 MB';"
5
6 psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
7
8 deadlock_timeout=(1 5 10 30 60)
9 for d in "${deadlock_timeout[@]}"; do
10   echo "-----Running $d deadlock_timeout"
11   ↪ -----
12   export PGPASSWORD=postgres
13   psql -U postgres -h localhost -c "ALTER SYSTEM SET deadlock_timeout TO '$d
14   ↪ s';"
15   psql -U postgres -h localhost -c "SELECT pg_reload_conf();"
16   sudo systemctl restart postgresql
17
18 done
```