ABD

Universidade do Minho
Escola de Engenharia
Mestrado em Engenharia Informática

Administração de Bases de Dados

Ano lectivo 2023/2024

Relatório do trabalho prático

Grupo 6

Ana Rita Moreira Vaz, PG53642

José Eduardo da Cunha Rocha, A97270

Miguel Silva Pinto, PG54105

Orlando José da Cunha Palmeira, PG54123

Pedro Miguel Castilho Martins, PG54146

Braga, 24 de Maio de 2024

Índice

1 Introdução **1**

1.1 Contexto e Desafios 1

1.2 Metodologia 1

2 Carga analítica no *PostgreSQL* **2**

2.1 Interrogação analítica 1 2

2.2 Interrogação analítica 2 8

2.3 Interrogação analítica 3 16

2.4 Interrogação analítica 4 24

3 Carga analítica no *Spark* **27**

3.1 Interrogação analítica 1 28

3.2 Interrogação analítica 2 32

3.3 Interrogação analítica 3 34

3.4 Interrogação analítica 4 37

3.5 Otimizações de configuração Spark 40

4 Carga transacional **41**

4.1 Otimização através de índices 42

4.1.1 GetQuestionInfo 42

4.1.2 GetPostPoints 42

4.1.3 Search 43

4.1.4 LatestQuestionByTag 43

4.1.5 Resultados obtidos 44

4.2 Otimização através de parâmetros 48

4.2.1 shared_buffers 48

4.2.2 work_mem 49

4.2.3 effective_cache_size 52

4.2.4 Paralelismo 53

4.2.5 Checkpoints e WAL 54

4.2.6 fsync 56

4.2.7 synchronous_commit 57

4.2.8 wal_compression 57

4.2.9 Conclusão parâmetros de configuração 58

5 Conclusão **59**

A Script de activação da VM **60**

B Script de desactivação da VM **60**

C Script de medição de tempo de *queries* **61**

D Resultados pormenorizados **68**

Índice de figuras

2.1 Plano de execução da query 1 base . . . 3
2.2 Plano de execução da query 1 com índices . . . 4
2.3 Plano de execução da query 1 reestruturada . . . 6
2.4 Plano do PostgreSQL para a versão base da *query* #2 . . . 9
2.5 Plano do PostgreSQL para a versão da *query* #2 com os índices . . . 10
2.6 Plano do PostgreSQL para a versão da *query* #2 com o índice do ano a ser utilizado . . . 11
2.7 Tabela *users* percorrida 17 vezes para encontrar a reputação máxima em cada ano . . . 12
2.8 Tabela *users* percorrida uma vez para encontrar a reputação máxima em cada ano . . . 13
2.9 *Merge join* versão original . . . 13
2.10 *Merge join* versão reestruturada . . . 13
2.11 *Sort* e *Merge join* versão original . . . 14
2.12 *Sort* e *Merge join* versão reestruturada . . . 14
2.13 Explain Analyse . . . 21
2.14 Resultados query original . . . 24
2.15 Resultados query com índice . . . 24
2.16 Resultados query com índice e materialized view . . . 24
2.17 Plano de execução da query 4 base . . . 25
3.1 Comparação da quantidade de dados lida sem e com particionamento pelo ano . . . 30
3.2 Tamanhos das duas tabelas da operação de *join* . . . 31
3.3 Comparação do scan inicial da Query 2 . . . 34
3.4 Comparação dos tempos de execução . . . 37
3.5 Comparação das estatísticas de *shuffle* . . . 39
4.1 Comparação entre o plano da query getQuestionInfo com e sem índice . . . 42
4.2 Comparação entre o plano da query getPostPoints com e sem índice . . . 43
4.3 Comparação entre o plano da query latestQuestionsByTag com e sem índice . . . 44
4.4 Throughput Variation with Shared Buffers . . . 48
4.5 Response Time Variation with Shared Buffers . . . 49
4.6 Impacto do work_mem nas diversas queries da componente transacional (gráfico em escala logarítmica) . . . 51
4.7 Execution Time for Different Cache Sizes . . . 52
4.8 Execution Time of Queries with Different Numbers of Workers . . . 53
4.9 Comparison of Response Time per Function . . . 55
4.10 Comparison of Response Times with and without `wal_compression` . . . 58

Índice de tabelas

2.1 Execution time of different arguments for Q1 8
2.2 Evolução do desempenho da query #2 com as optimizações aplicadas 16
2.3 Base Execution Times for Q4 25
2.4 Final Execution Times for Q4 27
3.1 Tempos de execução do Workload 1 base 28
3.2 Tempos sem a *broadcast hint* 31
3.3 Tempos com a *broadcast hint* 31
3.4 Tempos de execução do Workload 1 com partitionBy('year') 32
3.5 Resultados Base Q2 Spark 32
3.6 Resultados Finais Q2 Spark 34
3.7 Resultados Base Q3 Spark 35
3.8 Resultados Finais Q3 Spark 37
3.9 Resultados inicias da Q4 Spark 38
3.10 Resultados Finais Q4 Spark 40
3.11 Resultados (em segundos) para diferentes números de *workers*. 40
3.12 Resultados (em segundos) para diferentes opções de configuração 41
3.13 Tempos de execução finais em segundos 41
4.2 Performance Statistics 46
4.3 Estatísticas de Desempenho 47
4.4 Memory Information 48
4.5 Resultados do EXPLAIN ANALYZE 50
4.6 Performance Metrics for Different Configurations 53
4.7 Comparison of Results with and without `fsync=off` 56
4.8 Comparison of Results with and without `synchronous_commit=off` 57
D.1 Tempos de execução (ms) para as diferentes interrogações em várias configurações de work_mem 69

1 Introdução

Este relatório apresenta o trabalho prático realizado no âmbito da unidade curricular de Administração de Bases de Dados do curso de Engenharia Informática da Universidade do Minho. O objetivo principal foi a configuração, otimização e avaliação de um *benchmark* baseado num excerto do *dataset* do StackOverflow, abrangendo operações transacionais e analíticas.

1.1 Contexto e Desafios

O desafio envolveu a simulação de operações de *backend* do StackOverflow, tais como a criação de perguntas e respostas, votação, e obtenção de informações de posts e perfis de utilizadores. Adicionalmente, foram realizadas interrogações analíticas em dois ambientes distintos: diretamente na base de dados PostgreSQL e através de um *snapshot* no Apache Spark.

1.2 Metodologia

A metodologia adotada incluiu a otimização do desempenho tanto da carga transacional quanto das interrogações analíticas, considerando aspetos como redundância, paralelismo, configuração e código SQL/Java.

De modo a organizarmos o processo de evolução do trabalho prático decidimos criar um projeto partilhado por todos os membros do grupo. Neste projeto, foi inicialmente criada a máquina virtual na *Google Cloud* com especificações detalhadas no enunciado e posteriormente realizamos o provisionamento da máquina virtual com todos os requisitos para o trabalho prático. Para que não tenhamos que estar sempre a realizar este passo do provisionamento e para poupar recursos na *Google Cloud*, recorremos ao uso de *Machine Images*. Desta forma, sempre que criarmos uma VM, apenas temos de criar a instância a partir da *Machine Image* base e todos os recursos provisionados inicialmente já se encontram presentes na VM criada.

Para que a criação da VM seja ainda mais rápida decidimos implementar scripts que utilizam a gcloud CLI para automatizar o processo de levantamento, gravação e destruição da VM. Uma script serve para automatizar o processo de criação da VM a partir da *machine image* previamente criada. A outra desliga a VM e atualiza a *machine image* ou, no caso de não querermos atualizar a *machine image* e apenas desligar e destruir a VM, pode-se especificar isso com a flag `-d`. As scripts encontram-se nos anexos A e B, respetivamente.

2 Carga analítica no *PostgreSQL*

Para a análise e melhoria da carga analítica em PostgreSQL, procuramos identificar os gargalos de desempenho que afetam a eficiência das interrogações. As melhorias implementadas envolvem reestruturações do algoritmo das *queries*, criação de índices e, nalguns casos, a criação de vistas materializadas.

Para ajudar na identificação de limitações de desempenho e avaliar as melhorias implementadas, utilizamos a funcionalidade *EXPLAIN ANALYZE* do PostgreSQL, que faz uma descrição do plano gerado para a execução da *query*. Em conjunto com esta funcionalidade, também recorremos ao site explain.dalibo que permite visualizar e compreender os planos EXPLAIN do PostgreSQL de uma forma mais acessível e gráfica.

Para a avaliação do desempenho das interrogações analíticas criamos uma script Python *args_script.py*, presente no anexo C, em que passamos como argumento o ficheiro SQL da query a avaliar, para apresentar uma tabela com os tempos de execução média para diferentes argumentos possíveis da query, o que permite ter uma visão mais abrangente e informada das alterações de forma rápida, sem estar manualmente a alterar os argumentos da interrogação.

2.1 Interrogação analítica 1

```
SELECT id, displayname,
    count(DISTINCT q_id) + count(DISTINCT a_id) + count(DISTINCT c_id) total
FROM (
    SELECT u.*, q.id q_id, a.id a_id, c.id c_id
    FROM users u
    LEFT JOIN (
        SELECT *
        FROM questions
        WHERE creationdate BETWEEN now() - interval '6 months' AND now()
    ) q ON q.owneruserid = u.id
    LEFT JOIN (
        SELECT *
        FROM answers
        WHERE creationdate BETWEEN now() - interval '6 months' AND now()
    ) a ON a.owneruserid = u.id
    LEFT JOIN (
        SELECT *
        FROM comments
        WHERE creationdate BETWEEN now() - interval '6 months' AND now()
```

```
20      ) c ON c.userid = u.id
21  ) t
22  GROUP BY id, displayname
23  ORDER BY total DESC
24  LIMIT 100;
```

Esta interrogação realiza uma contagem total de atividades de utilizadores entre a atualidade e um limite inferior de tempo parametrizável. No caso padrão acima demonstrado, o período considerado é entre a data atual e seis meses atrás. Na contagem total são consideradas perguntas feitas, respostas dadas e comentários feitos, por cada utilizador. Após a contagem, os utilizadores são ordenados por ordem decrescente consoante a sua atividade, apresentando no final o top 100.

Após a interpretação da query, foi feita uma análise dos planos de execução gerados pelo PostgreSQL, recorrendo à funcionalidade *EXPLAIN ANALYZE* e ao site explain.dalibo. A query base apresentava um tempo médio de execução de **5.922 segundos** e o diagrama resultante do plano de execução da query 1 base foi o seguinte:

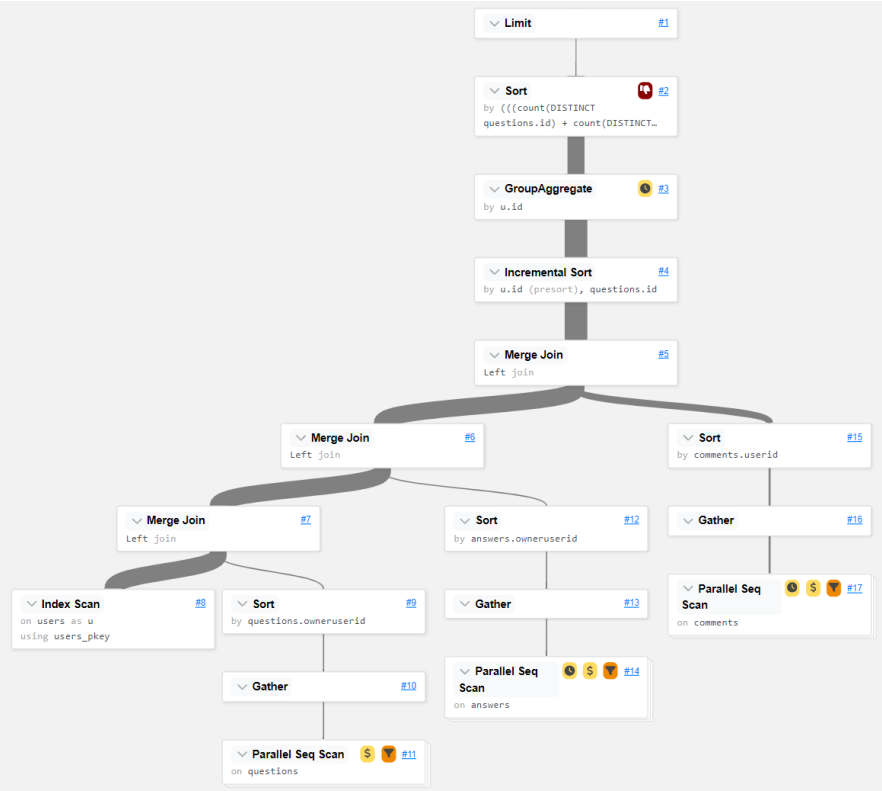


Figura 2.1: Plano de execução da query 1 base

Ao analisar o plano de execução podemos verificar que as partes que mais tempo demoram são os

"*Parallel Seq Scan*" que realizam o filtro de linhas das tabelas "*answers*", "*questions*" e "*comments*" pela coluna "*creationdate*" relativamente ao período temporal definido como argumento. Logo, surgiu a ideia de criar índices nas colunas "*creationdate*" das 3 tabelas, dado que a criação de índices em colunas que são utilizadas para operações de filtragem pode melhorar o desempenho da query.

```
1 CREATE INDEX idx_comments_creationdate ON comments (creationdate);
2 CREATE INDEX idx_questions_creationdate ON questions (creationdate);
3 CREATE INDEX idx_answers_creationdate ON answers (creationdate);
```

Com a criação destes índices voltamos a avaliar o desempenho da query e notou-se uma melhoria no tempo de execução passando para **3.67 segundos**, apresentando uma melhoria relativa de quase metade do tempo original. Com estas melhorias o diagrama do plano de execução passou a ter a seguinte estrutura.

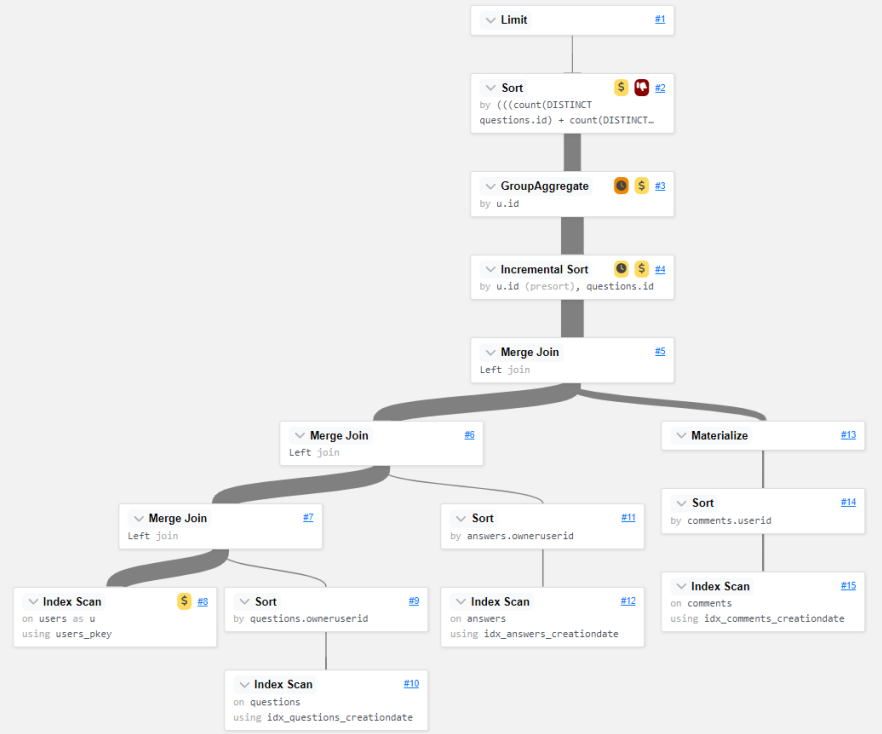


Figura 2.2: Plano de execução da query 1 com índices

Analisando o novo plano, passamos a ver que são utilizados "*Index Scans*" invés dos "*Parallel Seq Scan*", resultando numa filtragem muito mais eficiente. Agora a principal parte do tempo de execução está a ser dedicado ao processo de agregação ("*GroupBy*") de todos dos registos de atividade que cada utilizador.

Para tentar resolver este gargalo no agrupamento dos registos, decidimos seguir uma estratégia

de reestruturação da estratégia da interrogação. Esta reestruturação tem como objetivo, mitigar o impacto do tempo gasto no agrupamento dos resultados por cada utilizador através da redução da dimensão dos dados que chegam ao operador de agrupamento (*GroupBy*). Esta reformulação resultou na seguinte query:

```
SELECT u.id, u.displayname, (COALESCE(qcount, 0)+COALESCE(acount, 0)+COALESCE(ccount, 0))
    AS total
FROM users u
LEFT JOIN (
   SELECT owneruserid, count(*) AS qcount
   FROM questions WHERE creationdate BETWEEN now() - interval '6 months' AND now()
   GROUP BY owneruserid
) q ON q.owneruserid = u.id
LEFT JOIN (
   SELECT owneruserid, count(*) AS acount
   FROM answers WHERE creationdate BETWEEN now() - interval '6 months' AND now()
   GROUP BY owneruserid
) a ON a.owneruserid = u.id
LEFT JOIN (
   SELECT userid, count(*) AS ccount
   FROM comments WHERE creationdate BETWEEN now() - interval '6 months' AND now()
   GROUP BY userid
) c ON c.userid = u.id
ORDER BY total DESC
LIMIT 100;
```

Anteriormente, ao realizar um LEFT JOIN, por exemplo, entre as tabelas "users" e "questions", cada questão feita por um utilizador resultava numa linha distinta. Em seguida, essas linhas precisavam de ser unidas com a tabela "answers" no próximo *join*. Isso significava que, se um utilizador tivesse feito 20 perguntas e 30 respostas, dentro do intervalo de tempo definido, cada uma das 20 linhas resultantes da junção anterior teria que ser unida a 30 linhas das respostas, resultando numa dimensão de 20×30 linhas para apenas indicar que o utilizador no total fez 50 perguntas ou respostas.

Na abordagem atual, cada uma das tabelas "questions", "answers" e "comments" é examinada para contar antecipadamente quantas interações de cada tipo os utilizadores fizeram. Disto resulta uma tabela com duas colunas (id do utilizador, contagem de interações) para cada tipo de interação. Posteriormente, essas três tabelas são unidas à tabela utilizadores, e a soma das interações é realizada. Com a verificação prévia do número de interações para cada tabela, é reduzida a dimensão dos dados a agregar, sem ter que estar a carregar tantas linhas quantas as questões, respostas ou comentários de um utilizador, e consequentemente sem estar a propagar exponencialmente esse número de linhas em

cada junção de tabelas.

O uso de COALESCE é necessário para lidar com casos em que um utilizador não tenha algum tipo de interação, passando o valor de *NULL* para 0. Por exemplo, num caso em que um utilizador não tem nenhuma entrada na tabela "questions", a tabela resultante do cálculo prévio do número de perguntas não terá nenhuma ocorrência deste utilizador, e quando for feito o LEFT JOIN da tabela "users" com esta tabela, o valor de "qcount" (coluna que conta o número de perguntas de um utilizador) vai estar a *NULL*, representando que o utilizador fez 0 perguntas. Logo, convertemos esse valor de "NULL" para 0, de modo a permitir a soma do número total de interações de um utilizador.

Analisando o desempenho desta nova versão da query reorganizada, notou-se uma melhoria com um tempo de execução médio de **0.6 segundos**. O diagrama do plano de execução da nova versão da query é o seguinte:

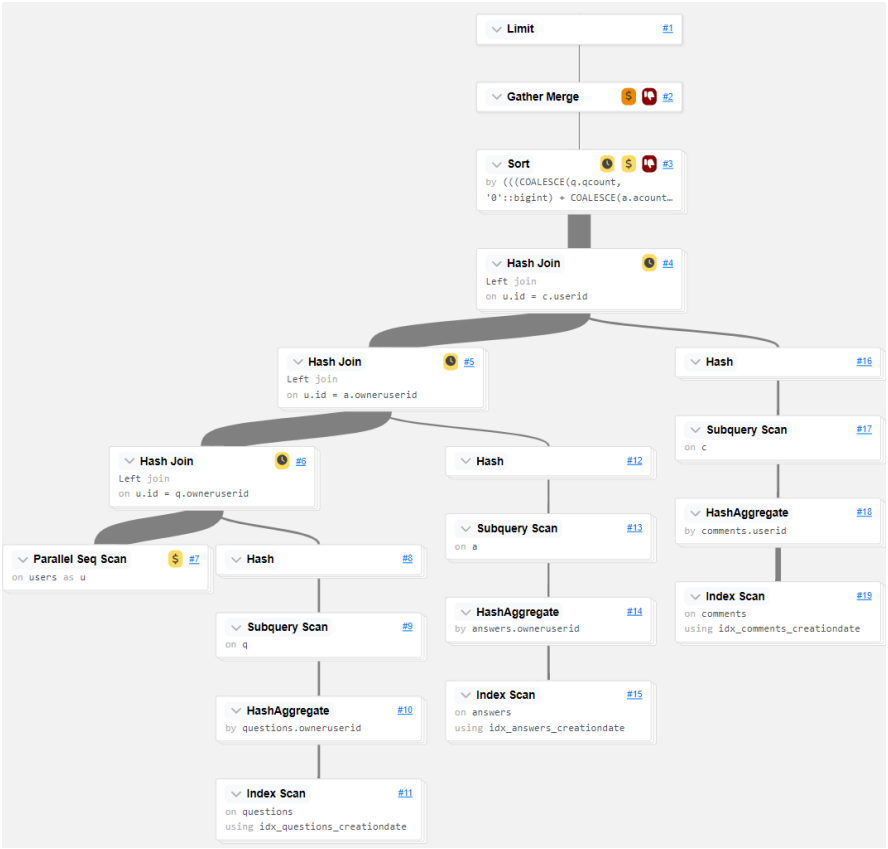


Figura 2.3: Plano de execução da query 1 reestruturada

Após a análise deste diagrama notou-se que a interrogação poderia beneficiar de uma nova reestruturação que fizesse com que apenas tivéssemos que fazer apenas um *join* com a tabela "users", uma tabela que irá ter sempre uma grande dimensão relativamente às outras tabelas intermédias à qual é

junta. A estratégia passará por juntar primeiro as três tabelas intermédias, que indicam o número de cada tipo de interação por utilizador, e apenas juntar essa tabela resultante com a tabela "users" uma vez. Desta forma reduzimos o número de junções com tabelas de grande dimensão que têm um maior impacto na eficiência de uma interrogação.

A query final com esta pequena reestruturação é a seguinte.

```
WITH interactions AS (
    SELECT
        owneruserid,
        COUNT(*) AS interaction_count
    FROM (
        SELECT owneruserid, creationdate FROM questions
        UNION ALL
        SELECT owneruserid, creationdate FROM answers
        UNION ALL
        SELECT userid AS owneruserid, creationdate FROM comments
    ) AS all_interactions
    WHERE creationdate BETWEEN NOW() - INTERVAL '6 months' AND NOW()
    GROUP BY owneruserid
)
SELECT
    u.id,
    u.displayname,
    COALESCE(q.interaction_count, 0) AS total
FROM
    users u
LEFT JOIN interactions q ON q.owneruserid = u.id
ORDER BY total DESC
LIMIT 100;
```

Com esta melhoria, o tempo de execução médio passa a **0.359 segundos**, uma melhoria pequena mas ainda assim notória.

Resumindo, conseguimos passar o tempo de execução de **5.922 segundos** para **0.359 segundos**, recorrendo a três índices na coluna "creationdate" das tabelas "questions", "answers" e "comments", e também a uma reestruturação da query original, que mantém a mesma funcionalidade mas segue uma estratégia mais otimizada para que a dimensão das tabelas intermédias da query seja menor e consequentemente obter melhor desempenho.

Na seguinte tabela são apresentados valores médios de tempos de execução para diferentes valores do argumento do limite inferior de "*creationdate*".

Limite inferior	Tempo base (s)	Tempo final (s)
1 month	5.398	0.188
3 months	5.542	0.255
6 months	5.922	0.359
1 year	7.717	0.983
2 years	19.49	2.232

Tabela 2.1: Execution time of different arguments for Q1

2.2 Interrogação analítica 2

```
WITH buckets AS (
    SELECT year,
        generate_series(0, (
            SELECT cast(max(reputation) as int)
            FROM users
            WHERE extract(year FROM creationdate) = year
        ), 5000) AS reputation_range
    FROM (
        SELECT generate_series(2008, extract(year FROM NOW())) AS year
    ) years
    GROUP BY 1, 2
)
SELECT year, reputation_range, count(u.id) total
FROM buckets
LEFT JOIN (
    SELECT id, creationdate, reputation
    FROM users
    WHERE id in (
        SELECT a.owneruserid
        FROM answers a
        WHERE a.id IN (
            SELECT postid
            FROM votes v
            JOIN votestypes vt ON vt.id = v.votetypeid
            WHERE vt.name = 'AcceptedByOriginator'
                AND v.creationdate >= NOW() - INTERVAL '5 year'
        )
    )
) u ON extract(year FROM u.creationdate) = year
```

```
30      AND floor(u.reputation / 5000) * 5000 = reputation_range
31  GROUP BY 1, 2
32  ORDER BY 1, 2;
```

Esta interrogação analítica consiste no cálculo de uma distribuição anual (para cada ano desde 2008 até ao ano corrente) da reputação dos utilizadores que publicaram respostas que foram "*AcceptedByOriginator*" nos últimos cinco anos. A distribuição da reputação é dividida em diversos intervalos (*buckets*) de amplitude igual a 5000.

Após a interpretação da *query*, foi feita uma análise dos planos de execução gerados pelo PostgreSQL. A *query* apresentava um tempo médio de execução de **7.933 segundos** para os argumentos padrão (o limite inferior de *creationdate* igual 5 anos e o intervalo de cada *bucket* valor predefinido igual a 5000). O plano gerado pelo PostgreSQL é o seguinte:

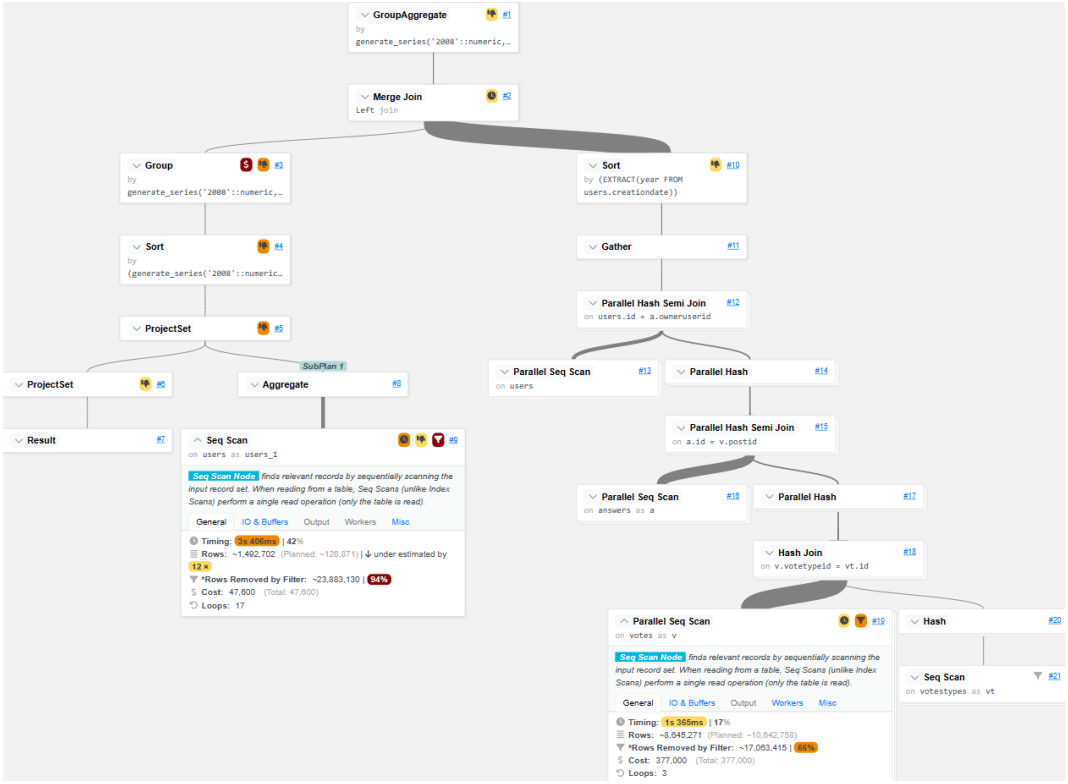


Figura 2.4: Plano do PostgreSQL para a versão base da *query* #2

É possível observar que existem dois *sequential scans* nas tabelas *users* e *votes* (em ambas as tabelas os registos são percorridos para analisar o campo *creationdate*) que ocupam uma parte significativa do tempo de execução da *query*. Desta forma, decidimos verificar o impacto da criação de índices nos campos *creationdate* das tabelas *users* e *votes*:

```sql
CREATE INDEX idx_votes_creationdate ON votes (creationdate);
CREATE INDEX idx_users_creationdate ON users (creationdate);
```

Ao executar a *query* com os novos índices, o tempo de execução médio é de aproximadamente **7.133 segundos** e o plano obtido é o seguinte:

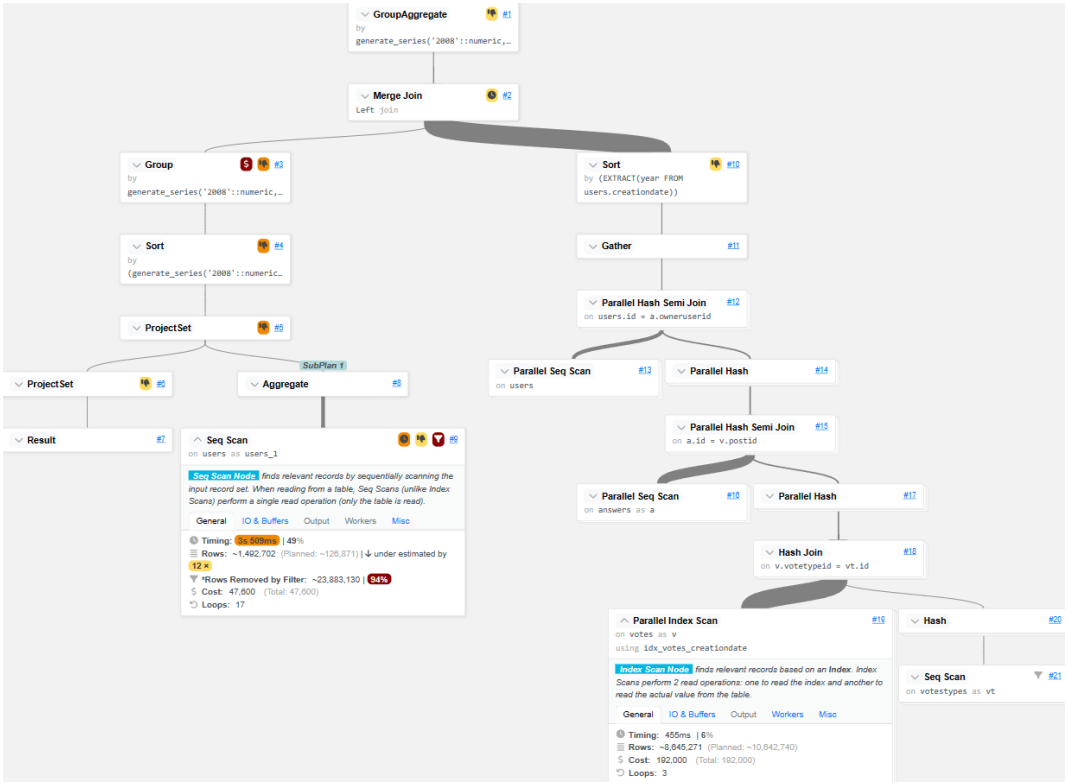


Figura 2.5: Plano do PostgreSQL para a versão da *query* #2 com os índices

Como se pode observar pelo plano acima, a utilização do *index scan* no campo *creationdate* da tabela *votes* contribuiu para a redução do tempo de execução da *query*. No entanto, o índice para o campo *creationdate* da tabela *users* não foi utilizado. A não utilização desse índice pode estar ligada ao facto do *sequential scan* ser realizado nesta parte da query:

```sql
SELECT cast(max(reputation) as int)
FROM users
WHERE extract(year FROM creationdate) = year
```

Como se pode observar, na execução do *sequential scan* é extraído o ano da coluna *creationdate*. Uma vez que que a filtragem está a ser feita pelo valor do ano e não pela coluna em si, o índice nunca será utilizado. Para resolver este problema, recorreu-se então ao seguinte índice:

```sql
CREATE INDEX idx_users_creationdate_year on users (extract(year from creationdate));
```

Ao executar a *query* com este novo índice, o tempo de execução médio é de aproximadamente **3.782 segundos** e o plano obtido é o seguinte:

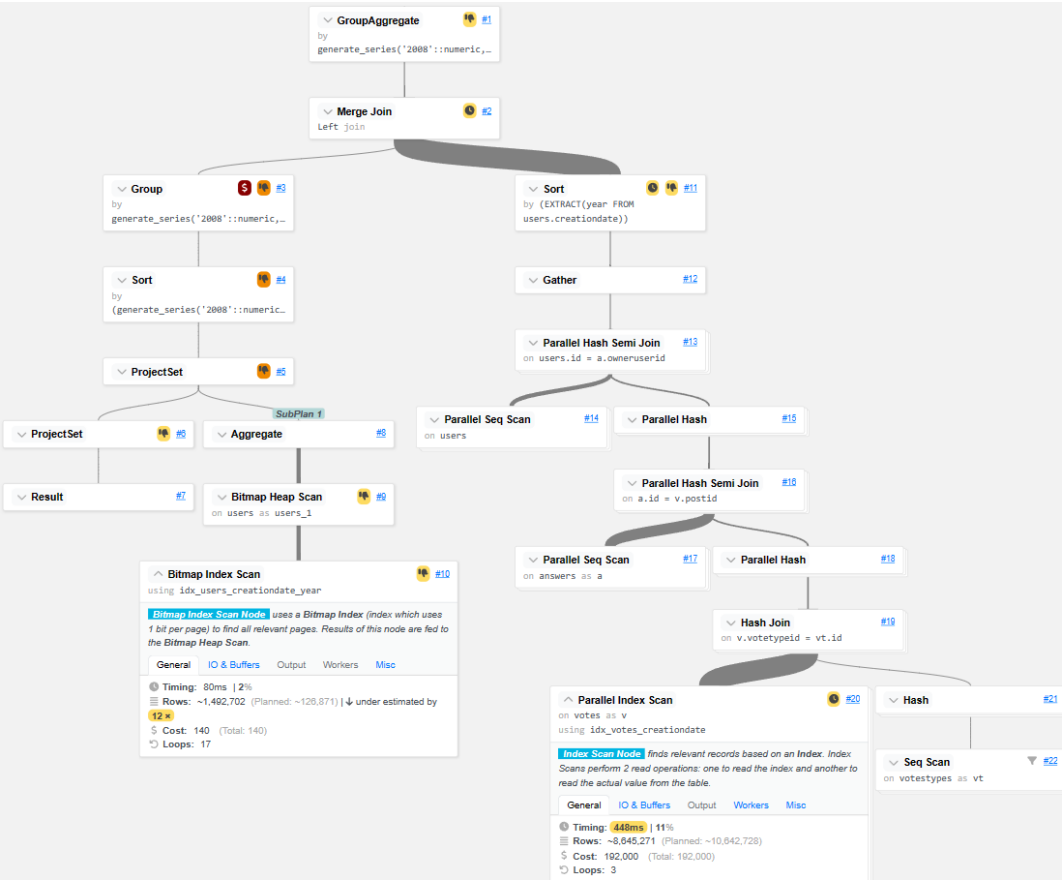


Figura 2.6: Plano do PostgreSQL para a versão da *query* #2 com o índice do ano a ser utilizado

Como se pode observar, este índice contribuiu para uma redução significativa do tempo de execução, pois permitiu que a filtragem pelo ano extraído da data se tornasse muito mais eficiente.

Relativamente ao código da *query*, verificamos que este poderia sofrer algumas alterações de modo a poder melhorar o seu desempenho, nomeadamente na CTE *buckets* utilizada na query:

```
1  WITH buckets AS (
2      SELECT year,
3          generate_series(0, (
4              SELECT cast(max(reputation) as int)
5              FROM users
6              WHERE extract(year FROM creationdate) = year
7          ), 5000) AS reputation_range
8      FROM (
```

```
        SELECT generate_series(2008, extract(year FROM NOW())) AS year
    ) years
    GROUP BY 1, 2
)
```

A CTE *buckets* calcula intervalos de amplitude 5000 desde 0 até à reputação máxima dos utilizadores para cada ano de criação da sua conta. Por exemplo, se a maior reputação entre todos os utilizadores que criaram conta em 2008 é 100000, então a CTE *buckets* terá várias linhas cuja coluna *year* é 2008 e a coluna *reputation_range* a variar de 0 a 100000 de 5000 em 5000. A CTE terá isso para todos os anos até 2024.

Ao analisar o código da CTE, podemos observar que existem dois pontos onde pode haver melhorias: (i) o `GROUP BY 1,2` existente é desnecessário, pois nunca haverá pares (*year*,*reputation_range*) iguais e (ii) para cada ano (2008 → 2024) faz-se uma travessia da tabela *users* para obter a reputação máxima para o respectivo ano. Ou seja, a tabela *users* está a ser percorrida 17 vezes. Para comprovar isto, podemos recorrer a esta informação presente no plano da query:

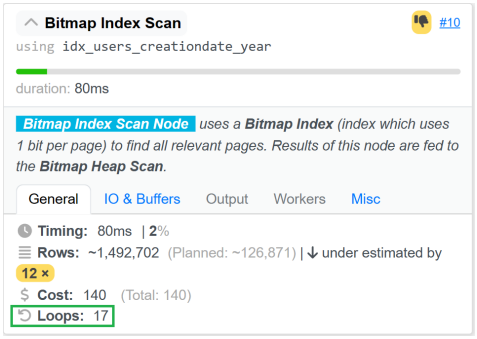


Figura 2.7: Tabela *users* percorrida 17 vezes para encontrar a reputação máxima em cada ano

Assim, para melhorar o desempenho do cálculo da CTE *buckets*, teremos de retirar o *group by* e ter uma nova abordagem para obter as reputações máximas para cada ano. Elaborou-se então a seguinte versão da CTE *buckets*:

```
WITH year_range AS (
    SELECT generate_series(2008, extract(year from NOW())) AS year
), max_reputation_per_year AS (
    SELECT extract(year FROM creationdate) AS year, cast(max(reputation) as int) AS max_rep
    FROM users
    GROUP BY extract(year FROM creationdate)
), buckets as (
    SELECT yr.year, generate_series(0, mr.max_rep, 5000) AS reputation_range
    FROM year_range yr
```

```
10      JOIN max_reputation_per_year mr ON yr.year = mr.year
11  )
```

Nesta versão, temos três CTEs, sendo que duas delas são apenas auxiliares da CTE *buckets*. A *year_range* é simplesmente a sequência de anos desde 2008 até 2024 e a *max_reputation_per_year* é a reputação máxima de utilizadores por cada ano da criação da conta (*creationdate*). A CTE *buckets* passa a ser um JOIN das duas CTEs auxiliares. Desta forma, a CTE *buckets* consegue gerar o mesmo resultado, pois consegue obter do join o ano e a reputação máxima nesse ano, só tendo posteriormente de calcular os intervalos de 0 à reputação máxima. A vantagem desta alteração, é que tivemos de percorrer a tabela *users* apenas uma única vez para obter as reputações máximas por ano. A prova disso é esta informação plano da *query* após ter sido efectuada essa alteração:

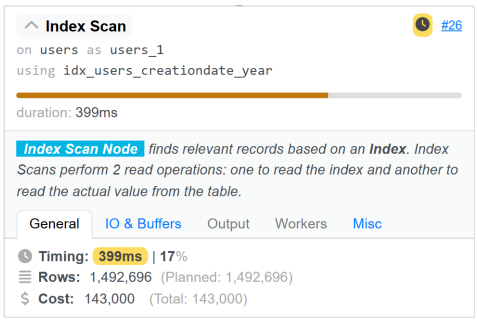


Figura 2.8: Tabela *users* percorrida uma vez para encontrar a reputação máxima em cada ano

Com a informação acima foi possível observar que, na realidade, apesar de se percorrer apenas uma vez a tabela *users* para obter a reputação máxima, a melhoria de desempenho não resultou dessa redução de execuções. Isto deve-se ao facto do tempo de execução dessa etapa da *query* ter passado de 80ms para 399ms (ou seja, devido à segunda versão não utilizar índices *bitmap*). Desta forma, tivemos de investigar outras etapas no plano da *query*.

Ao reanalisar e comparar os planos das duas versões da *query*, foi possível observar o seguinte:

Figura 2.9: *Merge join* versão original

Figura 2.10: *Merge join* versão reestruturada

Acima conseguimos ver que a verdadeira melhoria na reescrita da query foi neste *merge join* (correspondente ao `LEFT JOIN` da *query*). Esta melhoria ocorreu devido ao facto do optimizador alterar uma operação de ordenação que ocorre antes do *merge join*, conforme se pode observar nas figuras seguintes.

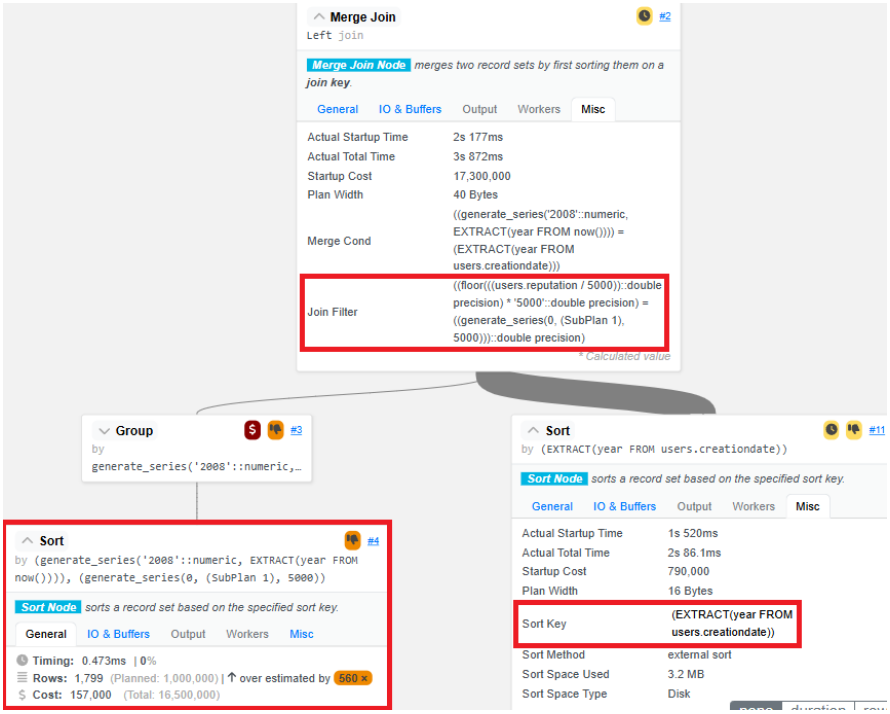


Figura 2.11: *Sort* e *Merge join* versão original

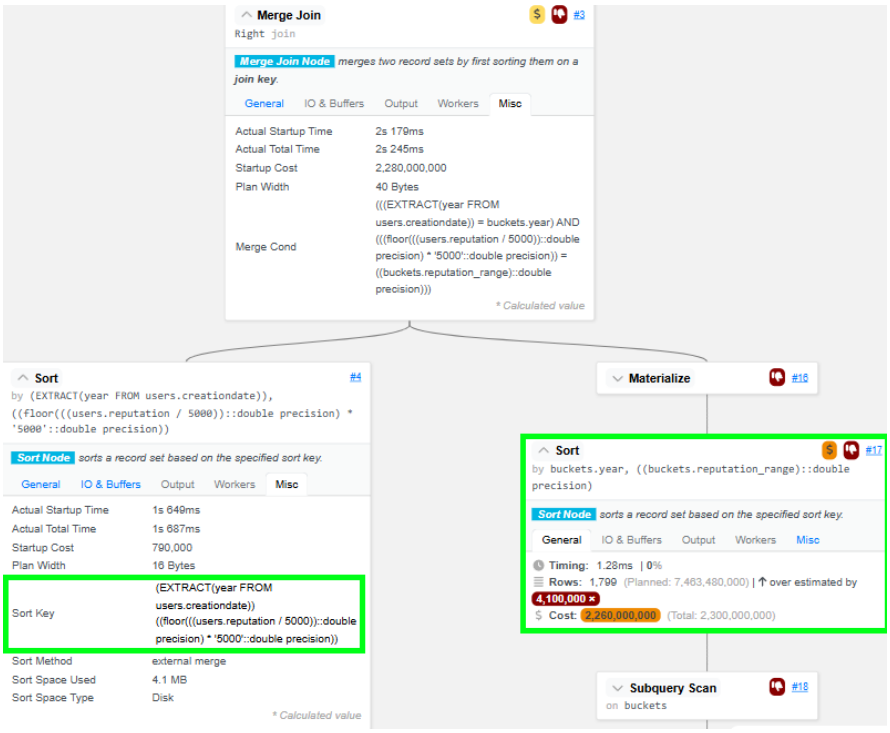


Figura 2.12: *Sort* e *Merge join* versão reestruturada

Nas figuras acima, podemos perceber que a CTE *buckets* é ordenada pelos campos *year* e *reputation* em ambas as versões da *query*. No entanto, o mesmo não se aplica à *subquery* que envolve as tabelas *users*, *answers*, *votes* e *votestypes*. No caso dessa *subquery*, na versão original ela é apenas ordenada pelo *year*, enquanto que na versão reestruturada ela é ordenada pelo *year* e *reputation*.

Na nova versão da *query*, o optimizador aparenta ter alterado a operação de ordenação à *subquery* de forma a permitir que tanto a CTE e a *subquery* fiquem ordenadas pelo *year* e *reputation*. Desta forma, a CTE e a *subquery* utilizadas na *query* estão ambas ordenadas pelas colunas *year* e *reputation* que constituem a condição do *merge join*. Consequentemente, o *merge join* será mais eficiente por não ter a necessidade de ler todas as linhas da *subquery*, uma vez que a CTE e a *subquery* estão ordenadas pela sua condição.

No caso da versão original, como as linhas da *subquery* estão apenas ordenadas pelo ano, o *merge join* terá de fazer operações adicionais por ter a necessidade de verificar a condição da coluna *reputation* (isto pode ser verificado na figura 2.11 em que o *merge join* realiza uma operação *Join Filter* que impõe a leitura da *subquery* inteira para filtrá-la pela coluna *reputation*). Esta operação adicional poderá constituir uma causa para a degradação de desempenho observada na versão original da *query*.

Com a reestruturação à CTE *buckets*, foi possível reduzir o tempo de execução (com os argumentos padrão) para aproximadamente **2.235 segundos**.

A versão final da query com todas as alterações ao seu código SQL é a seguinte:

```
WITH year_range AS (
    SELECT generate_series(2008, extract(year from NOW())) AS year
), max_reputation_per_year AS (
    SELECT extract(year FROM creationdate) AS year, cast(max(reputation) as int) AS max_rep
    FROM users
    GROUP BY extract(year FROM creationdate)
), buckets as (
    SELECT yr.year, generate_series(0, mr.max_rep, 5000) AS reputation_range
    FROM year_range yr
    JOIN max_reputation_per_year mr ON yr.year = mr.year
)
SELECT year, reputation_range, count(u.id) total
FROM buckets
LEFT JOIN (
    SELECT id, creationdate, reputation
    FROM users
    WHERE id in (
        SELECT a.owneruserid
        FROM answers a
```

```
        WHERE a.id IN (
            SELECT postid
            FROM votes v
            JOIN votestypes vt ON vt.id = v.votetypeid
            WHERE vt.name = 'AcceptedByOriginator'
                AND v.creationdate >= NOW() - INTERVAL '5 year'
        )
    )
) u ON extract(year FROM u.creationdate) = year AND floor(u.reputation / 5000) * 5000 =
    reputation_range
GROUP BY 1, 2
ORDER BY 1, 2;
```

Na tabela seguinte são apresentados valores médios de tempos de execução para diferentes valores dos argumentos da *query*:

Intervalo	Bucket	Base sem índices (seg)	Base com índices (seg)	CTE *buckets* alterado (seg)
1 year	5000	5.723	1.448	1.285
1 year	15000	5.688	1.451	1.282
1 year	25000	5.676	1.442	1.278
5 year	5000	7.933	3.782	2.235
5 year	15000	7.941	3.798	2.234
5 year	25000	7.941	3.791	2.256
7 year	5000	9.262	5.277	2.693
7 year	15000	9.259	5.259	2.697
7 year	25000	9.271	5.280	2.711

Tabela 2.2: Evolução do desempenho da query #2 com as optimizações aplicadas

Na tabela acima, as duas primeiras colunas correspondem a argumentos fornecidos à *query* para testar o desempenho, a terceira coluna corresponde a testes de desempenho à *query* original (sem qualquer optimização), a quarta coluna corresponde a testes de desempenho com a *query* original (código SQL original), mas com a aplicação dos índices que foram abordados anteriormente, e a última coluna corresponde a testes de desempenho com a *query* reescrita juntamente com os índices.

2.3 Interrogação analítica 3

```
SELECT tagname, round(avg(total), 3), count(*)
```

```
FROM (
    SELECT t.tagname, q.id, count(*) AS total
    FROM tags t
    JOIN questionstags qt ON qt.tagid = t.id
    JOIN questions q ON q.id = qt.questionid
    LEFT JOIN answers a ON a.parentid = q.id
    WHERE t.id IN (
        SELECT t.id
        FROM tags t
        JOIN questionstags qt ON qt.tagid = t.id
        JOIN questions q ON q.id = qt.questionid
        GROUP BY t.id
        HAVING count(*) > 10
    )
    GROUP BY t.tagname, q.id
)
GROUP BY tagname
ORDER BY 2 DESC, 3 DESC, tagname;
```

Esta query consiste no cálculo da média do número total de perguntas e respostas para cada tag, agrupadas por tag.

Primeiro procedemos à simplificação da sintaxe SQL da query em si. Notamos que havia vários joins desnecessários para a essência funcional da query. Para começar simplificou-se a subquery interior do `where t.id in ...` que não precisa de fazer o join com as outras duas tabelas 'questions' e 'tags', bastando apenas a tabela questiontags para deduzir em quantas questões estão as várias tags existentes.

```
WHERE t.id IN (
    SELECT tagid
    FROM questionstags
    GROUP BY tagid
    HAVING count(*) > 10
)
```

De seguida removeu-se o *join* com *questions* da subquery intermédia, uma vez que a informação do id das questões já estava presente na tabela *questionstags*, podendo trabalhar a partir desse *qt.tagid* em vez do *q.id*. Esta alteração é possível porque os valores da coluna *q.id* são únicos, logo não acrescentam linhas à tabela resultante do *join* removido e como também a *query* não precisa da informação extra

que vem da tabela *questions*, o *join* é desnecessário, tendo sido removido.

```
SELECT t.tagname, qt.questionid, count(*) AS total
FROM tags t
JOIN questionstags qt ON qt.tagid = t.id
LEFT JOIN answers a ON a.parentid = qt.questionid
WHERE t.id IN (
    SELECT tagid
    FROM questionstags
    GROUP BY tagid
    HAVING count(*) > 10
)
GROUP BY t.tagname, qt.questionid
```

De forma a retirar a complexidade da *inner query*, tentou-se realizar o *join* da tabela *tags* na *outer query*. Desta forma, apenas se juntava a informação da tabela tags no final. No entanto, esta query não obteve melhores resultados do que a anterior.

Analisando a *query* é possível verificar que seria interessante adicionar um índice que agrupa as colunas *tagid* e *questionid* da tabela *questionstags* já que estão presentes na ordenação dos resultados. No entanto, como já existe um índice na *Key* primária que envolve essas duas colunas, a adição desse índice não traz melhorias.

```
SELECT t.tagname, round(avg(total), 3), count(*)
FROM (
    SELECT qt.tagid, qt.questionid, count(*) AS total
    FROM questionstags qt
    LEFT JOIN answers a ON a.parentid = qt.questionid
    WHERE qt.tagid IN (
        SELECT tagid
        FROM questionstags
        GROUP BY tagid
        HAVING count(*) > 10
    )
    GROUP BY qt.tagid, qt.questionid
)
LEFT JOIN tags t ON t.id = tagid
GROUP BY t.tagname
ORDER BY 2 DESC, 3 DESC, tagname;
```

Tentamos estruturar a query desta maneira para fazer com que os filtros fossem feitos o mais rápido possível na query para que a computação seja mais leve com menos dados, mas verificando o plano da versão anterior da query, podemos verificar que esta filtragem prévia, já foi delineada pelo planeador, não se notando melhorias na performance.

```
SELECT t.tagname, round(avg(total), 3), count(*)
FROM (
    SELECT tagid, questionid, count(*) AS total
    FROM (
        SELECT qt.tagid, qt.questionid
        FROM questionstags qt
        WHERE qt.tagid IN (
            SELECT tagid
            FROM questionstags
            GROUP BY tagid
            HAVING count(*) > 10
        )
    ) as qtf
    LEFT JOIN answers a ON a.parentid = qtf.questionid
    GROUP BY qtf.tagid, qtf.questionid
)
LEFT JOIN tags t ON t.id = tagid
GROUP BY t.tagname
ORDER BY 2 DESC, 3 DESC, tagname;
```

De seguida, utilizamos uma *common table expression* (CTE) para pré-calcular a lista de tag IDs que têm mais de 10 ocorrências. No entanto, a performance desta query continuou a não ser a desejada.

```
WITH questionstags_tagid as
    (SELECT tagid
    FROM questionstags
    GROUP BY tagid
    HAVING count(*) > 10)
SELECT tagname, round(avg(total), 3), count(*)
FROM (
    SELECT t.tagname, qt.questionid, count(*) AS total
    FROM tags t
    JOIN questionstags qt ON qt.tagid = t.id
    inner join questionstags_tagid qtt on qtt.tagid = qt.tagid
    LEFT JOIN answers a ON a.parentid = qt.questionid
```

```
    GROUP BY t.tagname, qt.questionid
)
GROUP BY tagname
ORDER BY 2 DESC, 3 DESC, tagname;
```

Reestruturamos a query para que a lógica da filtragem das tags com mais de 10 ocorrências esteja separada numa CTE designada "FilteredTags". Isso torna a interrogação principal mais concisa e focada apenas no cálculo da média e da contagem de tags.

```
WITH TagQuestionCounts AS (
    SELECT qt.tagid, qt.questionid, COUNT(*) AS total
    FROM questionstags qt
    LEFT JOIN answers a ON a.parentid = qt.questionid
    GROUP BY qt.tagid, qt.questionid
),
FilteredTags AS (
    SELECT tagid
    FROM TagQuestionCounts
    GROUP BY tagid
    HAVING COUNT(*) > 10
)
SELECT t.tagname, ROUND(AVG(tqc.total), 3) AS avg_total, COUNT(*) AS tag_count
FROM TagQuestionCounts tqc
JOIN FilteredTags ft ON ft.tagid = tqc.tagid
LEFT JOIN tags t ON t.id = tqc.tagid
GROUP BY t.tagname
ORDER BY avg_total DESC, tag_count DESC, t.tagname;
```

Observando o `explain analyze` da query otimizada, podemos ainda verificar que existem muitos *seq scans*.

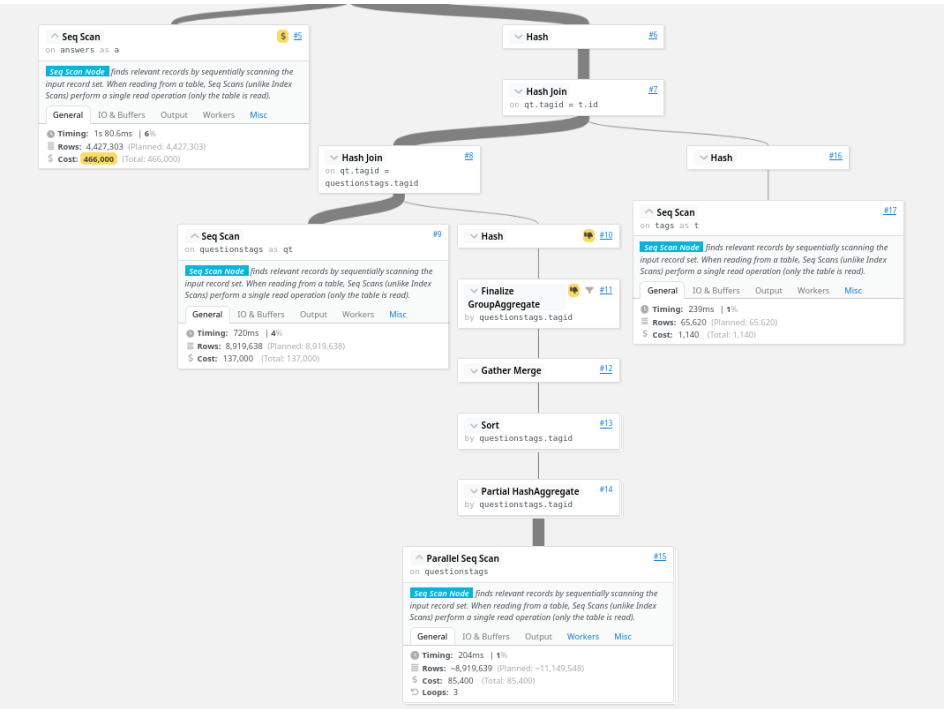
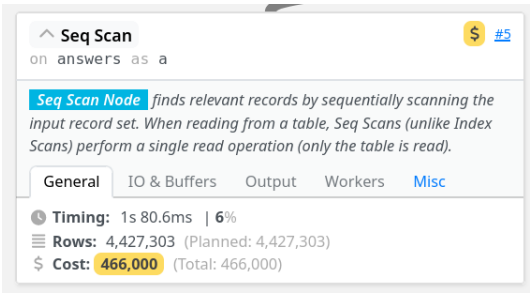


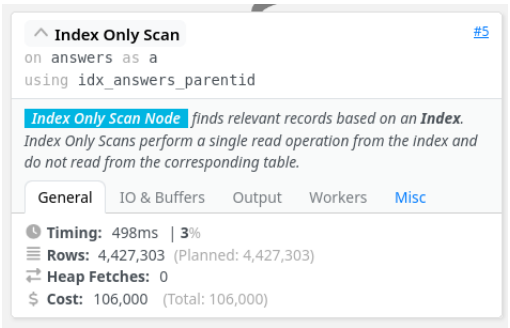
Figura 2.13: Explain Analyse

Foram testados os seguintes índices para aumentar a eficiência da pesquisa:

- CREATE INDEX idx_tags_id ON tags (id);
- CREATE INDEX idx_questionstags_tagid ON questionstags (tagid);
- CREATE INDEX idx_questionstags_questionid ON questionstags (questionid);
- CREATE INDEX idx_questionstags_tagid_questionid ON questionstags (tagid, questionid);
- CREATE INDEX answers_idx_parentid ON “answers” (“parentid”);
- CREATE INDEX tags_idx_id_tagname ON “tags” (“id”,“tagname”);
- CREATE INDEX idx_tags_tagname ON tags (tagname);

Apenas o índice na tabela answers melhorou os resultados.





Olhando para a segunda versão, até ao momento mais otimizada:

```
SELECT tagname, round(avg(total), 3), count(*)
FROM (
    SELECT t.tagname, qt.questionid, count(*) AS total
    FROM tags t
    JOIN questionstags qt ON qt.tagid = t.id
    LEFT JOIN answers a ON a.parentid = qt.questionid
    WHERE t.id IN (
        SELECT tagid
        FROM questionstags
        GROUP BY tagid
        HAVING count(*) > 10
    )
    GROUP BY t.tagname, qt.questionid
)
GROUP BY tagname
ORDER BY 2 DESC, 3 DESC, tagname;
```

Optamos por nos focar nos parâmetros de configuração do postgres. Recorrendo ao comando "show work_mem" pudemos verificar que por default tinha 4MB. Isso era o suficiente para esta query, já que, tal como é possível ver pelo explain analyse, não é preciso aumentar, porque está tudo em memória, não recorrendo a discos externos:

```
Sort Method: quicksort Memory: 2040kB
```

Observando o group by, percebemos que é realizado por uma HashAggregate.

```
HashAggregate (cost=1717501.83..1717504.83 rows=200 width=52) (actual
    time=21004.202..21015.858 rows=23951 loops=1)
        Group Key: t.tagname
        Batches: 1 Memory Usage: 4641kB
        -> HashAggregate (cost=1451035.92..1598600.71 rows=6794350 width=28) (actual
            time=14954.992..19450.880 rows=8802687 loops=1)
             Group Key: t.tagname, qt.questionid
             Planned Partitions: 128 Batches: 641 Memory Usage: 8249kB Disk Usage: 646336kB
```

Podemos desativar essa opção, para que seja realizado por um groupAggregate com o comando "set enable_hashagg=off;". No entanto, desta forma obteve uma perda significativa de performance.

```
GroupAggregate (cost=2511417.02..2766208.14 rows=200 width=52) (actual
    time=30386.024..35580.153 rows=23951 loops=1)
        Group Key: t.tagname
        -> GroupAggregate (cost=2511417.02..2647304.02 rows=6794350 width=28) (actual
            time=30386.001..34992.719 rows=8802687 loops=1)
             Group Key: t.tagname, qt.questionid
```

Com estas otimizações apenas conseguimos obter um ganho de cerca de 35%, o que representa 7 segundos. Porém, num caso singular de teste, ao recorrer a vistas materializadas o ganho pode passar para 82.5% porcento. No entanto, não é viável usar vistas materializadas uma vez que os registos das tabelas podem mudar, e sendo a tabela questiontags a que tem mais registos, a computação necessária para atualizar a vista materializada não compensa.

```sql
CREATE MATERIALIZED VIEW TagQuestionCounts AS
    SELECT qt.tagid, qt.questionid, COUNT(*) AS total
    FROM questionstags qt
    LEFT JOIN answers a ON a.parentid = qt.questionid
    GROUP BY qt.tagid, qt.questionid

WITH FilteredTags AS (
    SELECT tagid
    FROM TagQuestionCounts
    GROUP BY tagid
    HAVING COUNT(*) > 10
)

SELECT t.tagname, ROUND(AVG(tqc.total), 3) AS avg_total, COUNT(*) AS tag_count
FROM TagQuestionCounts tqc
JOIN FilteredTags ft ON ft.tagid = tqc.tagid
LEFT JOIN tags t ON t.id = tqc.tagid
GROUP BY t.tagname
ORDER BY avg_total DESC, tag_count DESC, t.tagname;
```

A query aborda três tabelas: *tags*, *questiontags* e *answers*. No enunciado é indicado que a tabela *tags* é estática, por isso pode ser completamente materializada. No entanto, tanto as tabelas *answers* como *questionstags* podem ter registos a serem eliminados ou inseridos. Pelo que nunca é possível materializar estas duas tabelas sem comprometer a performance na atualização da materialized view.

Assim, com a otimização do código, a inserção da materialized view na tabela *tags* e o índice na tabela *answers*, obtemos os seguintes resultados, onde se altera o argumento de número de ocorrências de cada *tagid* na tabela *questionstags*:

Arguments	Time (s)
10	24.85
20	24.611
30	24.472
40	24.401
50	24.143

Figura 2.14: Resultados query original

Arguments	Time (s)
10	19.481
20	19.148
30	19.006
40	18.828
50	18.66

Figura 2.15: Resultados query com índice

Arguments	Time (s)
10	17.808
20	17.32
30	17.358
40	17.117
50	16.907

Figura 2.16: Resultados query com índice e materialized view

2.4 Interrogação analítica 4

```sql
SELECT date_bin('1 minute', date, '2008-01-01 00:00:00'), count(*)
FROM badges
WHERE NOT tagbased
    AND name NOT IN (
        'Analytical',
        'Census',
        'Documentation Beta',
        'Documentation Pioneer',
        'Documentation User',
        'Reversal',
        'Tumbleweed'
    )
    AND class in (1, 2, 3)
    AND userid <> -1
GROUP BY 1
```

```
16  ORDER BY 1;
```

Com a query analítica base, foi executada a script *Python* que verifica os tempos para diversos argumentos com a query 4 de base e obtiveram-se os seguintes resultados.

Argumentos	Tempo (s)
1 minute	6.44
10 minutes	5.22
30 minutes	3.83
1 hour	3.45
6 hours	3.38

Tabela 2.3: Base Execution Times for Q4

De forma a perceber o plano de execução da query, recorremos à visualização do plano de execução gerado.

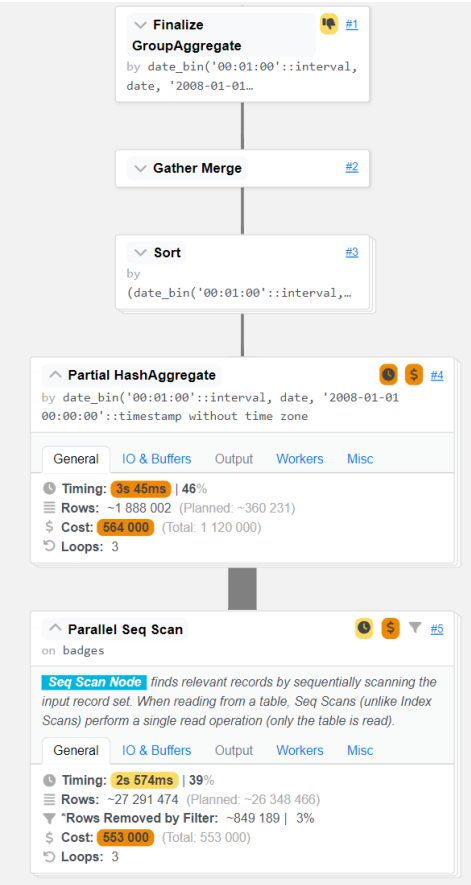


Figura 2.17: Plano de execução da query 4 base

Com base no plano gerado, podemos verificar que a query se divide em duas partes principais que

ocupam a maior parte do tempo de execução da query: o filtro da tabela "badges", e a agregação das linhas resultantes por date_bins.

Inicialmente, atacando a parte da agregação, pensamos em criar um índice na coluna "data", mas verificou-se que o índice não era utilizado efetivamente, uma vez que o agrupamento é feito em relação a uma nova coluna criada pela função `date_bin()`, não em relação à coluna original "data". Para além disso não é possível usar um *índice em expressão* porque o resultado da nova coluna que agrupa a data com a função `date_bin()`, é dependente de um argumento que se passa à query. Outra tentativa foi criar índices relativos às colunas que são alvo de uma filtragem, como a coluna "name" e "tagbased". Testamos índices individuais e índices compostos, no entanto esta abordagem também não refletiu em melhorias, uma vez que este filtro apenas remove uma pequena percentagem de linhas (3%), como se pode ver pelo operador *Parallel Seq Scan* na figura 2.17. Nesta situação, o otimizador de consultas pode não ter optado por usar os índices, por considerar mais eficiente fazer um *scan* na tabela do que usar os índices, pois o custo de aceder um índice e, em seguida, recuperar os registos correspondentes pode ser maior do que simplesmente examinar a tabela.

Por fim, decidimos envolver vistas materializadas na query, dada a falta de melhorias através de outros métodos. No plano de execução é possível verificar que a filtragem inicial da tabela badges, ainda ocupa uma certa parte do tempo da query. Como esta filtragem não está dependente de nenhum tipo de argumento da query, torna-se um alvo ideal para materialização.

```
1  CREATE MATERIALIZED VIEW badges_mat_view AS
2  SELECT date
3  FROM badges
4  WHERE NOT tagbased
5      AND name NOT IN (
6          'Analytical',
7          'Census',
8          'Documentation Beta',
9          'Documentation Pioneer',
10         'Documentation User',
11         'Reversal',
12         'Tumbleweed'
13     )
14     AND class in (1, 2, 3)
15     AND userid <> -1;
```

Dada a utilização de uma *materialized view*, é preciso garantir a sua consistência com a entrada de novos dados relativos às tabelas que envolvem uma *materialized view*, neste caso, apenas a tabela "badges", uma vez que é esperado que o resultado considere informação em tempo real.

Para isso surge a necessidade de *triggers* que mantenham esta *materialized view* atualizada à medida que a tabela `badges` é alterada.

As *materialized view* são boas no sentido em que melhoram a performance de uma query analítica, mas por outro lado, esta melhoria vem à custa de escritas mais lentas devido à necessidade de manter a informação entre as tabelas e a *materialized view* coerente. Se a frequência das atualizações das tabelas sobre as quais a *materialized view* está dependente for grande, esta melhoria pode não compensar. No entanto, pode-se argumentar que a criação desta *materialized view* não traz assim tanto impacto negativo se considerarmos que a *materialized view* apenas é dependente da tabela "badges" e que a carga transacional não tem nenhuma operação de atualização nesta tabela "badges".

Com esta *materialized view* é visível que a eliminação da computação do filtro se traduziu numa melhoria dos tempos na ordem dos 2 segundos, sendo estes os valores de tempo finais medidos:

Argumentos	**Tempo (s)**
1 minute	4.869
10 minutes	3.62
30 minutes	2.232
1 hour	1.841
6 hours	1.772

Tabela 2.4: Final Execution Times for Q4

3 Carga analítica no *Spark*

Esta secção dedica-se à explicação das estratégias abordadas para melhorar a performance da carga analítica num *snapshot* da base de dados em Apache Spark. Neste contexto, os dados permanecem estáticos durante longos períodos de tempo, o que permite seguir estratégias mais eficientes de otimização e pré-processamento.

O *Spark* foi instalado via Docker, e a avaliação inicial da performance das interrogações analíticas foram realizadas com base num *cluster* com 1 *master* e 3 *workers*, com as seguintes configurações:

- `sql.adaptive.enabled = true;`
- `spark.executor.instances = 3`
- `spark.driver.memory = 8g`

Para avaliar as interrogações analíticas, recorremos a várias ferramentas e utilidades. De modo a medir tempos de execução, recorremos a uma funcionalidade do Python, os *decorators*, que foi

utilizada para medir o tempo de uma determinada tarefa e gerar as médias da execução de tempo de uma query automaticamente. Com o intuito de analisar mais aprofundadamente a execução das queries, recorremos ao operador `explain(mode="extended")` mas principalmente ao *history server* que fornece uma perspetiva mais visual dos planos de execução das queries. Para aceder ao *history server* na máquina virtual, criamos um túnel *ssh* que permite aceder às portas da maquina virtual através do nossa máquina local.

Ao trabalhar com uma ferramenta de *data warehousing* como o Spark, é crucial selecionar o formato de ficheiro mais eficiente para armazenar os dados. Após avaliar o desempenho dos diferentes formatos de ficheiro com versões básicas das interrogações analíticas em Spark, foi possível verificar que a utilização do formato Parquet traz uma performance muito superior ao formato CSV. Em certas ocasiões, mais concretamente ao efetuar uma versão simples da interrogação analítica 1, a diferença chega a ser de 50 segundos para 10 segundos. Logo, optamos por utilizar os dados no formato Parquet, trazendo benefícios como armazenamento e operações I/O eficientes através do formato de armazenamento colunar, utilização reduzida de espaço em disco e mecanismos de otimização de desempenho de interrogações.

Relativamente aos testes de tempo de execução das queries, os valores que irão ser mencionados ao longo desta secção são o tempo médio relativo à execução das queries por 6 vezes para cada argumento apresentado, descartando a primeira execução que traz valores anormalmente altos devido à leitura inicial dos dados, resultante do *laziness* do Spark que adia a execução da operação de leitura até que uma ação seja chamada.

3.1 Interrogação analítica 1

A partir da estrutura SQL da versão final da respetiva query SQL, obtemos uma versão base de Spark que obtém os seguintes resultados temporais.

Limite inferior de *creationdate* (argumento)	Tempo médio(s)
1 month	2.047
3 months	1.825
6 months	1.741
1 year	1.924
2 years	2.171

Tabela 3.1: Tempos de execução do Workload 1 base

Na query 1, partindo desta estrutura base, não é possível obter grandes melhorias no que toca

a extrair o máximo de pré-computação possível para ficheiro, uma vez que o argumento da query é necessário relativamente cedo na query. A única possibilidade passaria por realizar previamente a união das três tabelas (questions, answers e comments), mas essa operação não é muito pesada e não se verifica nenhuma melhoria na prática.

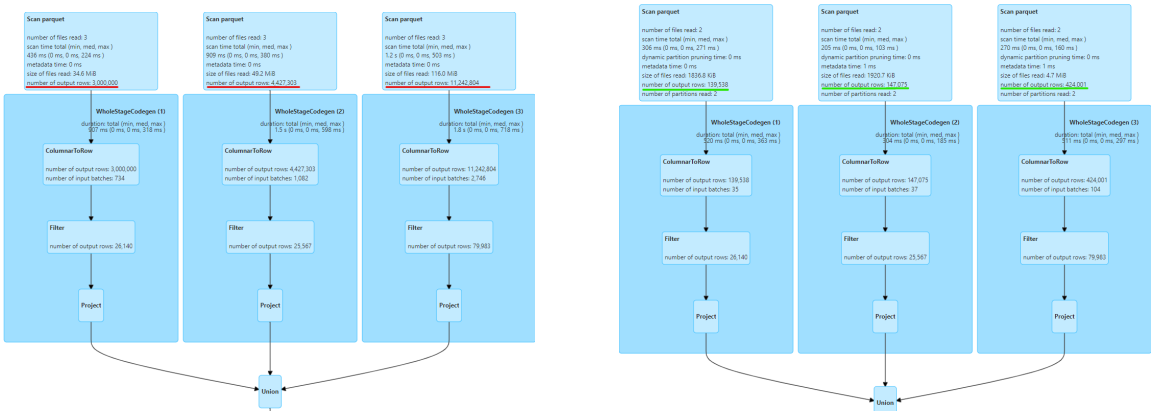
Abordando uma nova estratégia, pensamos em utilizar uma estratégia de particionamento dos ficheiros parquet pela coluna `creationdate` que é utilizada pelo filtro da query e permitir que o Spark possa saltar partições que não contêm os dados que a query procura, um processo também chamado *partition pruning*. Mas esta estratégia tinha um problema relativamente à cardinalidade da coluna `creationdate`, pois a função `partitionBy()` cria uma partição para cada valor único da coluna.

Aproveitando a ideia anterior, surgiu uma nova estratégia com a extração do ano da coluna `creationdate`, criando uma nova coluna `creationyear` que indica apenas o ano da data de criação. Desta forma reduzimos a cardinalidade da coluna pela qual é feito o particionamento e permite-nos tirar partido de *partition pruning*. Depois apenas tem de ser utilizada essa coluna no filtro da query para que seja o Spark seja capaz de detetar que pode aplicar *partition pruning* baseado na coluna `creationyear`.

```
1  def q1_year(users: DataFrame, questions: DataFrame, answers: DataFrame, comments:
       DataFrame, interval: StringType = "6 months") -> List[Row]:
2
3      questions_selected = questions.select("owneruserid", "creationdate", "creationyear")
4      answers_selected = answers.select("owneruserid", "creationdate", "creationyear")
5      comments_selected = comments.select(col("userid").alias("owneruserid"), "creationdate",
           "creationyear")
6
7      lower_interval = current_timestamp() - expr(f"INTERVAL {interval}")
8      lower_interval_year = year(lower_interval)
9
10     interactions = (
11         questions_selected
12         .union(answers_selected)
13         .union(comments_selected)
14         .filter((col("creationyear") >= lower_interval_year) &
               (col("creationdate").between(lower_interval, current_timestamp())))
15         .groupBy("owneruserid")
16         .agg(count("*").alias("interaction_count"))
17     )
18
19     result_df = (
20         users
```

```
        .join(broadcast(interactions), users["id"] == interactions["owneruserid"], "left")
        .select(
            users["id"],
            users["displayname"],
            coalesce(interactions["interaction_count"],
                lit(0)).cast(IntegerType()).alias("total")
        )
        .orderBy(col("total").desc())
        .limit(100)
    )
    return result_df.collect()
```

Na seguinte imagem comparamos lado a lado os planos de execução, mais especificamente os detalhes do operador "*Scan parquet*", da query 1 com o limite inferior de *creationdate* definido como "6 months". No campo de *number of output rows* do operador *Scan parquet*, é possível verificar que a estratégia de particionamento por ano, levou a uma redução significativa do número de linhas lidas, e consequentemente, na quantidade de dados que foram necessários ler de disco, através de uma técnica de *data pruning*.



(a) Sem particionamento

(b) Com particionamento

Figura 3.1: Comparação da quantidade de dados lida sem e com particionamento pelo ano

Para além do particionamento, reparamos que existia um *join* em que uma das tabelas era significativamente menor (tabela *users+questions+comments* filtrada) que a outra (tabela dos *users*) e surgiu a ideia de utilizar um *broadcast hint*.

Recorrendo ao plano de execução podemos ver a diferença de tamanhos das duas tabelas.

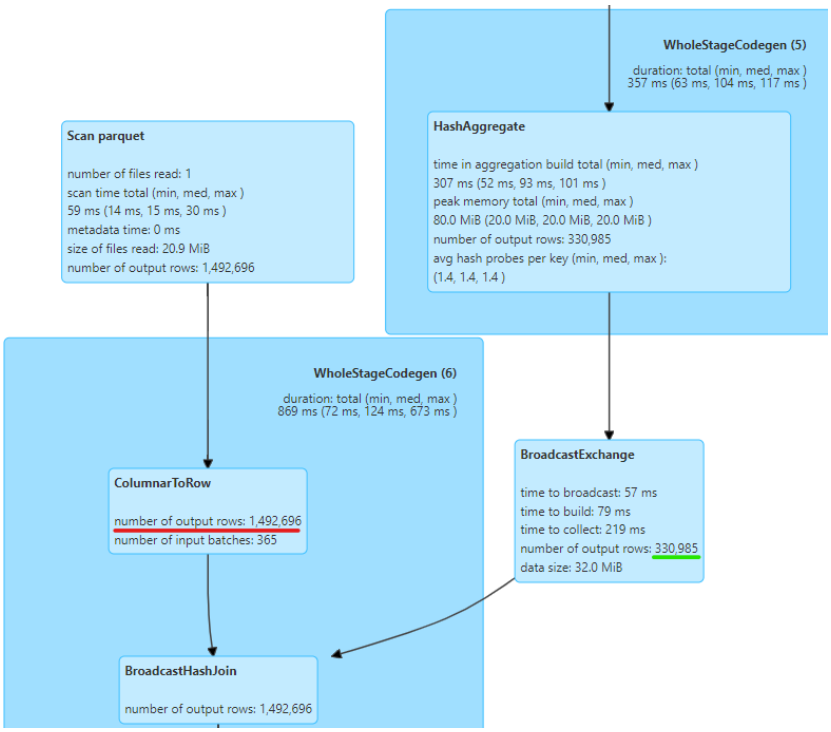


Figura 3.2: Tamanhos das duas tabelas da operação de *join*

Comparando com uma versão (já com a melhoria da repartição) sem o *broadcast hint*, podemos reparar que houve uma pequena melhoria nos tempos médios.

Limite inferior (Argumento)	**Tempo (s)**
1 month	2.366
3 months	1.698
6 months	1.55
1 year	1.667
2 year	2.431

Tabela 3.2: Tempos sem a *broadcast hint*

Limite inferior (Argumento)	**Tempo (s)**
1 month	1.241
3 months	1.131
6 months	1.337
1 year	1.438
2 year	1.838

Tabela 3.3: Tempos com a *broadcast hint*

Estas técnicas de particionamento e a utilização de um *broadcast hint* em conjunto, levaram a uma melhoria final de 1.741 segundos para 1.32 segundos com o limite inferior definido a '6 months'.

Limite inferior de *creationdate* (argumento)	Tempo médio(s)
1 month	1.143
3 months	1.086
6 months	1.32
1 year	1.411
2 years	1.88

Tabela 3.4: Tempos de execução do Workload 1 com partitionBy('year')

3.2 Interrogação analítica 2

Para a interrogação analítica 2, foi feito inicialmente o teste de desempenho para a versão base implementada segundo a versão final da query 2 em PostgreSQL.

Foram alterados os parâmetros "interval" e "bucketInterval" para perceber possíveis variações de tempo de execução, mas o resultado manteve-se bastante semelhante em todos os testes.

Limite inferior de *creationdate* (argumento)	Intervalo de cada *bucket* (argumento)	Tempo médio(s)
1 years	5000	1.953
1 years	1000	1.927
3 years	5000	2.067
3 years	1000	2.149
5 years	5000	1.962
5 years	10000	1.315
10 years	5000	1.204
10 years	10000	1.04

Tabela 3.5: Resultados Base Q2 Spark

Para gerar os ficheiros parquet, começamos por gerá-los a partir das CTE's "year_range" e "max_reputation_per_year" cujos valores são imutáveis para a *snapshot* dos dados utilizados.

```
1  year_range = spark.range(2008, int(spark.sql("SELECT year(CURRENT_DATE)").collect()[0][0]
       + 1), 1).toDF("year")
2    max_reputation_per_year = users \
3                              .withColumn("year", expr("year(creationdate)")) \
4                              .groupBy("year") \
5                              .agg({"reputation": "max"}) \
```

```
6                                   .withColumnRenamed("max(reputation)", "max_rep")
```

De seguida, para o resultado intermédio "u" da query 2 que envolve joins entre três tabelas, foi obtido um resultado parcial sem filtrar pela coluna "creationdate" da tabela votes e adicionando-a ao resultado. Desta forma, é possível obter o máximo de pré-computação para este resultado parcial da query.

```
1  accepted_answers = (
2      votes_selected.join(votesTypes_selected, votes_selected["votetypeid"] ==
           votesTypes_selected["id"])
3          .select("postid", "votes_creationdate")
4  )
5
6  filtered_answers = (
7      answers_selected.join(accepted_answers, answers["id"] == accepted_answers["postid"])
8          .select("owneruserid","votes_creationdate")
9  )
10
11 u = (
12     users_selected.join(filtered_answers, users_selected["id"] ==
           filtered_answers["owneruserid"])
13         .select(users_selected["id"], users_selected["creationdate"],
               users_selected["reputation"], filtered_answers["votes_creationdate"])
14 )
```

Após gerar os ficheiros parquet com resultados parciais e com o máximo de pré-computação, procedemos à realização da query onde poderia ser alterado o "interval" e o "bucketInterval". Para implementar isto, são aplicados os filtros com os parâmetros utilizados e por fim, realizar o resto de computação da query.

Com esta estratégia, é possível obter a resposta à query em entre 1 e 2 segundos (após correr a query uma vez - "warm-up").

Para otimizar o desempenho das interrogações pensamos em ordenar o resultado parcial "u" pela coluna "creationdate" e fazer uma repartição pela mesma já que é o maior resultado parcial e é filtrado posteriormente segundo os valores dessa coluna.

```
1  # versão com ord
2
3  u_ord = u.orderBy('votes_creationdate')
4  u_ord.write.parquet(f'{Q2_PATH}u_ord')
```

```
5
6  # versão com ord e part
7
8  u_ord_part = u_ord.repartitionByRange(col('votes_creationdate'))
9  u_ord_part.write.parquet(f'{Q2_PATH}u_ord_part')
```

Fizemos a comparação dos tempos de execução bem como análise do plano de execução da query pelo Spark UI e apercebemo-nos de que não existia qualquer benefício em termos de tempo de execução com a estratégia de ordenação, uma vez que o tamanho dos ficheiros aumentava com as operações efetuadas e as interrogações a ficheiros ficavam mais lentas, o que atrasava a fase inicial da execução da query.

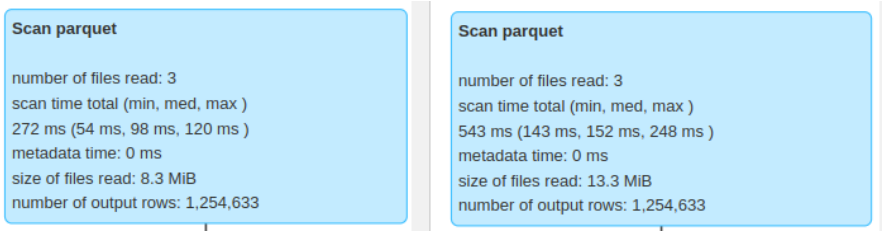


Figura 3.3: Comparação do scan inicial da Query 2

Limite inferior de *creationdate* (argumento)	Intervalo de cada *bucket* (argumento)	Tempo médio(s)
1 years	5000	1.967
1 years	1000	2.006
3 years	5000	2.320
3 years	1000	1.998
5 years	5000	2.571
5 years	10000	2.419
10 years	5000	2.599
10 years	10000	2.541

Tabela 3.6: Resultados Finais Q2 Spark

3.3 Interrogação analítica 3

Para fazer uma comparação de desempenho, fizemos uma tradução da nossa versão final em SQL para Spark, tendo resultado em tempos de execução na ordem dos 20 segundos.

Limite inferior de *count(*)* (argumento)	Tempo médio(s)
10	19.077
30	18.284
50	17.618

Tabela 3.7: Resultados Base Q3 Spark

Tendo em conta que estamos a trabalhar com um *snapshot* da base de dados, foi testada uma versão da query 3 que coloca o resultado de grande parte da computação da query em disco e a query passa apenas a calcular uma operação de filtragem `WHERE count > 'limite_inferior'` relativa ao argumento que a query recebe. Para obter esta versão da query 3, recorremos a uma das suas versões apresentadas na secção da carga transacional no Postgres (ver versão da query aqui).

Em primeiro lugar, a CTE "FilteredTags" deixa de filtrar os "tagid" pelo "`COUNT(*)`" e passa apenas a armazenar a sua contagem, isto é, o valor de "`COUNT(*)`". Para além disso, retiramos da parte principal da query o "`LEFT JOIN`" com a tabela "tags" e colocamo-lo também na CTE "FilteredTags". Esta deslocação do join foi feita, pois serve apenas para a obtenção do nome das tags e, desta forma, pode ser colocado na CTE para obter os nomes das tags antes de se executar a query principal.

Em seguida, podemos verificar que a query principal efectua um "JOIN" entre as CTE's "FilteredTags" e "TagQuestionCounts", sendo o resultado agrupado pelo nome da tag. Essa parte da query principal passou para uma nova CTE designada "mv_q3" para ficar isolada da query principal. Esta nova CTE contém os nomes das tags, a média arredondada e a respectiva contagem.

No final, a query principal passa a utilizar a nova CTE (mv_q3) filtrando-a apenas pela coluna "count", em que o filtro é o argumento da query. Como resultado de todas estas transformações, a query 3 passa a ter este formato:

```
1  WITH TagQuestionCounts AS (
2      SELECT qt.tagid, qt.questionid, COUNT(*) AS total
3      FROM questionstags qt
4      LEFT JOIN answers a ON a.parentid = qt.questionid
5      GROUP BY qt.tagid, qt.questionid
6  ),
7  FilteredTags AS (
8      SELECT tagid, tagname, count(*) as tag_count -- obtenção do tagname e da contagem
           (count(*))
9      FROM TagQuestionCounts tqc
10     LEFT JOIN tags t ON t.id = tqc.tagid -- left join da query principal deslocado
11     GROUP BY tagid, tagname
```

```
), mv_q3 as ( -- nova CTE com a computação da query principal filtrada posteriormente pelo
     argumento
    SELECT ft.tagname, ROUND(AVG(tqc.total), 3), ft.tag_count as count
    FROM TagQuestionCounts tqc
    JOIN FilteredTags ft ON ft.tagid = tqc.tagid
    GROUP BY ft.tagname, ft.tag_count
)
-- Query principal
SELECT *
FROM mv_q3
WHERE count > 10
```

Traduzindo esta versão do SQL para a respetiva versão Spark, exportamos os dados resultantes da tabela `mv_q3` da query SQL para um ficheiro no formato Parquet, e a função da query 3 consiste apenas no seguinte código:

```
def q3_mv(mat_view: DataFrame, inferiorLimit: IntegerType = 10):
    result = mat_view.filter(col("count") > inferiorLimit)
    return result.collect()
```

Uma vez que a parte analítica da query apenas se concentra em filtrar a tabela com base em valores da coluna "count", pensamos em fazer uma ordenação prévia que poderia ajudar a reduzir a quantidade de dados lidos para memória, através de *data pruning* dos blocos que não se encaixam na condição do filtro. No entanto recorremos à análise da tarefa no Spark UI e foi possível verificar que esta ordenação não trazia nenhuma melhoria significativa, uma vez que o tamanho dos dados inicial não é significativo para que o *data pruning* faça efeito. Na seguinte imagem pode-se ver à esquerda o plano da versão sem ordenação e à direita, o plano da versão com ordenação.

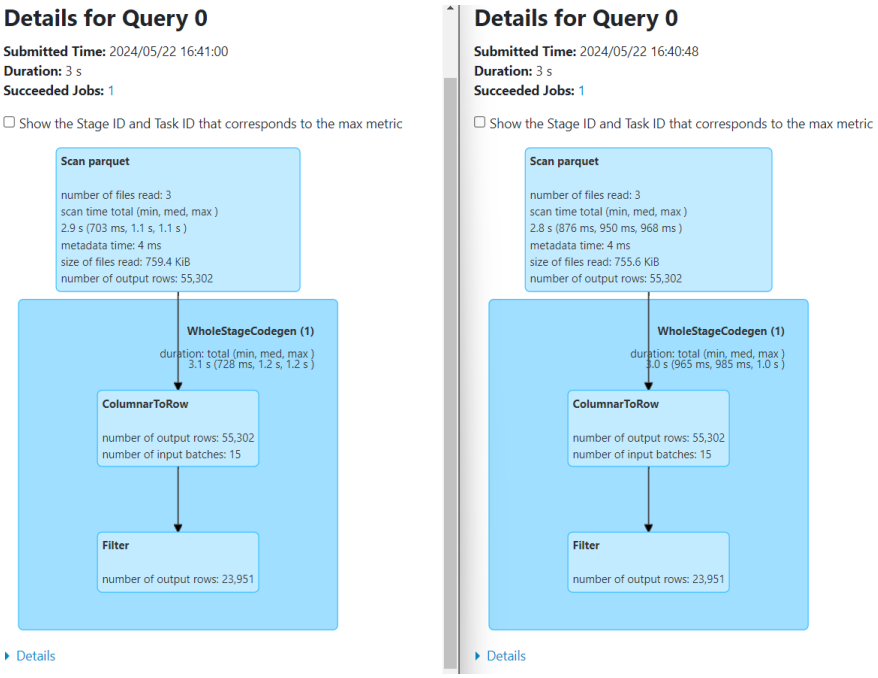


Figura 3.4: Comparação dos tempos de execução

Com esta versão que tira grande partido da pré-computação de uma boa parte da query, obtiveram-se os seguintes resultados médios ao testar a query para diferentes valores do argumento do limite inferior de *count* (10, 30, 50, 100):

Limite inferior de *count(*)* (argumento)	Tempo médio(s)
10	0.263
30	0.228
50	0.249
100	0.225

Tabela 3.8: Resultados Finais Q3 Spark

3.4 Interrogação analítica 4

Dado que a versão final da query 4 a que chegamos na análise da query no Postgres envolveu a criação de uma *materialized view*, partimos de uma versão que os dados da tabela `badges` já foram pré-filtrados, o que é equivalente à pré-computação que é feita pela *materialized view* no Postgres.

Esta estratégia apresenta os seguintes resultados:

Tamanho de cada bucket (argumento)	Tempo médio(s)
1 minute	28.387
10 minutes	13.941
30 minutes	7.063
2 hours	3.718
6 hours	3.056

Tabela 3.9: Resultados inicias da Q4 Spark

Como a query irá executar um agrupamento relativo a uma coluna que se baseia na coluna "date", decidimos testar uma versão em que o ficheiro parquet que irá ser lido está particionado (*RepartitionByRange*) pela coluna "date", o que pode reduzir a necessidade de ler outras partições para certas chaves de agrupamento. Testou-se com vários números de partições e o melhor número foi com 9 partições.

Na seguinte imagem, estão representados dois planos relativos à execução da query 4 com o argumento da query definido como "1 minute" em que à esquerda está a versão sem o particionamento e à direita com o particionamento.

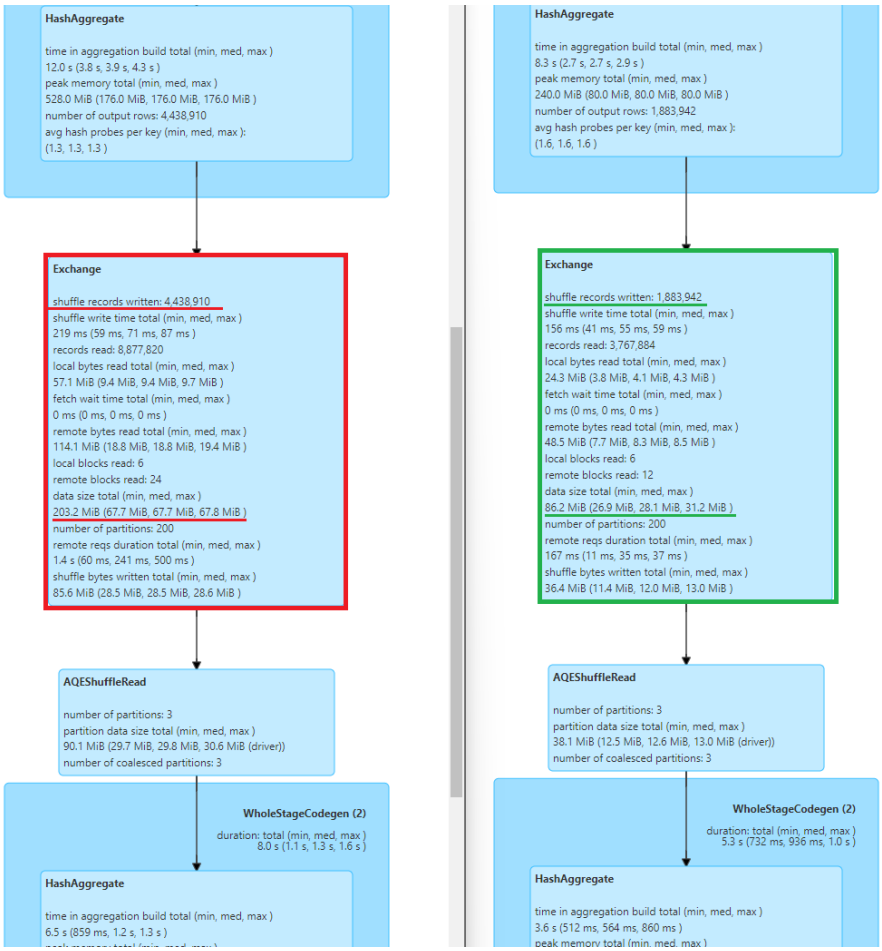


Figura 3.5: Comparação das estatísticas de *shuffle*

Na comparação, podemos verificar que a quantidade de dados envolvidos no *shuffle* é significativamente menor, como indicam os números de *shuffle records written* e *data size total*. Num cenário em que os diferentes nós do cluster estão distribuídos e conectados através da rede, a operação de *shuffling*, representada pelo operador *exchange*, é bastante custosa. Embora os nós do nosso cluster Spark estejam a correr numa mesma máquina e a comunicação entre nós não dependa da rede, o *shuffling* ainda tem um impacto significativo na performance, uma vez que envolve operações de leitura e escrita no disco.

De seguida apresentam-se os resultados finais dos tempos de execução médios da query 4 em Spark, com estas estratégias implementadas.

Tamanho de cada bucket (argumento)	Tempo médio (s)
1 minute	25.553
10 minutes	11.488
30 minutes	4.791
2 hours	2.187
6 hours	1.536

Tabela 3.10: Resultados Finais Q4 Spark

3.5 Otimizações de configuração Spark

Após otimizar as estratégias de execução para cada uma das interrogações analíticas no Spark, testamos a realização de modificações nas configurações do cluster Spark, de maneira a alcançar a melhor otimização possível dos tempos de execução das interrogações.

Primeiramente testamos configurar o cluster com diferentes números de *workers* e avaliar os seus resultados. De notar, que apresentamos os valores de tempo médio para a execução das várias queries, mas de modo a simplificar a tabela, os argumentos passados a cada query são os valores padrão.

Nº Workers	**Query 1**	**Query 2**	**Query 3**	**Query 4**
1	2.366	1.471	0.178	30.793
2	2.136	1.461	0.171	27.227
3	2.333	1.447	0.171	25.823
4	2.485	1.894	0.216	27.047
5	2.425	1.987	0.226	27.229
6	2.305	2.127	0.236	27.811

Tabela 3.11: Resultados (em segundos) para diferentes números de *workers*.

Apesar dos resultados não variarem imenso, decidimos considerar o melhor número de *workers* como 3 *workers*. De seguida apresentamos um conjunto de diferentes configurações com base numa configuração com 3 *workers*.

Configurações	Query 1	Query 2	Query 3	Query 4
driver.memory=8gb	2.333	1.447	0.171	25.823
driver.memory=12gb	2.066	1.334	0.168	25.699
driver.memory=16gb	2.32	1.497	0.175	25.812
shuffle.partitions=8	2.047	1.293	0.163	25.294
shuffle.partitions=16	2.113	1.4	0.16	25.636
shuffle.partitions=32	2.02	1.335	0.178	25.084
shuffle.partitions=64	2.024	1.419	0.181	25.402
shuffle.partitions=128	2.028	1.395	0.168	26.328
shuffle.partitions=200	2.166	1.524	0.166	26.014

Tabela 3.12: Resultados (em segundos) para diferentes opções de configuração

Tal como nos diferentes números de workers, não se notam diferenças significativas consoante os diferentes valores dos parâmetros de configuração. No entanto, tendo em conta os resultados, decidimos considerar uma configuração final com 3 *workers*, *driver.memory*=12 gb, com *shuffle.partitions*=8 e que apresentam os seguintes resultados de tempo.

Query 1	Query 2	Query 3	Query 4
2.014	1.28	0.154	25.91

Tabela 3.13: Tempos de execução finais em segundos

4 Carga transacional

Esta secção consiste na explicação das estratégias usadas para melhorar o desempenho da carga transacional. As estratégias exploradas durante este processo foram o uso de redundância na forma de índices, alterações no código Java/SQL do programa e a modificação dos parâmetros de configuração *Postgres*.

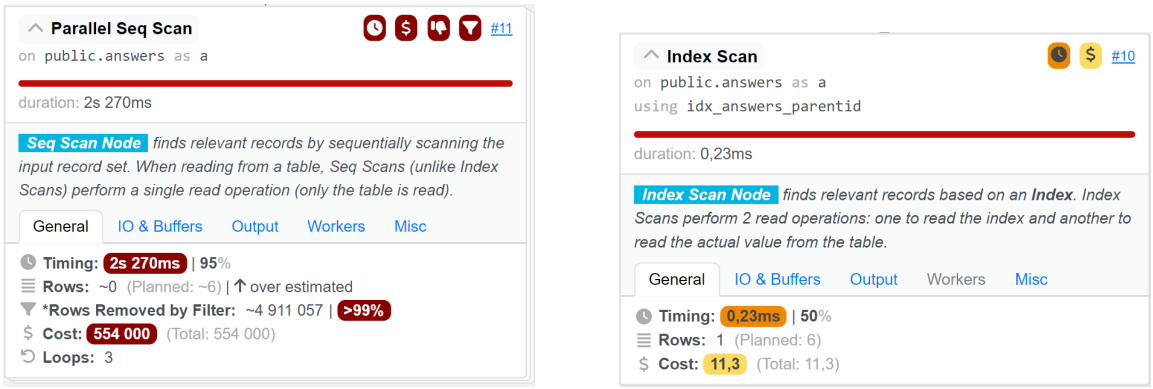
Inicialmente exploramos o código do *benchmark* para perceber se o poderíamos alterar de forma a melhorar o seu desempenho. De seguida exploramos o uso de redundância através de índices, fazendo uma análise às *queries* transacionais. Para terminar, exploramos os diversos parâmetros do *Postgres* para otimizar ainda mais o desempenho da carga transacional.

4.1 Otimização através de índices

Com a análise das *queries* presentes nos *prepareStatements*, o grupo identificou algumas otimizações possíveis através de índices.

4.1.1 GetQuestionInfo

Primeiramente analisamos o plano da query getQuestionInfo que demonstra que o *bootleneck* desta query está na execução da pesquisa na coluna *parentid* da tabela *answers*. Por isso criamos um índice que permite o uso de um "Index Scan" invés de um "Parallel Seq Scan" que melhora significativamente o tempo de pesquisa na coluna *parentid*.



(a) Sem Índice

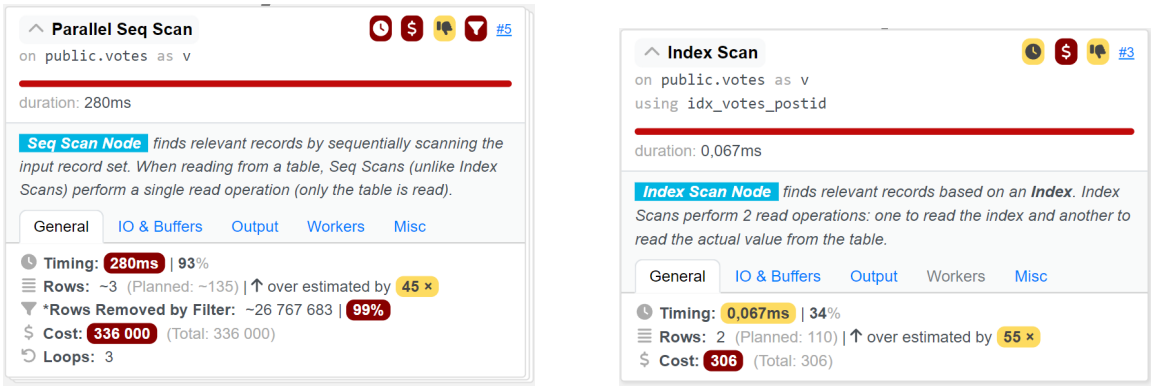
(b) Com Índice

Figura 4.1: Comparação entre o plano da query getQuestionInfo com e sem índice

```
1 CREATE INDEX idx_answers_parentid ON answers (parentid);
```

4.1.2 GetPostPoints

De seguida passamos para a análise do plano da query getPostPoints e percebemos que a maior demora encontra-se na pesquisa de votos na coluna *postid*. Por forma a otimizar essa pesquisa criamos um índice que permite que essa pesquisa seja feira com um "Index Scan" invés de um "Parallel Seq Scan" o que melhora bastante o seu tempo de execução.



(a) Sem Índice

(b) Com Índice

Figura 4.2: Comparação entre o plano da query getPostPoints com e sem índice

Ainda nesta query criamos um novo índice na coluna *votetypeid* da tabela votes por forma a otimizar a operação de *join* feita nesta query.

```
1 CREATE INDEX idx_votes_postid ON votes (postid);
2 CREATE INDEX idx_votes_votetypeid ON votes (votetypeid);
```

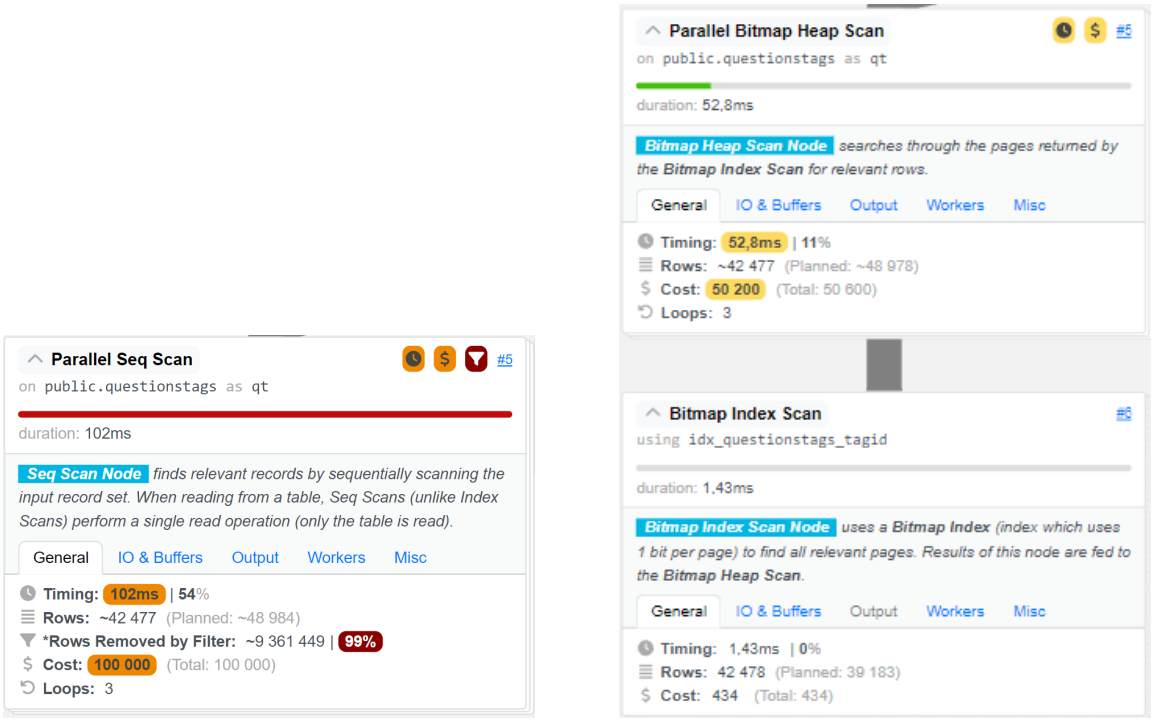
4.1.3 Search

Para a query Search visto que é uma query de pesquisa textual criamos um índice GIN (Generalized Inverted Index) que é adequado para pesquisas de textual no PostgresSQL. Ao utilizar a expressão (to_tsvector('english', title)) o sistema converte o conteúdo da coluna *title* em um vetor de texto processado e aplica técnicas de *stemming* conforme as regras da língua inglesa. Dessa forma a query ao fazer a consulta com (to_tsquery('english', 'java')) pode utilizar esse indice para acelerar o processo de identificação de correspondências relevantes.

```
1 CREATE INDEX idx_questions_title_gin ON questions USING GIN(to_tsvector('english', title));
```

4.1.4 LatestQuestionByTag

Para terminar fizemos uma análise no plano da query LatestQuestionByTag e percebemos que o a maior demora estava na leitura da tabela *questiontags*. Para otimizar esta leitura criamos um índice na coluna *tagid* de forma a melhorar essa pesquisa.



(a) Sem Índice

(b) Com Índice

Figura 4.3: Comparação entre o plano da query latestQuestionsByTag com e sem índice

De seguida exploramos o uso de um índice na tabela *questions* na coluna *creationdate* para otimizar a operação de ordenação através desse valor. Este índice foi capaz de otimizar ainda mais esta query melhorando o tempo de execução de 189 ms para 83,3 ms.

```
1 CREATE INDEX idx_questionstags_tagid ON questionstags (tagid);
2 CREATE INDEX idx_questions_creationdate ON questions (creationdate);
```

4.1.5 Resultados obtidos

Foi realizada uma comparação inicial de como os índices utilizados para melhorar o desempenho das queries analíticas influenciavam os tempos das queries transacionais.

Métrica	**Sem índices**	**Apenas índices Analíticas**	**Apenas índices Transacionais**	**Com Índices das Analíticas e Transacionais**
NewQuestion (ms)	5.6065	3.8855	4.3899	4.5383
NewAnswer (ms)	7.0716	6.3281	4.7175	5.0101

Métrica	**Sem índices**	**Apenas índices Analíticas**	**Apenas índices Transacionais**	**Com Índices das Analíticas e Transacionais**
Vote (ms)	4.0492	4.0426	3.4672	3.7902
QuestionInfo (ms)	4072.8008	3988.0834	2.0582	1.9292
PostPoints (ms)	5487.3535	7.2357	0.5772	2.7026
UserProfile (ms)	6.5390	4.9441	2.5794	1.7865
Search (ms)	1471.1648	1132.9358	15.4340	15.2653
LatestByTag (ms)	2347.2005	1151.4900	108.0811	127.8355
Throughput (txn/s)	12.4389	19.6	1073.0111	934.0556
Response Time (ms)	1203.4697	751.4883	14.2465	16.3006
Abort Rate (%)	0.0	0.0	0.0	0.0

De forma a perceber quais eram as queries que estavam a demorar mais tempo, foi utilizado o módulo adicional *pg_stat_statements*. Dessa forma obtiveram-se os seguintes resultados, com a seguinte query:

```
1 SELECT query, calls, total_exec_time, rows, 100.0 * shared_blks_hit /
2              nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
3          FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 5;
```

Query	Calls	Total Execution Time (ms)	Rows	Hit Percent (%)
select id, title from questions where to_tsvector($2, title) @@ to_tsquery($3, $1) limit $4	2137	1400938.08	53425	21.71
select id, title from questions q join questionstags qt on qt.questionid = q.id where qt.tagid = $1 order by q.creationdate desc limit $2	4171	658799.80	104275	74.40
select id from questions order by random() limit $1	16	613479.43	1600000	97.72
select id from answers order by random() limit $1	16	133056.12	1600000	97.63
select tagid from questionstags group by 1 having count(*) > $1 order by random() limit $2	16	88512.48	16560	35.52

Tabela 4.2: Performance Statistics

Os resultados mostram as interrogações mais demoradas e fornecem detalhes sobre frequência de execução, tempo total gasto, linhas processadas e eficiência do cache. Alto *total_exec_time* indica interrogações que podem beneficiar mais com a otimização. *Hit_percent* baixo sugere oportunidades para melhorar o desempenho aumentando o uso do cache, talvez ajustando configurações relacionadas à memória ou adicionando índices.

Assim, segundo os resultados que obtivemos, retiramos que:

- Existe um tempo de execução alto para interrogações executadas com frequência, exemplo das queries 1 e 2, em que a dois acaba por ser melhor porque tem uma percentagem de hit maior.
- Existe uma taxa de hit na cache para algumas queries, caso da terceira e quarta queries, em que têm *Hit-Percent* superior a 97%, indicando uso eficiente de memória para essas interrogações.
- Podem existir problemas de indexação já que a percentagem de hits na primeira interrogação sugere que pode beneficiar de uma indexação adicional ou de índices otimizados.
- Pode existir falta de otimização no design das queries já que, apesar do *hit-percent* da query dois ser alto, ainda leva um tempo considerável, indicando que, embora a cache ajude, uma otimização adicional pode ser possível.

Considerando estes resultados, primeiro foram verificados novamente os índices, sendo criado um novo índice composto para a diminuição do tempo de execução da query 2: CREATE INDEX idx_ques-

tions_creationdate_id ON questions (creationdate DESC, id);

Dessa forma obteve-se o segundo conjunto de resultados:

Query	**Calls**	**Total Execution Time (ms)**	**Rows**	**Hit Percent (%)**
select id, title from questions q join questionstags qt on qt.questionid = q.id where qt.tagid = $1 order by q.creationdate desc limit $2	37684	4831308.93	942100	74.68
select id, title from questions where to_tsvector($2, title) @@ to_tsquery($3, $1) limit $4	18813	1617837.79	470325	35.11
select id from questions order by random() limit $1	48	821287.23	4800000	92.20
select id from answers order by random() limit $1	48	396709.77	4800000	97.36
select tagid from questionstags group by 1 having count(*) > $1 order by random() limit $2	48	266889.59	49680	19.57

Tabela 4.3: Estatísticas de Desempenho

O segundo conjunto de resultados demonstra uma diminuição significativa no tempo total de execução para determinadas interrogações, sugerindo uma capacidade aprimorada para o processamento de interrogações mais rápido, potencialmente resultando num aumento de *throughput*. Embora haja um aumento no número de chamadas para algumas interrogações, indicativo de maior procura ou concorrência, isso poderia significar uma melhoria na eficiência e, consequentemente, um maior *throughput* se o sistema gerir eficazmente a carga de trabalho aumentada. Apesar de uma ligeira redução na percentagem de *hits* em cache, o equilíbrio geral entre tempo de execução e eficiência em cache no segundo conjunto de resultados implica um ambiente favorável para alcançar um maior *throughput*, especialmente sob cargas pesadas. Portanto, considerando estes fatores, o segundo conjunto de resultados é recomendado para maximizar o *throughput*, pois as melhorias significativas no tempo total de execução indicam uma capacidade aprimorada para processar um maior volume de transações rapidamente, tornando-o mais adequado para cenários que priorizam a capacidade de processamento de transações em grande escala.

4.2 Otimização através de parâmetros

4.2.1 shared_buffers

Verificando o parâmetro de configuração de memória, obtemos o seguinte valor:

```
stack=# SHOW shared_buffers ;
----------------
 128MB
(1 row)
```

Executando o comando *"free -m"* podemos verificar que existe uma grande quantidade de memória disponível, com aproximadamente 800MB usadas e 15GB livres.

Category	Total (MB)	Used (MB)	Free (MB)	Shared (MB)	Buffer/Cache (MB)
Mem	15980	708	1864	2113	13407
Swap	0	0	0	0	0

Tabela 4.4: Memory Information

Sabendo que é aconselhável que o valor seja entre 20% a 25% da memória total livre, foram realizados testes para se perceber qual era o melhor valor para este parâmetro. Assim foram realizados testes em que se começou por aumentar a memória dos *shared_buffers* até 4GB: 256MB, 1GB, 2GB e 4GB.

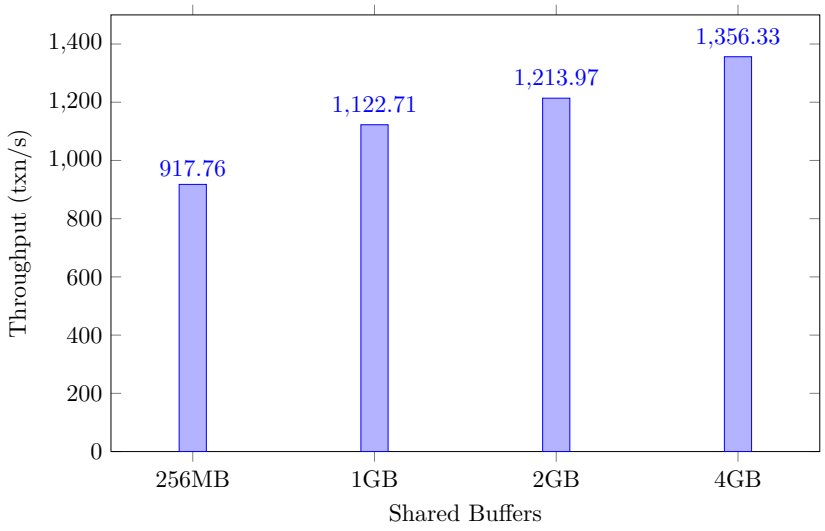


Figura 4.4: Throughput Variation with Shared Buffers

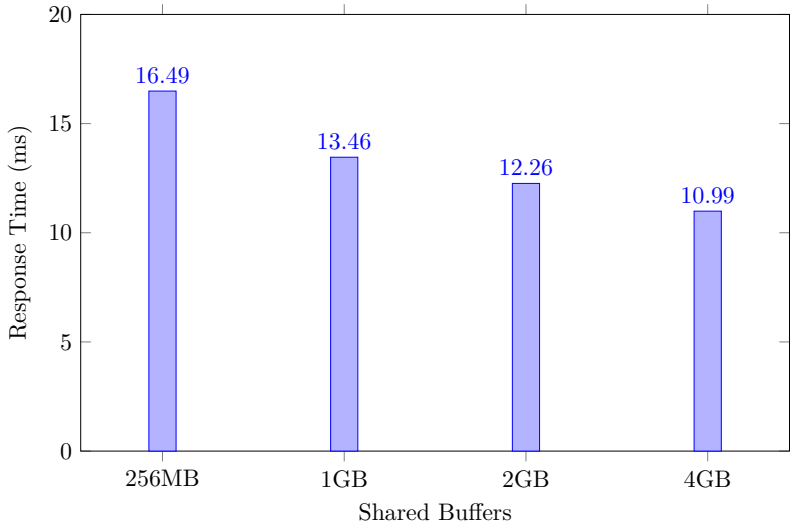


Figura 4.5: Response Time Variation with Shared Buffers

Ao analisar as métricas de desempenho e estatísticas de execução de interrogações para cada configuração de *buffers* partilhados, observa-se que o tempo de resposta geral é relativamente alto na configuração padrão de 256MB, especialmente para interrogações como *LatestByTag* e *Search*, com tempos de resposta entre 84 ms e 129 ms. Isso sugere dificuldades do sistema em manter dados frequentemente acedidos na memória devido aos *buffers* partilhados limitados. Através de configurações de 1GB, 2GB e 4GB, houve melhorias progressivas no tempo de resposta, *throughput* e execução de interrogações, com o maior destaque na configuração de 4GB, que apresentou os melhores resultados em termos de tempo de resposta e *throughput*. Essa análise indica que aumentar o tamanho dos *buffers* partilhados pode proporcionar ganhos significativos de desempenho, especialmente para interrogações com tempos de execução mais altos, como a *Search* e *LatestByTag*.

4.2.2 work_mem

Verificando o parâmetro de configuração de memória, obtemos o seguinte valor:

```
1 stack=# SHOW work_mem ;
2 ----------------
3  4MB
4 (1 row)
```

Observando com **EXPLAIN ANALYSE** as diferentes queries transacionais, obtiveram-se os valores presentes na tabela a seguir 4.5.

Interrogação	Operação	Tempo de Planeamento (ms)	Tempo de Execução (ms)
Insert questions	Insert	3.296	71.976
Insert questionstags	Insert	0.042	9.831
Insert answers	Insert	0.034	33.734
Update questions	Update	8.661	1.059
Insert votes	Insert	0.036	24.260
QuestionInfo	Nested Loop Left Join	21.817	0.137
PostPoints	Hash Join	1.188	0.643
UserProfile	Index Scan	3.762	3.271
UserBadges	Index Only Scan	2.044	5.191
Search	Bitmap Heap Scan	0.419	37.027
LatestByTag	Nested Loop	4.245	5055.761

Tabela 4.5: Resultados do EXPLAIN ANALYZE

As principais conclusões são:

- **Inserções e Atualizações Simples:** Estas operações geralmente têm tempos de execução moderados. Embora não mostrem problemas significativos de desempenho relacionados à memória, ajustar o `work_mem` é pouco provável que traga melhorias substanciais.
- **Joins Complexos e Agregações:** Interrogações envolvendo joins complexos, como a seleção de informações da pergunta, geralmente têm bom desempenho. No entanto, a query *LatestByTag*, que envolve nested loops e ordenação, apresenta um tempo de execução muito elevado (5055.761 ms). Esta interrogação pode beneficiar significativamente do aumento do `work_mem`, pois permite manter uma maior parte do conjunto de trabalho na memória, reduzindo o I/O de disco.
- **Pesquisa de Texto Completo:** A query *search* mostra tempos de execução moderados (37.027 ms). Aumentar o `work_mem` pode ajudar a melhorar o desempenho ao permitir que maiores porções do índice invertido sejam mantidas na memória.
- **Index Scans:** Interrogações de seleção que utilizam index scans, como as querys *UserProfile* e *UserBadges*, são eficientes. Estas interrogações são menos propensas a beneficiar do aumento do `work_mem` uma vez que já estão otimizadas para o uso de índices.

Com base nestas observações, decidimos aumentar o `work_mem` de forma incremental, começando por 16MB até valores potencialmente mais elevados, e reavaliar as melhorias de desempenho utilizando `EXPLAIN ANALYZE`.

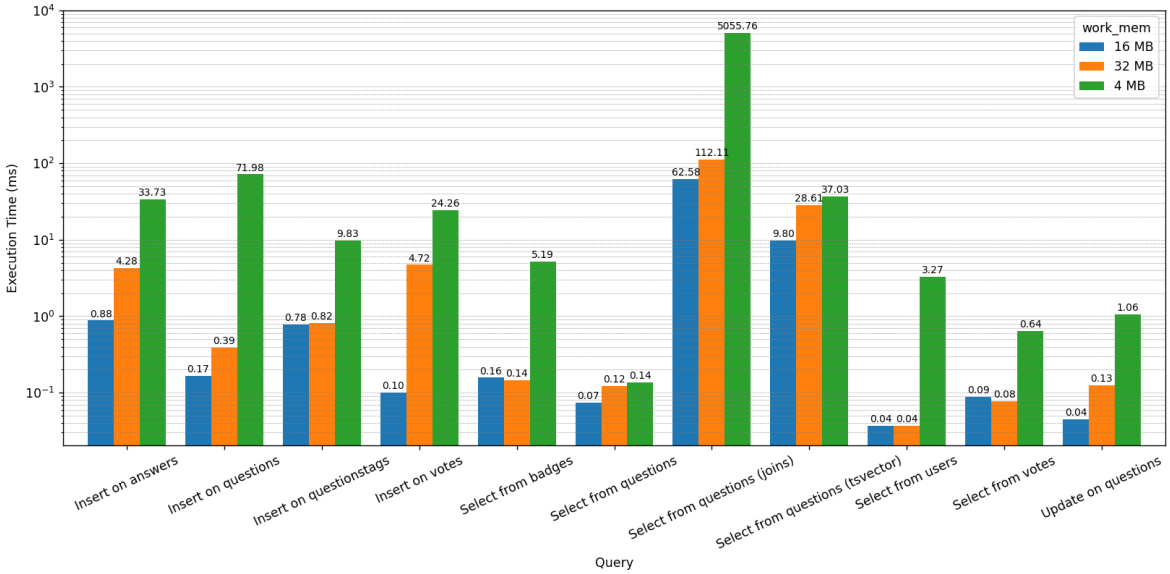


Figura 4.6: Impacto do work_mem nas diversas queries da componente transacional (gráfico em escala logarítmica)

A tabela com os valores pormenorizados encontra-se no **Anexo D**.

Consoante os resultados podemos efetuar a seguinte análise:

- **work_mem de 4MB:** A maioria das operações tem tempos de execução razoáveis. No entanto, as interrogações envolvendo múltiplas junções (por exemplo, *questions*, *questiontags*, *answers* e *questionsInfo*) são mais lentas devido ao processamento de grandes volumes de dados. Aumentar o work_mem poderia melhorar o desempenho para interrogações intensivas em ordenação ou hash.
- **work_mem de 16MB:** Apresenta melhorias gerais em comparação com 4MB. Inserções, atualizações e seleções simples são significativamente mais rápidas. Interrogações complexas, especialmente aquelas com múltiplas junções, exibem melhorias significativas de desempenho.
- **work_mem de 32MB:** Semelhante ao 16MB, mas com um desempenho ligeiramente pior na maioria dos casos.

Assim podemos concluir que aumentar o work_mem de 4MB para 16MB melhora significativamente o desempenho das interrogações. No entanto, aumentá-lo ainda mais para 32MB não traz benefícios para estas interrogações.

4.2.3 effective_cache_size

A query seguinte indica as taxas de *hit-rate* da cache. Uma taxa alta significa um uso eficaz de memória e um tamanho de cache bem configurado.

```
SELECT
    datname,
    blks_hit * 100.0 / (blks_hit + blks_read) AS buffer_cache_hit_ratio
FROM
    pg_stat_database
WHERE
    (blks_hit + blks_read) > 0;
  datname | buffer_cache_hit_ratio
-----------+------------------------
 stack    |   87.8317996252115783
```

Assim obtemos o valor de aproximadamente 88% de hit rate, o que já é bom, mas pode ser melhorado. Para base de dados com grande desempenho é desejável taxas de *hit-rate* próximas de 99%.

Em termos de tamanho da cache, na query *'latestQuestionsByTag'* obtemos um custo estimado relativamente grande e o atual custo também é considerável (Execution Time: 62.575 ms), o que indica que pode beneficiar de uma melhor utilização da cache. Embora o *explain analyse* da query não indique explicitamente que se deva alterar o parâmetro de configuração *'effective_cache_size'*, o custo elevado de acesso a disco (cost=0.86..3002719.04) indica que talvez ajude.

Assim, tentamos alterar o parâmetro de configuração para diferentes valores, dos quais resultou o seguinte:

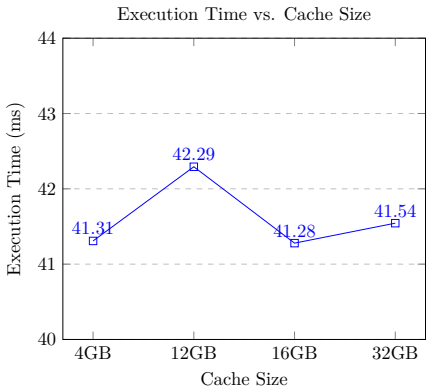


Figura 4.7: Execution Time for Different Cache Sizes

Com base nestes resultados, parece que a alteração do parâmetro *effective_cache_size* não teve um impacto significativo no desempenho da interrogação testada. Os tempos de execução mantiveram-se dentro de uma faixa muito estreita em todas as dimensões de cache testadas.

4.2.4 Paralelismo

A seguir realizamos testes para melhorar o paralelismo, focando na query *'latestQuestionsByTag'*. Para tal, começamos por analisar o parâmetro de configuração *'max_parallel_workers_per_gather'*, obtendo os seguintes resultados:

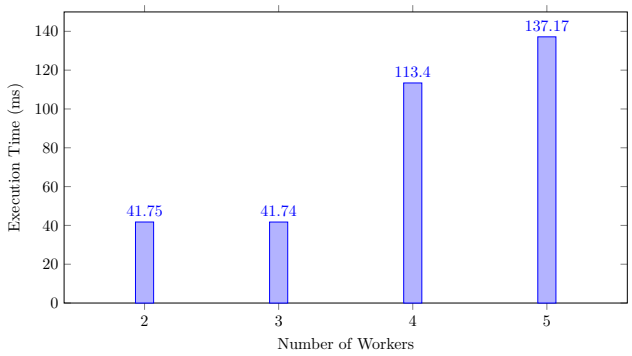


Figura 4.8: Execution Time of Queries with Different Numbers of Workers

Com estes resultados, podemos observar que, à medida que o número de *workers* aumenta de 2 para 5, o tempo de execução também aumenta. Isso sugere que os custos adicionais associados à coordenação dos trabalhadores em paralelo e à disputa por recursos do sistema podem superar os benefícios potenciais da execução paralela para a carga de trabalho e configuração de hardware fornecidas.

Tentamos, portanto, utilizar outro parâmetro de configuração para melhorar o paralelismo, nomeadamente o parâmetro *max_parallel_workers*.

Configuration	**Throughput (txn/s)**	**Response Time (ms)**
4,8	824.06	18.37
4,16	843.59	18.29
4,32	816.11	18.78
2,32	831.81	18.45

Tabela 4.6: Performance Metrics for Different Configurations

Ao observar a tabela, podemos observar que as configurações com *max_parallel_workers*=4 geralmente apresentam maior throughput em comparação com a configuração com *max_parallel_wor-*

kers=2. No entanto, há alguma variação no tempo de resposta entre diferentes configurações, sem uma tendência clara que indique uma vantagem significativa para qualquer configuração específica.

Globalmente, a configuração *max_parallel_workers*=16 destaca-se ligeiramente com o maior *throughput* e um dos menores tempos de resposta. Portanto, com base nesses resultados, *max_parallel_workers_per_gather*=4 e *max_parallel_workers*=16 poderia ser considerada a melhor configuração entre as testadas.

4.2.5 Checkpoints e WAL

A partir dos resultados da benchmark, observamos que as operações de inserção, como aquelas nas tabelas *questions* e *answers*, apresentam tempos de execução relativamente baixos, indicando que escritas individuais não são o principal gargalo. No entanto, operações de escrita frequentes e possivelmente simultâneas podem causar checkpoints frequentes, especialmente com o intervalo padrão de 5 minutos. Além disso, algumas operações de leitura, particularmente *search* e *LatestByTag*, exibem tempos de resposta mais elevados, que podem ser afetados pela contenção de I/O causada por esses checkpoints frequentes.

Posto isto, tentamos alterar os parâmetros que se relacionam com os *checkpoints* e com *write-ahead-logging*, para:

- **tentar melhorar o *throughput* ao suavizar os padrões de I/O:** checkpoints maiores e menos frequentes ajudam a distribuir a carga de I/O de forma mais equilibrada, reduzindo picos que podem interferir no desempenho das consultas;
- **reduzir a sobrecarga de escrita:** aumentando o max_wal_size, o PostgreSQL pode armazenar mais alterações antes de forçar um checkpoint, o que reduz a frequência de explosões de escrita;
- **equilibrar recuperação e desempenho:** embora intervalos de checkpoint mais longos possam aumentar o tempo de recuperação, muitas vezes levam a um melhor desempenho durante operações normais, o que geralmente é uma troca vantajosa para muitas aplicações.

A análise dos tempos de resposta em diferentes funções revela um desempenho consistente e estável para a maioria das operações, com *NewQuestion*, *NewAnswer*, *Vote*, *QuestionInfo*, *PostPoints* e *UserProfile* mantendo tempos de resposta semelhantes em ambos os conjuntos de dados. Notavelmente, a função *Search* apresenta uma melhoria significativa no desempenho no segundo conjunto de dados, reduzindo de aproximadamente 30,25 ms para 16,56 ms, o que sugere esforços de otimização bem-sucedidos.

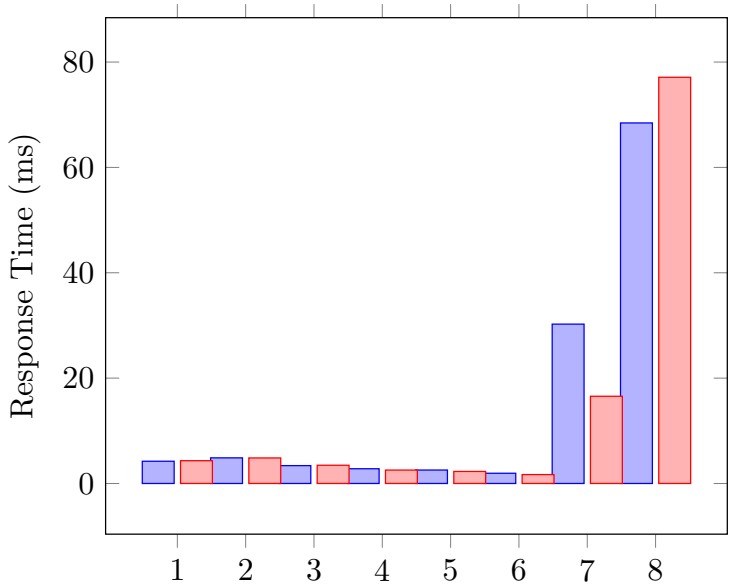


Figura 4.9: Comparison of Response Time per Function

No entanto, a função *LatestByTag* mostra consistentemente os tempos de resposta mais elevados, com valores em torno de 68,44 ms e 77,11 ms, respetivamente, nos dois conjuntos de dados. Verificando com *explain analyse* podemos detetar que na primeira forma havia paralelismo com 4 workers, enquanto que na segunda só havia com 2, sendo portanto, esta uma possível justificação do tempo da query ter piorado.

```
Base:

 Limit (cost=1000.92..1980.45 rows=25 width=67) (actual time=144.912..164.081 rows=25
     loops=1)
   -> Gather Merge (cost=1000.92..1852276.95 rows=47249 width=67) (actual
      time=144.910..164.075 rows=25 loops=1)
        Workers Planned: 4
        Workers Launched: 4
        (...)
 Planning Time: 0.314 ms
 Execution Time: 164.128 ms
(11 rows)


Com novos parâmetros:

```

```
15  Limit (cost=1000.89..2518.90 rows=25 width=67) (actual time=107.264..125.774 rows=25
        loops=1)
16    -> Gather Merge (cost=1000.89..2892394.18 rows=47618 width=67) (actual
         time=107.262..125.770 rows=25 loops=1)
17          Workers Planned: 2
18          Workers Launched: 2
19          (...)
20  Planning Time: 0.745 ms
21  Execution Time: 125.842 ms
```

Em termos de *throughput*, o valor baixou de aproximadamente 1351.18 para aproximadamente 1337.74, o que significa que de forma geral houve uma pioria com a alteração destes parâmetros.

4.2.6 fsync

A seguir, experimentamos desativar o parâmetro de configuração *fsync*. O fsync garante que os dados são efetivamente escritos no disco, assegurando a durabilidade das transações. No entanto, desativar o fsync pode reduzir significativamente os tempos de resposta, melhorando o desempenho geral da base de dados.

Tabela 4.7: Comparison of Results with and without `fsync=off`

Metric	**Original Value**	**With `fsync=off`**
NewQuestion	4.224 ms	1.538 ms
NewAnswer	4.866 ms	2.033 ms
Vote	3.392 ms	0.794 ms
QuestionInfo	2.798 ms	3.160 ms
PostPoints	2.554 ms	3.158 ms
UserProfile	1.943 ms	2.007 ms
Search	30.251 ms	18.036 ms
LatestByTag	68.441 ms	86.306 ms
Throughput (txn/s)	1351.18	1325.5
Response Time (ms)	11.063	11.116

Desativar o parâmetro fsync resultou em melhorias significativas nos tempos de resposta para várias funções, como *NewQuestion*, *NewAnswer*, *Vote* e *Search*. No entanto, também houve um aumento significativo nas funções *QuestionInfo*, *PostPoints*, *UserProfile* e *LatestByTag* e a taxa de transferência teve uma ligeira queda de 1351.18 txn/s para 1325.5 txn/s, o que sugere um pequeno impacto negativo na capacidade de processamento global do sistema.

4.2.7 synchronous_commit

Configurar *synchronous_commit* como *off* altera o comportamento das transações, permitindo que avancem sem esperar que os registos de *Write-Ahead Logging* (WAL) sejam escritos no disco antes de sinalizar sucesso para o cliente. Essa modificação pode aumentar o débito ao minimizar os tempos de espera, potencialmente melhorando o desempenho do sistema. No entanto, tal ajuste introduz uma compensação, embora possa aumentar a eficiência, também aumenta o risco de perda de dados no caso de uma falha do sistema imediatamente após a confirmação de uma transação, pois as alterações podem não ter sido permanentemente armazenadas no disco.

Tabela 4.8: Comparison of Results with and without `synchronous_commit=off`

Metric	**Original Value**	**With `synchronous_commit=off`**
NewQuestion	4.224 ms	1.545 ms
NewAnswer	4.866 ms	1.990 ms
Vote	3.392 ms	0.818 ms
QuestionInfo	2.798 ms	2.997 ms
PostPoints	2.554 ms	2.710 ms
UserProfile	1.943 ms	1.875 ms
Search	30.251 ms	15.797 ms
LatestByTag	68.441 ms	94.993 ms
Throughput (txn/s)	1351.18	1274.572
Response Time (ms)	11.063	11.748

Comparando os resultados, ao desativar o *synchronous_commit*, conseguiu-se uma redução significativa nos tempos de resposta para todas as funções, com exceção da função "*LatestByTag*", que teve um aumento considerável no tempo de resposta. Para além disso, a métrica geral de *throughput* diminuiu um pouco, enquanto o tempo de resposta médio geral aumentou ligeiramente.

Isso sugere que desativar o *synchronous_commit* pode melhorar a eficiência das operações individuais, resultando em tempos de resposta mais rápidos para a maioria das funções. No entanto, pode haver um aumento no tempo de resposta para certas operações, como "*LatestByTag*", que pode ser mais sensível à falta de confirmação síncrona.

4.2.8 wal_compression

Outro parâmetro que tentamos verificar foi *wal_compression*, que comprime os dados do WAL para reduzir a quantidade de I/O de disco, trocando ciclos de CPU por uma utilização reduzida do disco.

Observamos que tanto o tempo de resposta como o throughput aumentam, pelo que a ativação deste parâmetro não ajudou no nosso caso.

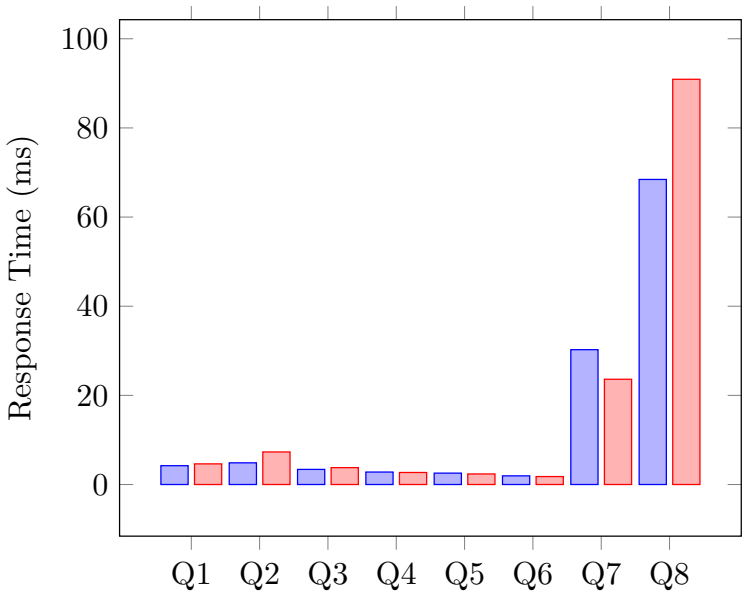


Figura 4.10: Comparison of Response Times with and without `wal_compression`

4.2.9 Conclusão parâmetros de configuração

Concluindo, retiramos que apesar dos diferentes parâmetros de configuração, o melhor *throughput* observado foi quando:

- shared_buffers = 4GB;
- work_mem = 16MB;
- max_parallel_workers_per_gather = 4;
- max_parallel_workers = 16;

5 Conclusão

O desenvolvimento deste relatório documenta o trabalho realizado durante a configuração, otimização e avaliação do *benchmark* baseado num excerto do dataset do StackOverflow. Durante este processo conseguimos consolidar os conceitos lecionados nas aulas da UC de Administração de Bases de Dados e explorar um caso prático da utilização desses conceitos.

A implementação de diversas técnicas de otimização como índices, vistas materializadas e reestruturação de queries permitiu uma melhoria significante do desempenho das interrogações no PostgresSQL e Spark. Para além disso também exploramos as configurações do PostgresSQL por forma a melhorar a eficiência do processamento da carga transacional.

Em suma, o trabalho realizado permitiu-nos adquirir uma melhor compreensão sobre os conceitos de administração e otimização de uma base de dados e como documentar esse processo.

A Script de activação da VM

```
#!/bin/bash

# Var com o nome da vm
VM_NAME="vm-abd"

# Command to get instance status
status=$(gcloud compute instances describe $VM_NAME --zone us-central1-a 2>/dev/null | grep
     "status:")

# Check if status contains "RUNNING"
if [[ $status == *"RUNNING"* ]]; then
    echo "Instance is already running. The external IP is:"
    gcloud compute instances describe $VM_NAME --zone us-central1-a | grep "natIP:"
else
    echo "Instance is not running."
    echo "Creating vm instance ..."
    gcloud compute instances create $VM_NAME --project=abd-2024-419311 --zone=us-central1-a
        --machine-type=n2-custom-8-16384 --network-interface=network-tier=PREMIUM,stack-type
        =IPV4_ONLY,subnet=default --metadata=... --source-machine-image=vm-abd
    echo "VM instance created!"
fi
```

B Script de desactivação da VM

```
#!/bin/bash

VM_NAME="vm-abd"

# Verificar a existencia de uma flag -d
if [ "$1" == "-d" ]; then
    echo "Only deleting vm instance ..."
    gcloud compute instances stop $VM_NAME \
    && \
```

```
    gcloud compute instances delete $VM_NAME --zone=us-central1-a --quiet
else
    gcloud compute instances stop $VM_NAME \
    && \
    gcloud compute machine-images delete $VM_NAME --quiet \
    && \
    gcloud compute machine-images create $VM_NAME --source-instance=$VM_NAME --source-
        instance-zone=us-central1-a --storage-location=us-central1 \
    && \
    gcloud compute instances delete $VM_NAME --zone=us-central1-a --quiet
fi
```

C Script de medição de tempo de *queries*

```
import subprocess, psycopg2, time, sys
from prettytable import PrettyTable

DBNAME = "stack"
USER = "postgres"
PASSWORD = "postgres"
HOST = "127.0.0.1"
PORT = "5432"

TIMES = 2 # Number of times to execute the query for average

# Shell
def run_cmd(command: str):
    command_split = command.split(" ")
    subprocess.run(command_split)


def execute_query(query: str):
    conn = psycopg2.connect(f"dbname={DBNAME} user={USER} password={PASSWORD} host={HOST}
        port={PORT}")
    cur = conn.cursor()
    cur.execute(query)
```

```
    res = cur.fetchall()
    conn.close()
    return res

def measure_query(query: str, times=TIMES):
    query_explain = f"EXPLAIN ANALYZE {query}"
    exec_times = []
    execute_query(query_explain) # Warm-up query
    for i in range(times):
        plan = execute_query(query_explain)
        # print(plan)
        time = plan[-1][0].split(" ")[-2] # in ms
        time = float(time) * 0.001 # converto to secs
        print(f"Execution time: {time} secs")
        exec_times.append(time)
    avg = sum(exec_times) / len(exec_times)
    avg = round(avg, 3)
    return avg


#* Q1
def q1_change_arguments(query_str: str, interval: str) -> str:
    STANDARD = "6 months"
    query_str_res = query_str.replace(STANDARD, f"{interval}")
    if STANDARD != interval and query_str_res == query_str:
        print(f"ERROR while switching q1 args. No arguments were changed in query. Exiting...
            ")
        sys.exit(1)
    elif STANDARD == interval and query_str_res != query_str:
        print(f"WARNING: Standard argument changed query 1!!")
        sys.exit(1)
    return query_str_res

def create_q1_table(query_str):
    myTable = PrettyTable(["Arguments", "Time (s)"])
    args = ["1 month", "3 months", "6 months", "1 year", "2 year"]
    for arg in args:
        print(f"Running query with argument: {arg}")
        query = q1_change_arguments(query_str, arg)
```

```
        avg_final = measure_query(query)
        myTable.add_row([arg, avg_final])
    print(myTable)
    return myTable

def compare_q1_results(query_1, query_2):
    # Compare the results of the queries
    args = ["1 month", "3 months", "6 months", "1 year", "2 year"]
    for arg in args:
        print(f"Running query with argument: {arg}")
        # Remove LIMIT 100 from queries
        new_query_1 = q1_change_arguments(query_1, arg)
        new_query_2 = q1_change_arguments(query_2, arg)
        new_query_1 = query_1.replace("LIMIT 100", "")
        new_query_2 = query_2.replace("LIMIT 100", "")
        res1 = execute_query(new_query_1)
        res2 = execute_query(new_query_2)
        res1 = sorted(res1)
        res2 = sorted(res2)
        if res1 == res2:
            print("Results are equal!")
        else:
            print("Results are different!")

#* Q2
def q2_change_arguments(query_str: str, interval: str, bucketInterval: str) -> str:
    STANDARD_BUCKET = "5000"
    STANDARD_INTERVAL = "5 year"
    query_str_res = query_str.replace(STANDARD_BUCKET, f"{bucketInterval}")
    query_str_res = query_str.replace(STANDARD_INTERVAL, f"{interval}")
    if STANDARD_INTERVAL != interval and STANDARD_BUCKET != bucketInterval and query_str_res
         == query_str:
        print(f"ERROR while switching q2 args. No arguments were changed in query. Exiting...
            ")
        sys.exit(1)
    elif STANDARD_INTERVAL == interval and STANDARD_BUCKET == bucketInterval and
        query_str_res != query_str:
        print(f"WARNING: Standard argument changed query 2!!")
        sys.exit(1)
```

```
    return query_str_res

def create_q2_table(query_str):
    myTable = PrettyTable(["Interval", "Bucket", "Time (s)"])
    args_interval = ["1 year", "2 year", "3 year", "4 year", "5 year"] # Provisorio
    args_bucket = ["5000", "10000", "15000", "20000", "25000"] # Provisorio
    for arg_i in args_interval:
        for arg_b in args_bucket:
            print(f"Running query with arguments: {arg_i}, {arg_b}")
            query = q2_change_arguments(query_str, arg_i, arg_b)
            avg_final = measure_query(query)
            myTable.add_row([arg_i, arg_b, avg_final])
    print(myTable)
    return myTable

def compare_q2_results(query_1, query_2):
    # Compare the results of the queries
    args_interval = ["1 year", "2 year", "3 year", "4 year", "5 year"]
    args_bucket = ["5000", "10000", "15000", "20000", "25000"]
    for arg_i in args_interval:
        for arg_b in args_bucket:
            print(f"Running query with arguments: {arg_i}, {arg_b}")
            new_query_1 = q2_change_arguments(query_1, arg_i, arg_b)
            new_query_2 = q2_change_arguments(query_2, arg_i, arg_b)
            res1 = execute_query(new_query_1)
            res2 = execute_query(new_query_2)
            res1 = sorted(res1)
            res2 = sorted(res2)
            if res1 == res2:
                print("Results are equal!")
            else:
                print("Results are different!")


#* Q3
def q3_change_arguments(query_str: str, minim: str) -> str:
    STANDARD = "10"
    query_str_res = query_str.replace(STANDARD, f"{minim}")
    if STANDARD != minim and query_str_res == query_str:
```

```
135         print(f"ERROR while switching q3 args. No arguments were changed in query. Exiting...
                ")
136         sys.exit(1)
137     elif STANDARD == minim and query_str_res != query_str:
138         print(f"WARNING: Standard argument changed query 3!!")
139         sys.exit(1)
140     return query_str_res
141 
142 def create_q3_table(query_str):
143     myTable = PrettyTable(["Arguments", "Time (s)"])
144     args = ["10", "20", "30", "40", "50"]
145     for arg in args:
146         print(f"Running query with argument: {arg}")
147         query = q3_change_arguments(query_str, arg)
148         avg_final = measure_query(query)
149         myTable.add_row([arg, avg_final])
150     print(myTable)
151     return myTable
152 
153 def compare_q3_results(query_1, query_2):
154     # Compare the results of the queries
155     args = ["10", "20", "30", "40", "50"]
156     for arg in args:
157         print(f"Running query with argument: {arg}")
158         new_query_1 = q3_change_arguments(query_1, arg)
159         new_query_2 = q3_change_arguments(query_2, arg)
160         res1 = execute_query(new_query_1)
161         res2 = execute_query(new_query_2)
162         res1 = sorted(res1)
163         res2 = sorted(res2)
164         if res1 == res2:
165             print("Results are equal!")
166         else:
167             print("Results are different!")
168 
169 #* Q4
170 def q4_change_arguments(query_str: str, bucketSize: str) -> str:
171     STANDARD = "1 minute"
172     query_str_res = query_str.replace(STANDARD, f"{bucketSize}")
```

```
173     if STANDARD != bucketSize and query_str_res == query_str:
174         print(f"ERROR while switching q4 args. No arguments were changed in query. Exiting...
                ")
175         sys.exit(1)
176     elif STANDARD == bucketSize and query_str_res != query_str:
177         print(f"WARNING: Standard argument changed query 4!!")
178         sys.exit(1)
179     return query_str_res
180 
181 def create_q4_table(query_str):
182     myTable = PrettyTable(["Arguments", "Time (s)"])
183     args = ["1 minute", "10 minutes", "30 minutes", "1 hour", "6 hours"]
184     for arg in args:
185         print(f"Running query with argument: {arg}")
186         query = q4_change_arguments(query_str, arg)
187         avg_final = measure_query(query)
188         myTable.add_row([arg, avg_final])
189     print(myTable)
190     return myTable
191 
192 def compare_q4_results(query_1, query_2):
193     # Compare the results of the queries
194     args = ["1 minute", "10 minutes", "30 minutes", "1 hour", "6 hours"]
195     for arg in args:
196         print(f"Running query with argument: {arg}")
197         new_query_1 = q4_change_arguments(query_1, arg)
198         new_query_2 = q4_change_arguments(query_2, arg)
199         res1 = execute_query(new_query_1)
200         res2 = execute_query(new_query_2)
201         res1 = sorted(res1)
202         res2 = sorted(res2)
203         if res1 == res2:
204             print("Results are equal!")
205         else:
206             print("Results are different!")
207 
208 if __name__ == "__main__":
209     # python3 arg_script.py file.sql [file2.sql] --> (opcional para comparar scripts)
210     if len(sys.argv) < 2:
```

```
211         print("Usage: python3 arg_script.py file.sql [file2.sql]")
212         sys.exit(1)
213 
214     file_sql = sys.argv[1]
215     with open(file_sql, "r") as f:
216         query_str = f.read()
217 
218     compare = False
219 
220     if len(sys.argv) == 3:
221         compare = True
222         print("Comparing scripts...")
223         file_2 = sys.argv[2]
224         with open(file_2, "r") as f:
225             query_str_2 = f.read()
226 
227     if "q1" in file_sql:
228         if compare:
229             compare_q1_results(query_str, query_str_2)
230         else:
231             create_q1_table(query_str)
232     elif "q2" in file_sql:
233         if compare:
234             compare_q2_results(query_str, query_str_2)
235         else:
236             create_q2_table(query_str)
237     elif "q3" in file_sql:
238         if compare:
239             compare_q3_results(query_str, query_str_2)
240         else:
241             create_q3_table(query_str)
242     elif "q4" in file_sql:
243         if compare:
244             compare_q4_results(query_str, query_str_2)
245         else:
246             create_q4_table(query_str)
```

D Resultados pormenorizados

A seguir encontra-se uma tabela com os resultados exatos das comparações das queries transacionais com diferentes valores do parâmetro de configuração de *work_mem*.

work_mem	Query	Execution time	Planning time	Total time
4 MB	Insert on questions	71.976	3.296	75.272
	Insert on questionstags	9.831	0.042	9.873
	Insert on answers	33.734	0.034	33.768
	Update on questions	1.059	8.661	9.72
	Insert on votes	24.26	0.036	24.296
	Select from questions	0.137	21.817	21.954
	Select from votes	0.643	1.188	1.831
	Select from users	3.271	3.762	7.033
	Select from badges	5.191	2.044	7.235
	Select from questions (tsvector)	37.027	0.419	37.446
	Select from questions (joins)	5055.761	4.245	5060.006
16 MB	Insert on questions	0.167	0.053	0.22
	Insert on questionstags	0.779	0.031	0.81
	Insert on answers	0.882	0.04	0.922
	Update on questions	0.045	0.088	0.133
	Insert on votes	0.101	0.037	0.138
	Select from questions	0.074	0.436	0.51
	Select from votes	0.089	0.155	0.244
	Select from users	0.037	0.064	0.101
	Select from badges	0.157	0.082	0.239
	Select from questions (tsvector)	9.796	0.149	9.945
	Select from questions (joins)	62.575	0.276	62.851
32 MB	Insert on questions	0.388	0.055	0.443
	Insert on questionstags	0.818	0.029	0.847
	Insert on answers	4.277	0.033	4.31
	Update on questions	0.126	0.102	0.228
	Insert on votes	4.717	0.04	4.757
	Select from questions	0.123	3.723	3.846
	Select from votes	0.077	0.144	0.221
	Select from users	0.037	0.076	0.113
	Select from badges	0.145	0.08	0.225
	Select from questions (tsvector)	28.613	0.149	28.762
	Select from questions (joins)	112.105	0.273	112.378

Tabela D.1: Tempos de execução (ms) para as diferentes interrogações em várias configurações de work_mem