



Universidade do Minho  
Departamento de Informática

# Administração de Bases de Dados

Ano Letivo 2024/2025

## Relatório do Trabalho Prático - Grupo 6

Diogo Araújo - PG55933

Pedro Leiras - PG55995

Lucas Oliveira - PG57886

Ricardo Pereira - PG56001

# Índice

<b>1. Introdução .....</b>	<b>3</b>
1.1. Contexto e Desafios .....	3
1.2. Metodologia .....	3
<b>2. Carga Transacional .....</b>	<b>4</b>
2.1. Otimizações .....	5
2.1.1. getGameRecentReviews .....	5
2.1.2. getRecentGamesPerTag .....	6
2.1.3. getGamesByTitle .....	8
2.1.4. getGamesSemantic .....	9
2.1.5. Abort rate .....	11
2.1.6. Otimização posterior .....	12
<b>3. Interrogações Analíticas .....</b>	<b>13</b>
3.1. Query 1 .....	13
3.1.1. Otimizações .....	13
3.1.2. Impacto na carga transacional .....	16
3.2. Query 2 .....	17
3.2.1. Otimizações .....	17
3.2.2. Impacto na carga transacional .....	20
3.3. Query 3 .....	20
3.3.1. Otimizações .....	20
3.3.2. Impacto na carga transacional .....	24
3.4. Query 4 .....	26
3.4.1. 1ª Alteração .....	27
3.4.2. 2ª Alteração .....	27
3.4.3. 3ª Alteração .....	27
3.4.4. 4ª Alteração .....	28
3.4.5. Impacto na carga transacional .....	29
3.5. Query 5 .....	29
3.5.1. Impacto na carga transacional .....	30
<b>4. Conclusão .....</b>	<b>31</b>

# 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 pequeno excerto do *dataset* da loja online Steam, abrangendo operações transacionais e analíticas.

O relatório está dividido em duas partes principais: numa primeira fase são abordadas as otimizações relativas à carga transacional, e numa segunda fase as interrogações analíticas. Para cada uma das otimizações implementadas, são descritos os passos realizados, as decisões técnicas e o respetivo impacto no desempenho do sistema.

## 1.1. Contexto e Desafios

O trabalho consistiu na simulação de operações de *backend* da Steam, como por exemplo, a adição de tempo jogado por utilizador, compra de jogos, críticas, amizades, e consultas sobre jogos e utilizadores. Estas operações compõem a componente transacional. Adicionalmente, foram realizadas interrogações analíticas sobre o mesmo conjunto de dados, focando em aspetos estatísticos e de *data warehousing*, com o objetivo de obter métricas e indicadores úteis para a plataforma.

A execução das tarefas foi feita numa máquina virtual na Google Cloud, usando PostgreSQL como sistema de gestão de base de dados, e considerando cargas transacional (com 16 clientes) e analítica (com 1 cliente), tal como especificado no enunciado.

## 1.2. Metodologia

A metodologia adotada inclui a otimização do desempenho tanto da carga transacional como das interrogações analíticas, considerando aspetos como redundância (índices, materialização), paralelismo, configuração do sistema, e alterações ao código SQL e Java.

De forma a facilitar a gestão do ambiente de trabalho, foi criado um projeto na Google Cloud partilhado por todos os membros do grupo. A instância da máquina virtual foi configurada com base nas especificações fornecidas no enunciado, e os recursos necessários foram provisionados de forma automatizada.

## 2. Carga Transacional

O grupo decidiu que antes de iniciar qualquer otimização deveria analisar a *workload* transacional para encontrar os maiores gargalos de desempenho existentes de forma a alocar os seus esforços em otimizá-los, figura 1.

```
Response time per function (ms)
-----
AddPlaytime = 2.665
ReviewGame = 2.072
BuyGame = 1.934
NewFriendship = 2.903
NewGame = 4.169
NewUser = 2.614
GameInfo = 2.180
GameReviews = 1945.589
UserInfo = 8.443
RecentGamesPerTag = 174.408
SearchGames = 1113.274

Overall metrics
-----
throughput (txn/s) = 93.071
response time (ms) = 171.701
abort rate (%) = 0.381
```

Figura 1: Análise do *workload*

Com recurso ao pgbadger analisamos qual a frequência de cada tipo de *query* (figura 2) e a distribuição dos tempos médios durante uma *workload* (figura 3).

Type	Count	Percentage
SELECT	4,258	45.01%
INSERT	479	5.06%
UPDATE	418	4.42%
DELETE	0	0.00%
COPY FROM	0	0.00%
COPY TO	0	0.00%
CTE	23	0.24%
DDL	0	0.00%
TCL	4,282	45.26%
CURSOR	0	0.00%
OTHERS	1	0.01%

Figura 2: Análise do pgbadger

Range	Count	Percentage
0-1ms	9,732	88.47%
1-5ms	520	4.73%
5-10ms	96	0.87%
10-25ms	8	0.07%
25-50ms	2	0.02%
50-100ms	4	0.04%
100-500ms	268	2.44%
500-1000ms	251	2.28%
1000-10000ms	118	1.07%
> 10000ms	1	0.01%

Figura 3: Análise do pgbadger

Como podemos ver na figura 3, o tempo que mais é frequente nas operações transacionais são entre 0 e 1ms, porém existem bastantes operações que contem um tempo médio muito alto, levando a um *throughput* maior.

## 2.1. Otimizações

O grupo decidiu que iria começar as otimizações pela operação com maior tempo médio, analisando o seu plano inicial.

### 2.1.1. getGameRecentReviews

```
select u.id, u.username, r.created_date, r.recommend, r.text, l.playtime
from review r
join users u on u.id = r.user_id
join library l on l.user_id = r.user_id and l.game_id = r.game_id
where r.game_id = ?
order by r.created_date desc
limit 25
```

Plano: <https://explain.dalibo.com/plan/agf74bb14g266a4d>

Criamos um índice no **game\_id** da tabela **review**, juntamente com o **created\_date** já organizado assim não é necessário pesquisar as *entries* todas da tabela **review** e também não é necessário ordenar uma vez que o índice já está ordenado:

```
CREATE INDEX IF NOT EXISTS index_getGameRecentReviews ON review (game_id, created_date DESC);
```

Novo plano: <https://explain.dalibo.com/plan/e5fadece37gfb729>

Também fazemos um **SELECT** do **score** para obter o **score** do jogo através da seguinte query:

```
select round(sum(recommend::int) / count(*)::decimal * 100, 3)
from review
where game_id = ?
```

Plano: <https://explain.dalibo.com/plan/deb3g7ehg474249e>

Como podemos ver no plano a *query* usa um *bitmap index scan*, pouco eficiente para o tipo de dados usado. O grupo optou então por realizar a seguinte otimização:

```
CREATE INDEX IF NOT EXISTS index_review_gameid_recommend ON review (game_id, recommend);
```

Novo plano: <https://explain.dalibo.com/plan/c31gb75b024b37aa>

Como podemos ver na figura 4, com este índice conseguimos usar o *index scan* tornando a *query* mais rápida. Esta otimização diminuiu bastante o tempo médio da pesquisa de **reviews** recentes. Uma vez que esta é uma das operações com maior tempo de resposta o grupo conseguiu aumentar o *throughput* e diminuir o tempo médio geral. Notamos que o acréscimo destes índices podem ter aumentado o tempo médio das operações e **inserts** na tabela **review** uma vez que existe a necessidade por parte do PostgreSQL de ter em conta todos os índices de cada tabela.

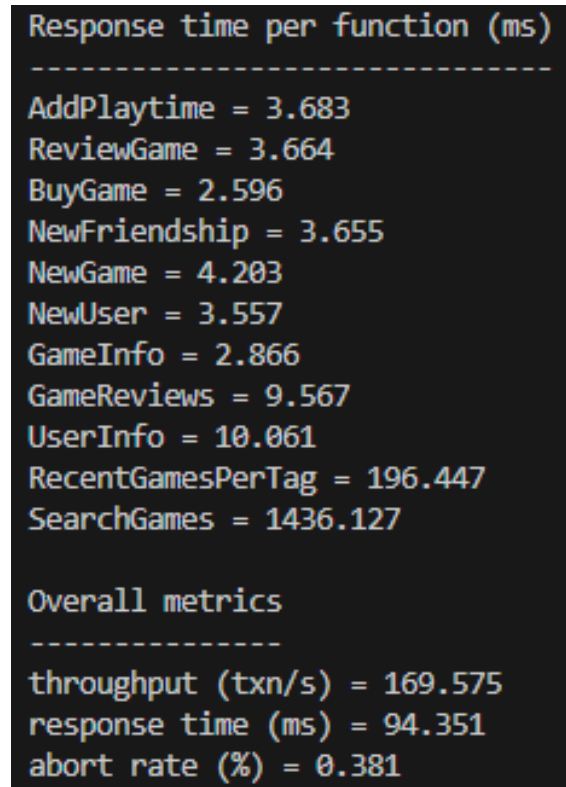


Figura 4: Análise do *workload*

### 2.1.2. getRecentGamesPerTag

```
select g.id, g.name, g.release_date
  from tag t
 join games_tags gt on gt.tag_id = t.id
 join game g on g.id = gt.game_id
 where t.id = ?
 order by g.release_date desc
 limit 25
```

Plano: <https://explain.dalibo.com/plan/184g1e8c45e8ef4g>

A maior parte do tempo de execução desta *query* passa pelo *scan* do **ORDER BY**. Como tal criamos um índice com os dados ordenados para esta *query*. Decidimos ainda criar um índice para o *scan* do **tag\_id** juntamente com o **game\_id**.

*Query* para criação dos índices:

```
CREATE INDEX IF NOT EXISTS index_game_release_date ON game (release_date DESC);
CREATE INDEX IF NOT EXISTS index_games_tags_tag_id_game_id ON games_tags (game_id, tag_id);
```

Novo plano: <https://explain.dalibo.com/plan/d89g88acb3b22c6h>

Observando a *query* podemos ver que o **search** na tabela **tag** não é necessária uma vez que apenas é utilizada para o **tag\_id**, sendo que já é obtido no **game\_tags**, decidimos remover completamente a tabela da *query*.

```
select g.id, g.name, g.release_date
  from games_tags gt
 join game g on g.id = gt.game_id
 where gt.tag_id = ?
 order by g.release_date desc
 limit 25
```

Novo plano: <https://explain.dalibo.com/plan/hbefgagc0731572b>



```
Response time per function (ms)
-----
AddPlaytime = 3.504
ReviewGame = 3.490
BuyGame = 2.485
NewFriendship = 3.599
NewGame = 4.343
NewUser = 3.334
GameInfo = 2.723
GameReviews = 9.009
UserInfo = 9.884
RecentGamesPerTag = 137.425
SearchGames = 1322.074

Overall metrics
-----
throughput (txn/s) = 190.129
response time (ms) = 83.983
abort rate (%) = 0.430
```

Figura 5: Análise do *workload*

Como podemos ver na análise do pgbadger (figura 5) e as operações com mais tempo do **RecentGamesPerTag**, conseguimos observar que em determinadas *queries* com diferentes parâmetros, como por exemplo, usando a **tag\_id** com o valor de 334 o PostgreSQL tem de percorrer 90000 linhas até encontrar 25 registos com essa **tag** ao contrário das 2000 com **tag\_id** = 1. De forma a evitar estes resultados o grupo decidiu alterar a *query* que, apesar de a tornar ligeiramente mais lenta, quando existe um avultado número de registos realizamos primeiro a filtragem do **game\_id**.

```

Response time per function (ms)
-----
AddPlaytime = 2.417
ReviewGame = 2.661
BuyGame = 1.980
NewFriendship = 2.598
NewGame = 2.498
NewUser = 2.201
GameInfo = 2.028
GameReviews = 6.281
UserInfo = 5.608
RecentGamesPerTag = 13.529
SearchGames = 3.984

Overall metrics
-----
throughput (txn/s) = 4012.569
response time (ms) = 3.987
abort rate (%) = 0.379

```

Figura 6: Análise do *workload*

### 2.1.3. getGamesByTitle

```

select id, name, release_date
  from game
 where to_tsvector('english', name) to_tsquery('english', ?)
 limit 25

```

Plano: <https://explain.dalibo.com/plan/178fc5ee19f4b17b>

Inicialmente, observámos que a *query* baseada em **to\_tsvector** e **to\_tsquery** estava a causar um *Sequential Scan* com um custo de cerca de 370ms apenas na leitura.

Exploramos alternativas como alterar a *query* para utilizar **name LIKE '%name%'**, o que se revelou mais eficaz inicialmente em termos de legibilidade e simplicidade, mas trouxe limitações importantes: um índice *B-tree* não pode ser utilizado quando o padrão começa com %, porque o PostgreSQL não consegue prever onde na string o valor poderá ocorrer, obrigando assim a um *Sequential Scan*, mesmo com o índice criado.

Para mitigar isso, tentamos adaptar o padrão para **name LIKE 'name%'**, o que já permitiria o uso de um índice *B-tree* e traria ganhos de desempenho. No entanto, esta abordagem altera a semântica da *query*.

Dado que o objetivo era manter a expressividade e robustez da pesquisa textual com **to\_tsquery**, a solução adequada foi criar um índice do tipo GIN (*Generalized Inverted Index*) com base no **to\_tsvector**, da seguinte forma:

```

CREATE INDEX IF NOT EXISTS index_game_exact ON game USING GIN (to_tsvector('english',
name));

```

Novo plano: <https://explain.dalibo.com/plan/fbg22b73agce0e2c>



## 2.1.4. getGamesSemantic

```
select g.id, g.name, g.release_date
  from game g
 join game_search_embedding gse on gse.game_id = g.id
 join query_embedding_sample qes on true
 where qes.id = ?
 order by gse.embedding <-> qes.embedding
 limit 25;
```

Plano: <https://explain.dalibo.com/plan/aa2fcg055c021bea>

Para esta *query* estamos a trabalhar com vetores (pgvector) de *l2 distance*, para isto existem dois índices: **ivfflat** ou **hnsw**.

```
CREATE INDEX IF NOT EXISTS index_qes_ivfflat ON query_embedding_sample USING ivfflat
(embedding vector_l2_ops) WITH (lists = 100);
CREATE INDEX IF NOT EXISTS index_gse_ivfflat ON game_search_embedding USING ivfflat
(embedding vector_l2_ops) WITH (lists = 100);
```

```
CREATE INDEX index_game_search_embedding_embedding_hnsw ON game_search_embedding USING
hnsw (embedding vector_l2_ops);
```

A diferença entre os índices não é muita em termos de tempo de execução/*planning time* da *query*, sendo só diferente o tempo de criação dos índices, demorando bastante mais tempo a criar um índice **hnsw** do que um índice **ivfflat**.

No entanto, foi necessário alterar a estrutura da *query* para que o PostgreSQL utilizasse corretamente os índices de similaridade vetorial. A *query* original não forçava a aplicar o índice de proximidade.

Por isso, reescrevemos da seguinte forma:

```
SELECT g.id, g.name, g.release_date
FROM query_embedding_sample qes,
LATERAL (
  SELECT g.id, g.name, g.release_date
  FROM game_search_embedding gse
  JOIN game g ON g.id = gse.game_id
  ORDER BY gse.embedding <-> qes.embedding
  LIMIT 25
) g
WHERE qes.id = ?;
```

Esta versão utiliza uma *subquery LATERAL*, permitindo passar o valor do vetor de **query\_embedding\_sample** para o **ORDER BY** interno, forçando o PostgreSQL a considerar o índice vetorial.

Novo plano: <https://explain.dalibo.com/plan/442fh4fd83he781f>

Usando os índices **ivfflat** notamos a redução do tempo da *query* alvo, por volta de 6 segundos, mas as operações **OLTP** aumentaram bastante o seu tempo de resposta.

Com o índice **hnsw** também obtivemos um aumento nas outras operações apesar de não ser tão acentuado. O *response time* foi um bocado menor comparando com os outros índices o que nos levou a optar a usar o **hnsw**, apesar do tempo que demorava a criar o índice.

```

Response time per function (ms)
-----
AddPlaytime = 2.230
ReviewGame = 2.578
BuyGame = 2.356
NewFriendship = 3.095
NewGame = 2.478
NewUser = 2.232
GameInfo = 2.186
GameReviews = 4.153
UserInfo = 3.369
RecentGamesPerTag = 15.562
SearchGames = 4.537

Overall metrics
-----
throughput (txn/s) = 4054.228
response time (ms) = 3.946
abort rate (%) = 0.388

```

Figura 7: Análise do *workload*

Usando o pgbadger, olhamos novamente para a tabela da distribuição do tempos médios da *queries* (figura 8).

Podemos ver que os tempo acima de 100ms foram todos removidos, sendo agora uma pequena percentagem entre 50 a 100ms.

Entretanto, apesar de termos uma distribuição que favorece o *throughput* uma vez que todos os tempos são geralmente mais baixos a percentagem de *queries* entres os 0 a 1ms diminuiu em relação ao início, isto deve-se às operações de **OLTP** que normalmente seriam a maior percentagem destas *queries* que agora têm o *overhead* de atualizar os índices criados nas tabelas que possivelmente estão a inserir ou alterar dados aumentando ligeiramente os tempos destas operações de tempo baixo.

Range	Count	Percentage
0-1ms	11,469	81.01%
1-5ms	1,572	11.10%
5-10ms	597	4.22%
10-25ms	245	1.73%
25-50ms	247	1.74%
50-100ms	28	0.20%
100-500ms	0	0.00%
500-1000ms	0	0.00%
1000-10000ms	0	0.00%
> 10000ms	0	0.00%

Figura 8: Análise do pgbadger

Analisando a tabela *Slowest Individual Queries* (figura 9), podemos ver *queries* como a **getUser-TopGames**, pertencente à **UserInfo**, que apresenta um tempo de 92ms, apesar de o seu tempo médio ser apenas de 5ms.

## 🔍 Slowest individual queries

Rank	Duration	Query
1	92ms	<pre>SELECT id, name, playtime FROM library JOIN game ON id = game_id WHERE user_id = '477523' ORDER BY 3 DESC LIMIT 5;</pre> <div>[ Date: 2025-05-04 20:44:06 - Database: abd - User: postgres - Remote: 127.0.0.1 - Application: PostgreSQL JDBC Driver - Bind query: yes ]</div>

Figura 9: Análise tabela SIQ

As possíveis causas para este pico de tempo de execução, podem estar relacionadas com a ausência de um índice adequado na tabela **library**, fazendo com que na primeira vez tenha que ir buscar muitos dados ao disco. A *query* em questão utiliza filtros por **user\_id**, ordena por **playtime DESC** e associa-se ao jogo via **game\_id**. Assim, uma solução natural seria a criação do seguinte índice:

```
CREATE INDEX index_library_test_tripli ON library(user_id, playtime DESC, game_id);
```

Após a criação deste índice, verificámos que o problema do pico de tempo na **getUserTopGames** foi efetivamente resolvido. A *query* passou a executar-se de forma consistente dentro dos tempos esperados.

No entanto, esta otimização teve um impacto negativo notório noutro ponto do sistema: o *throughput* geral caiu drasticamente. Verificámos um aumento significativo no tempo médio das operações que fazem **inserts** na tabela e **update** como **AddPlaytime**.

Estas *queries* fazem atualizações do campo **playtime**, o que significa que ao existir um índice que inclui essa mesma coluna (**playtime DESC**), o PostgreSQL é forçado a atualizar esse índice em cada **UPDATE**, o que tem um custo elevado.

Além disso, a **AddPlaytime** é uma das *queries* mais frequentes no sistema (executada 731 vezes), o que agrava ainda mais o impacto desta escolha de índice.

Assim, podemos concluir que, apesar de este índice evitar um pico na *query* da performance da **getUserTopGames**, ele não é eficaz do ponto de vista global do sistema, dado o impacto negativo nas operações de escrita críticas e frequentes.

### 2.1.5. Abort rate

Analisando o nosso *workload* obtemos uma média de 0.388 de *abort rate*, que analisando os *logs* reparamos que se deve a *duplicate entries* na tabela **reviews**. Analisando o *schema* e os valores reparamos que provavelmente são tentativas de atualização de uma **review** ou seja, são operações válidas que deviam estar a ser realizadas. Para evitar que as novas **reviews** sejam perdidas, atualizamos o **addReview** para quando existir conflito em vez de dar erro ou descartar a nova entrada, realizar um **UPDATE** com os novos valores:

```
insert into review (user_id, game_id, created_date, recommend, text)
values (?, ?, now(), ?, ?)
on conflict (user_id, game_id)
do update set
created_date = NOW(),
recommend = excluded.recommend,
text = excluded.text;
```

```

Response time per function (ms)
-----
AddPlaytime = 2.340
ReviewGame = 2.278
BuyGame = 2.138
NewFriendship = 2.801
NewGame = 2.319
NewUser = 2.113
GameInfo = 2.035
GameReviews = 3.103
UserInfo = 4.128
RecentGamesPerTag = 49.705
SearchGames = 4.949

Overall metrics
-----
throughput (txn/s) = 2178.395
response time (ms) = 7.343
abort rate (%) = 0.000

```

Figura 10: Abort rate

## 2.1.6. Optimização posterior

Reparamos o tempo da *query* **getGameRecentReviews** por vezes era bastante demorado. Analisando o pgBadger vimos que os testes efetuados anteriormente que eram feitos com o **game\_id** = 10 eram realmente mais rápidos uma vez que percorria poucos registos na tabela **library** e quando era um jogo como, por exemplo, **game\_id** = 386360 percorria bastante mais registos tornando esta operação muito lenta (figura 11).

🐞 Slowest individual queries

Rank	Duration	Query
1	2s652ms	<pre> SELECT u.id, u.username, r.created_date, r.recommend, r.text, l.playtime FROM review r JOIN users u ON u.id = r.user_id JOIN library l ON l.user_id = r.user_id AND l.game_id = r.game_id WHERE r.game_id = '386360' ORDER BY r.created_date DESC LIMIT 25; </pre>

[ Date: 2025-05-13 14:16:59 - Bind query: yes ]

Figura 11: Análise tabela SIQ

Para mitigar isto alteramos a *query* para limitar inicialmente o número de registos a 25, fazendo com que mesmo que tenha muitos registos apenas percorra os 25 antes de fazer **JOIN**, não necessitando assim de juntar várias colunas que vão ser descartadas:

```

SELECT u.id, u.username, r.created_date, r.recommend, r.text, l.playtime
FROM (
    SELECT * FROM review
    WHERE game_id = ?
    ORDER BY created_date DESC
    LIMIT 25
) r
JOIN users u ON u.id = r.user_id
JOIN library l ON l.user_id = r.user_id AND l.game_id = r.game_id;

```

## 3. Interrogações Analíticas

### 3.1. Query 1

#### 3.1.1. Otimizações

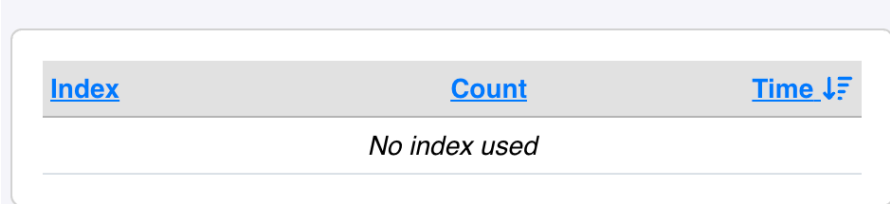
```
SELECT year, id, name, sales
FROM (
    SELECT year, id, name, sales,
           rank() OVER (PARTITION BY year ORDER BY sales DESC) AS rank
    FROM (
        SELECT l.year, g.id, g.name, count(*) AS sales
        FROM game g
        JOIN (
            SELECT extract(year FROM added_date) AS year, game_id
            FROM library
        ) l ON l.game_id = g.id
        WHERE price > 0
        GROUP BY 1, 2, 3
    )
)
WHERE rank = 1
ORDER BY year DESC;
```

A *query* em questão retorna os jogos mais vendidos de cada ano, ordenados do mais recente para o mais antigo, considerando apenas aqueles com preço superior a zero. Após compreendermos a lógica da *query*, realizámos uma análise dos planos de execução gerados pelo PostgreSQL, utilizando as ferramentas **EXPLAIN ANALYZE** e o site **explain.dalibo.com**. Verificou-se que o tempo médio de execução da *query* base foi de 2m41s, tendo o diagrama correspondente ao plano de execução da *query* 1 base sido o seguinte: <https://explain.dalibo.com/plan/ggfd03da83977378>

Ao analisar o plano, conseguimos verificar que a parte que mais tempo demorou foi o **sort**, que era responsável por ordenar os resultados por ano e vendas. Então inicialmente a 1ª alteração que fizemos foi a criação do seguinte índice:

```
CREATE INDEX idx_library_year ON library ((extract(year FROM added_date)));
```

Este índice facilitaria a extração do ano, o que poderia reduzir a quantidade de dados processados durante a agregação e ordenação. Com isso, voltamos a avaliar o desempenho da *query*, e reparamos que o índice que criamos não estava a ser usado pelo *planner*.



Index	Count	Time ↓
No index used		

Figura 12: Não utilização do Índice na Q1

O que nos levou a analisar o porquê da não utilização do índice e reparamos que a extração do ano ocorre dentro de uma *subquery*, o que dificultava o *planner* a reconhecer a vantagem de utilizar o índice ao planear a *query* externa. Para além disso, o uso do **RANK()** com **PARTITION BY** e **ORDER BY** obriga o *planner* a fazer uma ordenação dos dados, o que normalmente leva à escolha de um *Seq Scan* com *Sort*, em vez de um *Index Scan*.

Após esta análise, decidimos então tentar reestruturar a *query* de forma a que o *planner* consiga utilizar o índice criado e também tentamos arranjar forma de reestruturar de forma a que também houvesse melhoria no seu desempenho, chegando então a esta *query*:

```
SELECT DISTINCT ON (year)
    EXTRACT(YEAR FROM l.added_date) AS year,
    g.id,
    g.name,
    COUNT(*) AS sales
FROM library l
JOIN game g ON l.game_id = g.id
WHERE g.price > 0
GROUP BY year, g.id, g.name
ORDER BY year DESC, sales DESC;
```

Nesta reestruturação da *query* tentamos simplificar a lógica de ordenação e agregação, retirando o uso de **RANK()** com **PARTITION BY**, que forçava o *planner* a realizar *Sort* e *WindowAgg* dispendiosos. Ao trocarmos por um **DISTINCT ON (year)**, passámos a aproveitar a ordenação direta por **year DESC**, **sales DESC**, permitindo obter apenas o jogo mais vendido por ano de forma mais eficiente.

Para além disso, ao manter o **EXTRACT(YEAR FROM l.added\_date)** diretamente no **SELECT** e **GROUP BY**, aumentámos a probabilidade de o *planner* utilizar o índice *idx\_library\_year*, previamente criado.

Plano: <https://explain.dalibo.com/plan/d40258d10g73e9hb>

Mas apesar da reestruturação da *query*, não houve melhoria no desempenho, o *planner* continuou a mostrar o *Sort* e o *Hash Join* como os principais responsáveis pelo custo total, e o índice ainda não foi utilizado.

Então percebemos que a razão de isto acontecer é porque como o índice é funcional, ele só é usado se a expressão **EXTRACT(YEAR FROM added\_date)** for aplicada diretamente numa cláusula **WHERE**, o que não é o caso nesta *query*. Como apenas agrupamos e ordenamos pelo ano, o *planner* opta por um *Parallel Seq Scan* seguido de ordenação externa, mantendo o tempo de execução elevado.

Tentamos então outra abordagem, e reestruturamos a *query* utilizando **CTEs** (*Common Table Expressions*)

```
WITH sales AS (
    SELECT
        EXTRACT(YEAR FROM l.added_date) AS year,
        g.id,
        g.name,
        COUNT(*) AS sales
    FROM library l
    JOIN game g ON l.game_id = g.id
    WHERE g.price > 0
    GROUP BY year, g.id, g.name
),
ranked AS (
    SELECT *,
        ROW_NUMBER() OVER (PARTITION BY year ORDER BY sales DESC) AS rn
    FROM sales
)
SELECT year, id, name, sales
FROM ranked
WHERE rn = 1
ORDER BY year DESC;
```

Utilizamos também a função *ROW\_NUMBER()* para selecionar o jogo mais vendido por ano. A ideia com isto seria separar a contagem de vendas e aplicar a ordenação por *ranking* de forma mais explícita e modular. Mas no entanto ao analisarmos o plano de execução: <https://explain.dalibo.com/plan/ddfd85619ae84g43>

Verificamos que o custo continuava elevado. O *Sort* e o *Hash Join* permaneceram como operações dominantes, e o índice funcional continuava a não ser utilizado. O que nos levou a confirmar que apesar da clareza estrutural trazida pelos CTEs, esta abordagem não resultou em melhorias de desempenho e também não ajudou o *planner* a utilizar o índice funcional.

Durante estas tentativas, também testamos outros índices, como por exemplo:

```
CREATE INDEX idx_library_year ON library ((extract(year FROM added_date)));
CREATE INDEX idx_library_year_gameid_price ON library (
  extract(year FROM added_date), game_id
) WHERE price > 0;
CREATE INDEX idx_library_gameid_year ON library(game_id, extract(year FROM added_date));
```

Mas nenhum revelou ser efetivo para o *planner* durante as tentativas.

Então, decidimos tentar outra alternativa para melhorar a *query*.

Como verificamos que os dados referentes a anos anteriores nunca é alterado, optamos por criar um tabela com os dados já processados para melhorar a *performance*. Criamos a tabela nova e preenchemos com os dados referentes ao jogo mais vendido cada ano e o número de vendas:

```
CREATE TABLE IF NOT EXISTS top_sales_game_per_year (
  year      INTEGER PRIMARY KEY,
  game_id   INTEGER NOT NULL,
  game_name TEXT,
  sales     INTEGER
);

INSERT INTO top_sales_game_per_year (year, game_id, game_name, sales)
SELECT year, game_id, game_name, sales
FROM (
  SELECT EXTRACT(YEAR FROM l.added_date) AS year,
         l.game_id,
         g.name AS game_name,
         COUNT(*) AS sales,
         ROW_NUMBER() OVER (PARTITION BY EXTRACT(YEAR FROM l.added_date) ORDER BY
COUNT(*) DESC) AS rn
  FROM library l
  JOIN game g ON l.game_id = g.id
  WHERE EXTRACT(YEAR FROM l.added_date) <= 2024
        AND g.price > 0
  GROUP BY EXTRACT(YEAR FROM l.added_date), l.game_id, g.name
) t
WHERE rn = 1;
```

Para complementar, criamos uma tabela como o objetivo de manter os registos atualizados do ano atual, de forma a que esta tabela contenha menos registos uma vez que são apenas dados referentes a um ano, tornando assim mais eficaz os *updates* aquando da inserção de dados na tabela **library**.

Para manter esta tabela atualizada adicionamos um *trigger* e, sempre que for inserido algum registo na tabela **library**, esta nova tabela referente ao ano atual também é atualizada:

```

CREATE TABLE IF NOT EXISTS sales_currentYear (
  game_id    INTEGER PRIMARY KEY,
  game_name  TEXT,
  sales      INTEGER
);

INSERT INTO sales_currentYear (game_id, game_name, sales)
SELECT
  g.id,
  g.name,
  COUNT(*) AS sales
FROM library l
JOIN game g ON l.game_id = g.id
WHERE EXTRACT(YEAR FROM l.added_date) = EXTRACT(YEAR FROM CURRENT_DATE)
  AND g.price > 0
GROUP BY g.id, g.name;

```

Com as novas tabelas podemos alterar a *query* para percorrer essas tabelas, faltando fazer os cálculos na tabela do **currentYear** mas sendo muito mais eficaz:

```

Select * FROM top_sales_game_per_year
UNION ALL
SELECT EXTRACT(YEAR FROM CURRENT_DATE)::INT AS year, game_id, game_name, sales
FROM (
  SELECT game_id, game_name, sales
  FROM sales_currentyear
  ORDER BY sales DESC
  LIMIT 1
) t
ORDER BY year DESC;

```

Plano: <https://explain.dalibo.com/plan/fd3fb5g7ggabf5de>

Um problema detectado foi a necessidade de realizar um *sequencial scan*, para evitar percorrer a tabela toda, o que pode se tornar num gargalo maior com o número de registos. Para tal adicionamos um índice:

```

CREATE INDEX IF NOT EXISTS index_sales_currentyear_sales_desc ON sales_currentYear
(sales DESC);

```

Com o novo índice já conseguimos realizar um *indice scan* reduzindo bastante o tempo que para percorrer a tabela passando agora a demorar cerca de 0.05ms.

Plano: <https://explain.dalibo.com/plan/b56eg11445ee72d7>

### 3.1.2. Impacto na carga transacional

Com estas otimizações, a carga transacional precisará de lidar com a adição para a nova tabela que contém os dados da quantidade de vendas de cada jogo para o ano atual realizado pelo *trigger*. Como a tabela apenas contém dados de compras dos jogos de um ano, o impacto no transacional do **BuyGame**, responsável por adicionar um jogo à **library**, é mínimo e tem muito pouco *overhead* na *workload*.



## 3.2. Query 2

### 3.2.1. Otimizações

```
WITH args AS (
  SELECT 3331100 AS id
)
SELECT g.id, g.name
FROM game g
JOIN library l ON l.game_id = g.id
JOIN (
  SELECT u.id
  FROM args
  JOIN users u ON true
  JOIN friendship f
    ON (f.user_id_1 = u.id AND f.user_id_2 = args.id)
    OR (f.user_id_1 = args.id AND f.user_id_2 = u.id)
  WHERE u.id <> args.id
) u ON u.id = l.user_id
WHERE (
  SELECT count(*) = 0
  FROM args
  JOIN library ON library.user_id = args.id
  JOIN game ON game.id = library.game_id
  WHERE game.id = g.id
)
GROUP BY 1, 2
ORDER BY count(*) DESC
LIMIT 15;
```

Plano: <https://explain.dalibo.com/plan/3f82cahdfbc1a3cg>

O tempo de resposta inicial foi cerca de 30 segundos.

Começamos por analisar a *query* e notamos que na cláusula **WHERE** havia um **JOIN** com a tabela **game** que era desnecessário e decidimos remover.

Para além disso, fazer um **WHERE (SELECT count(\*) = 0 FROM library...)** obrigava sempre a percorrer a tabela **library** inteira para no final contar as linhas e comparar com 0, o que poderia ser ineficiente.

Optamos por mudar para **WHERE NOT EXISTS (SELECT 1 FROM library ...)** uma vez que deste modo a *subquery* é terminada assim que encontrar uma entrada que preencha as condições, percorrendo a tabela inteira apenas para os casos em que não haja entrada na tabela e percorrendo menos entradas em alguns casos.

Analisando o output do **EXPLAIN** notamos que a maior parte do tempo de resposta está concentrado no cálculo dos amigos do **user** (figura 13).

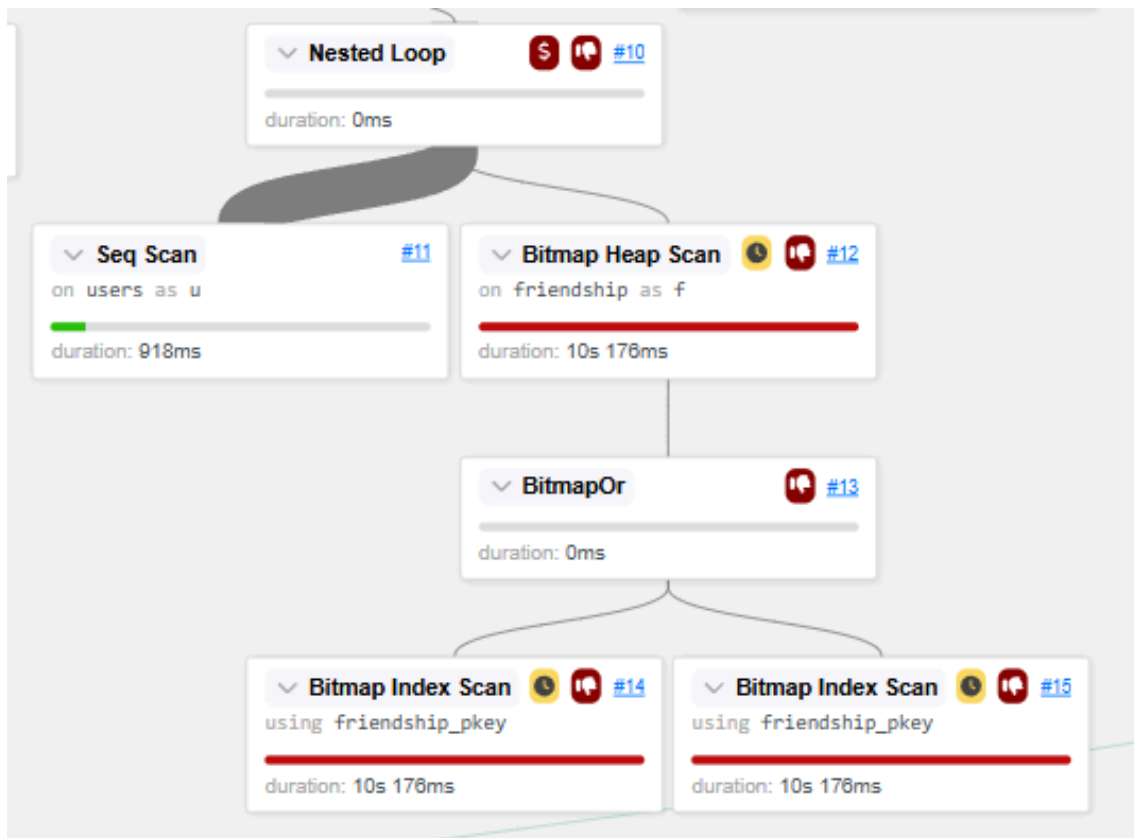


Figura 13: Gargalo da query

Na figura 13 podemos observar que a tabela **friendship** estava a ser percorrida 3 vezes e, apesar de 2 delas indicarem o uso do índice da sua *primary key*, o tempo elevado provou que algo não estava a ser incluído nesse índice.

Procuramos fazer com que cada leitura da tabela extraísse o máximo proveito do índice. Para tal começamos por mudar a condição do **JOIN** para ler apenas uma coluna de cada vez, ao invés de ambas as colunas em cada lado do **OR**.

Query original:

```
SELECT u.id
FROM args
JOIN users u ON true
JOIN friendship f
  ON (f.user_id_1 = u.id AND f.user_id_2 = args.id)
  OR (f.user_id_1 = args.id AND f.user_id_2 = u.id)
WHERE u.id <> args.id
```

Query otimizada:

```
friends AS (
  SELECT
    CASE
      WHEN f.user_id_1 = args.id THEN f.user_id_2
      ELSE f.user_id_1
    END AS friend_id
  FROM friendship f, args
  WHERE f.user_id_1 = args.id OR f.user_id_2 = args.id
)
```

Uma vez que não precisávamos de nenhuma informação adicional da tabela **users** que não estivesse já presente na tabela **friendship**, não houve problema em remover o **JOIN** de ambas.

Com estas mudanças a *query* ficou a seguinte:

```

WITH args AS (
  SELECT 3331100 AS id
),
friends AS (
  SELECT
    CASE
      WHEN f.user_id_1 = args.id THEN f.user_id_2
      ELSE f.user_id_1
    END AS friend_id
  FROM friendship f, args
  WHERE f.user_id_1 = args.id OR f.user_id_2 = args.id
)
SELECT g.id, g.name
FROM friends f
JOIN library l ON f.friend_id = l.user_id
JOIN game g ON g.id = l.game_id
WHERE NOT EXISTS (
  SELECT 1
  FROM library lib, args
  WHERE lib.user_id = args.id
  AND lib.game_id = g.id
)
GROUP BY 1, 2
ORDER BY count(*) DESC
LIMIT 15;

```

Plano: <https://explain.dalibo.com/plan/782h92g7dde6e1g7>

O tempo de resposta baixou para cerca de 3 segundos, sendo que 96% desse tempo ainda era gasto a percorrer a tabela **friendship** para ler a coluna **user\_id\_2**. No entanto, a mesma leitura na coluna **user\_id\_1** já demorava apenas 0,061ms (figura 14).

Node Type	Count	Time ↓	
▼ Bitmap Index Scan	2	3s 122ms	96%
#15 Bitmap Index Scan		3s 122ms	96%
#14 Bitmap Index Scan		0,061ms	0%
> Limit	1	80,8ms	2%
> Index Scan	1	36,6ms	1%
> Merge Join	2	4,39ms	0%

Figura 14: Algumas das operações usadas na execução do plano

Como agora cada lado do **OR** lida com uma só coluna, foi necessário criar um índice apenas na coluna **user\_id\_2** para tornar a leitura desta coluna tão eficiente como a primeira:

```
IF NOT EXISTS CREATE INDEX idx_friendship_user_id_2 ON friendship (user_id_2);
```

Com o novo índice obtivemos um tempo de resposta de 2ms, com um pequeno gargalo na leitura da tabela **game**. Acreditamos que seja possível diminuir ainda mais o tempo de resposta, possivelmente com a criação de um ou dois índices, no entanto, a diferença para o tempo atual muito dificilmente iria justificar o expectável impacto na carga transacional então decidimos não seguir esse caminho e terminar aqui a otimização desta *query*.

Plano: <https://explain.dalibo.com/plan/2ad3eddcceh78gc2>

### 3.2.2. Impacto na carga transacional

Da nossa análise destas otimizações o que pode afetar o desempenho do *workload* é o índice criado na tabela **friendship**. Analisando as transações do mesmo, a única que utiliza essa tabela é a **newFriendship**, em que é apenas feito um **INSERT** na tabela, portanto é esperado que o impacto deste índice seja baixo.

Otimização	Tempo Resposta (s)	Trans. Throughput (txn/s)
Base	31,152	4200
Query	3,251	4200
Index	0,002	4000

Tabela 1: Comparação de otimizações

## 3.3. Query 3

### 3.3.1. Otimizações

```
SELECT count(*) AS total, count(DISTINCT id) AS unique
FROM (
  (
    SELECT p.name, u.id AS id
    FROM publisher p
    JOIN games_publishers gp ON gp.publisher_id = p.id
    JOIN game g ON g.id = gp.game_id
    JOIN library l ON l.game_id = g.id
    JOIN users u ON u.id = l.user_id
  )
  UNION ALL
  (
    SELECT d.name, u.id AS id
    FROM developer d
    JOIN games_developers gd ON gd.developer_id = d.id
    JOIN game g ON g.id = gd.game_id
    JOIN library l ON l.game_id = g.id
    JOIN users u ON u.id = l.user_id
  )
)
WHERE name LIKE 'Ubisoft';
```

Plano: <https://explain.dalibo.com/plan/4e78539dd24f83d2>

O tempo de resposta inicial rondava os 15,4 segundos. Os maiores gargalos são operações de **SCAN** e **JOIN** com a tabela **library**, e ainda um **SCAN** na tabela **users**.

Começamos novamente por procurar otimizações na *query*, antes de tentar outras estratégias que modifiquem a base de dados e afetem outras vertentes.

Detalhe: Como a *query* de pesquisar um prefixo foi alterada pelos professores, para pesquisar um nome exato de uma empresa, alteramos o filtro para **WHERE name = 'Ubisoft'**, apenas por uma questão de legibilidade/preferência, já que o PostgreSQL acabaria por tratar o **LIKE** como um = por não haver um padrão no nome a pesquisar.

Removemos 2 **JOIN** de cada *subquery* (**JOIN users** e **JOIN game**), pois notamos que não precisávamos de nenhuma informação dessas tabelas que não conseguíssemos obter através dos restantes **JOIN**. Precisávamos apenas do nome da empresa e do **id** de cada **user** que possuía um jogo da mesma, informações essas que estão nas tabelas **publisher/developer** e **library**,

necessitando apenas das tabelas **games\_publishers/games\_developers** para fazer a ligação entre a **library** e a respetiva tabela.

Query atualizada:

```
SELECT count(*) AS total, count(DISTINCT id) AS unique
FROM (
  (
    SELECT p.name, l.user_id AS id
    FROM publisher p
    JOIN games_publishers gp ON gp.publisher_id = p.id
    JOIN library l ON l.game_id = gp.game_id
  )
  UNION ALL
  (
    SELECT d.name, l.user_id AS id
    FROM developer d
    JOIN games_developers gd ON gd.developer_id = d.id
    JOIN library l ON l.game_id = gd.game_id
  )
)
WHERE name = 'Ubisoft';
```

Plano: <https://explain.dalibo.com/plan/46h3d9d449b3d9f7>

Obtivemos então uma média de tempo de 11,3 segundos, sendo a grande diferença os 3,5 segundos que se perdia a ler a tabela **users**. O gargalo nas duas leituras da tabela **library** continua a existir, pelo que o próximo passo foi tentar passar a ler apenas uma vez essa tabela.

Decidimos mover o **JOIN library** para a *query* principal, passando a obter primeiro todas as empresas e só no final percorrer a **library** à procura dos **users** de cada uma.

Em sentido inverso, movemos a filtragem pelo nome da empresa para cada uma das *subqueries*, para ser executada mais cedo e assim produzir resultados menores para melhorar o tempo do **UNION ALL** ainda na *subquery* e do **JOIN library** na *query* principal.

Novo estado da *query*:

```
SELECT count(*) AS total, count(DISTINCT l.user_id) AS unique
FROM (
  (
    SELECT gp.game_id AS id
    FROM publisher p
    JOIN games_publishers gp ON gp.publisher_id = p.id
    WHERE p.name = 'Ubisoft'
  )
  UNION ALL
  (
    SELECT gd.game_id AS id
    FROM developer d
    JOIN games_developers gd ON gd.developer_id = d.id
    WHERE d.name = 'Ubisoft'
  )
) AS g
JOIN library l ON l.game_id = g.id;
```

Plano: <https://explain.dalibo.com/plan/aef66a8b4ag4b4cb>

Com esta *query* obtivemos uma melhoria no tempo, o que era esperado pelos motivos referidos, passando a registar uma média de 6,2 segundos. Sendo o gargalo ainda o mesmo, que acreditávamos ser facilmente resolvido por um índice.

Ainda insatisfeitos com a *query* atual, tentamos procurar mais alguma otimização. Notamos que algum tempo ainda era gasto nas operações de **SORT**, **MERGE** e **AGGREGATE**. Procuramos reduzir os

dados que chegam a esta fase da execução, uma vez que apenas necessitamos da coluna **user\_id** para o resultado pretendido.

Invertamos a ordem das tabelas no **JOIN** da **library** com o resultado da *subquery*, fazendo com que o método utilizado para as juntar fosse um **Semi Join**, utilizado nas operações **IN** e **EXISTS** pelo PostgreSQL, fazendo assim com que apenas as colunas da tabela **library** fossem guardadas para as operações seguintes.

Query atualizada:

```
SELECT count(*) AS total, count(DISTINCT l.user_id) AS unique
FROM library l
WHERE l.game_id IN (
    SELECT gp.game_id
    FROM publisher p
    JOIN games_publishers gp ON gp.publisher_id = p.id
    WHERE p.name = 'Ubisoft'

    UNION ALL

    SELECT gd.game_id
    FROM developer d
    JOIN games_developers gd ON gd.developer_id = d.id
    WHERE d.name = 'Ubisoft'
);
```

Plano: <https://explain.dalibo.com/plan/42b3g503g5a705h7>

Node Type	Count		Time ↓
▼ Seq Scan	5	2s 663ms	43%
#5 Seq Scan		2s 492ms	41%
#15 Seq Scan		105ms	2%
#11 Seq Scan		58,3ms	1%
#13 Seq Scan		4,13ms	0%
#9 Seq Scan		3,99ms	0%
▼ Hash Semi Join	1	2s 607ms	43%
#4 Hash Semi Join		2s 607ms	43%
> Gather Merge	1	390ms	6%
> Sort	1	295ms	5%
> Aggregate	1	166ms	3%
> Hash Join	2	9,58ms	0%
> Hash	3	0,32ms	0%
> Append	1	0,007ms	0%

Figura 15: Operações usadas na execução do plano

Index	Count	Time ↓
No index used		

Figura 16: Índices usados na execução do plano

Através do **EXPLAIN ANALYZE** pudemos verificar que 84% do tempo continuava a ser gasto no *Parallel Sequential Scan* e posteriormente no **Hash Semi Join** na tabela **library**, e que não eram usados quaisquer índices durante toda a execução.

Passamos então para a análise dos índices, começando claro pela tabela **library** e pelas colunas **game\_id** e **user\_id** (o inverso da *primary key*), otimizando assim primeiramente o *matching* do **game\_id** com os id's obtidos na *subquery*, e de seguida a obtenção do **user\_id** correspondente a um *match* para usar no **SELECT**.

```
CREATE INDEX idx_library_game_user ON library(game_id, user_id);
```

Plano: <https://explain.dalibo.com/plan/088f2328ec9gge2b>

Conseguimos uma boa melhoria no tempo de resposta, passando para 1,7 segundos.

```
CREATE INDEX idx_developer_name ON developer(name);
CREATE INDEX idx_publisher_name ON publisher(name);
CREATE INDEX idx_gd_devid_gameid ON games_developers(developer_id, game_id);
CREATE INDEX idx_gp_pubid_gameid ON games_publishers(publisher_id, game_id);
```

A criação destes índices ajuda a remover os *sequential scans* que temos na *query* mas como estavam só em média com 4ms entre cada *sequential scan* não valia a pena visto que o **SORT** continuava a ser o principal gargalo.

Mexendo com os parâmetros de configuração do PostgreSQL, testamos vários valores de *work memory*, desde 4MB a 1024MB, obtendo os melhores tempos com 256MB.

```
SET work_mem = '256MB';
```

Plano: <https://explain.dalibo.com/plan/b23c961g916ecfg8>

Evolução da query 3 com diferentes work_mem	
work_mem	Tempo Resposta (s)
4MB	1,819
8MB	1,652
16MB	1,682
32MB	1,724
64MB	1,712
128MB	1,537
256MB	1,534
512MB	1,536
1024MB	1,543

Figura 17: Análise dos tempos de resposta

Não estávamos a conseguir melhorar o tempo, então decidimos tentar uma abordagem nova. Criámos uma nova tabela persistida na base de dados para guardar os dados necessários para a

*query*, de forma a não ter de calcular sempre que executámos a *query*. Populámos previamente a tabela com os dados existentes na base de dados na altura da criação:

```
-- Create the table
CREATE TABLE company_users_count (
  company_name VARCHAR PRIMARY KEY,
  total_users INTEGER NOT NULL DEFAULT 0,
  unique_users INTEGER NOT NULL DEFAULT 0
);

-- Populate the table from existing data
INSERT INTO company_users_count (company_name, total_users, unique_users)
SELECT
  g.name AS company_name,
  COUNT(*) AS total_users,
  COUNT(DISTINCT l.user_id) AS unique_users
FROM (
  SELECT p.name, gp.game_id
  FROM publisher p
  JOIN games_publishers gp ON gp.publisher_id = p.id

  UNION ALL

  SELECT d.name, gd.game_id
  FROM developer d
  JOIN games_developers gd ON gd.developer_id = d.id
) AS g
JOIN library l ON l.game_id = g.game_id
GROUP BY g.name;
```

Assim pudemos ajustar a *query* para ler os dados desta nova tabela, não precisando de calcular nenhuns dados em tempo real, diminuindo drasticamente o tempo de resposta.

Versão final da query:

```
SELECT cuc.total_users, cuc.unique_users
FROM company_users_count cuc
WHERE cuc.company_name = 'Ubisoft';
```

<https://explain.dalibo.com/plan/e196903ec67g9a3b>

Abandonámos assim todos os índices criados nas alterações anteriores e também as alterações nas configurações de memória, e ficamos com um tempo de resposta de cerca de 0,01 milissegundos.

### 3.3.2. Impacto na carga transaccional

Após estarmos contentes com as otimizações conseguidas nesta *query*, fomos analisar o impacto da nova tabela na carga transaccional. Era de prever que fosse piorar a performance do caso de uso **BuyGame**, o que se verificou.

Adicionamos um *trigger* à tabela **library** para que a nova tabela seja devidamente atualizada sempre que um **user** compra um jogo.



```

CREATE OR REPLACE FUNCTION update_company_users_count() RETURNS TRIGGER AS $$
BEGIN
    WITH companies AS (
        SELECT p.name AS company_name
        FROM publisher p
        JOIN games_publishers gp ON gp.publisher_id = p.id
        WHERE gp.game_id = NEW.game_id
        UNION ALL
        SELECT d.name AS company_name
        FROM developer d
        JOIN games_developers gd ON gd.developer_id = d.id
        WHERE gd.game_id = NEW.game_id
    ),
    company_counts AS (
        SELECT company_name, COUNT(*) AS appearances
        FROM companies
        GROUP BY company_name
    ),
    new_uniques AS (
        SELECT cc.company_name
        FROM company_counts cc
        WHERE NOT EXISTS (
            SELECT 1
            FROM (
                SELECT p.name AS company_name, gp.game_id
                FROM publisher p
                JOIN games_publishers gp ON gp.publisher_id = p.id
                WHERE p.name = cc.company_name
                UNION ALL
                SELECT d.name AS company_name, gd.game_id
                FROM developer d
                JOIN games_developers gd ON gd.developer_id = d.id
                WHERE d.name = cc.company_name
            ) company_games
            JOIN library l ON l.game_id = company_games.game_id
            WHERE l.user_id = NEW.user_id
            AND l.game_id <> NEW.game_id -- exclude the game just added
        )
    )
    UPDATE company_users_count cuc
    SET
        total_users = cuc.total_users + cc.appearances,
        unique_users = cuc.unique_users +
            (CASE WHEN cuc.company_name IN (SELECT company_name FROM new_uniques) THEN 1
        ELSE 0 END)
    FROM company_counts cc
    WHERE cuc.company_name = cc.company_name;

    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

-- Ensure the trigger doesn't get created multiple times
DROP TRIGGER IF EXISTS trg_update_company_users_count ON library;

CREATE TRIGGER trg_update_company_users_count
AFTER INSERT ON library
FOR EACH ROW
EXECUTE FUNCTION update_company_users_count();

```

```

Response time per function (ms)
-----
AddPlaytime = 2.487
ReviewGame = 2.310
BuyGame = 267.882
NewFriendship = 2.432
NewGame = 2.123
NewUser = 2.252
GameInfo = 1.762
GameReviews = 3.941
UserInfo = 3.259
RecentGamesPerTag = 52.815
SearchGames = 4.710

Overall metrics
-----
throughput (txn/s) = 467.000
response time (ms) = 34.253
abort rate (%) = 0.001

```

Figura 18: Análise da *workload*

Como os resultados obtidos foram bastante piores do que era esperado, decidimos que o melhor para manter o equilíbrio entre o desempenho da carga analítica e da carga transacional decidimos voltar atrás na otimização ao criar uma tabela nova.

Assim, removemos a tabela, o *trigger* e a função, voltando à *query* anterior com um tempo de resposta de cerca de 1,5 segundos, com o índice referido na tabela **library**.

### 3.4. Query 4

Na *query* original, geramos *bins* com intervalos fixos de 12 horas desde 2003 e a filtrar apenas os *bins* posteriores a 2020. Por si só isto não é um problema, pois é uma operação rápida, no entanto, ao fazer o **JOIN** com a *library* de todos os *bins* mais tempo é perdido a organizar a tabela inteira e depois a comparar com mais *bins* do que os necessários, acabando imensos por nem sequer fazer *match* com nenhuns dados da tabela **library**.

```

WITH bins AS (
    SELECT generate_series('2003-09-12', now(), '12 hours') AS bin
)
SELECT bin,
    count(*) FILTER (WHERE buy_price > 0) AS paid_copies,
    count(*) FILTER (WHERE buy_price = 0) AS free_copies,
    sum(buy_price) AS money_generated
FROM bins
JOIN library ON bin = date_bin('12 hours', added_date, '2020-01-01')
WHERE bin >= '2020-01-01'
GROUP BY bin
ORDER BY bin;

```

Plano: <https://explain.dalibo.com/plan/cedfdagfg86c4703>

Tempo original: 1m38s

### 3.4.1. 1ª Alteração

Ao filtrar diretamente a tabela **library** pela data (`added_date >= '2020-01-01'`) conseguimos reduzir a dimensão dos dados com que a *query* tem de lidar (**Sort**, **Group**, **Count**,...). Além disso, a lógica foi reformulada para gerar os *bins* apenas uma vez e apenas para os dados necessários, removendo o esforço de fazer **JOIN**. O principal gargalo passou assim a estar no **Sort** e no **Merge**, derivados das operações de **ORDER BY** e **GROUP BY**. É de notar também que o *scan* na **library** também consome bastante tempo de execução da *query*.

```
SELECT
    date_bin('12 hours', added_date, '2003-09-12') AS bin,
    count(*) FILTER (WHERE buy_price > 0) AS paid_copies,
    count(*) FILTER (WHERE buy_price = 0) AS free_copies,
    sum(buy_price) AS money_generated
FROM library
WHERE added_date >= '2020-01-01'
GROUP BY bin
ORDER BY bin;
```

Novo plano: <https://explain.dalibo.com/plan/642760d0ecbfgf8b#plan>

Novo tempo: 32s

### 3.4.2. 2ª Alteração

Para otimizar a leitura da tabela **library** tentamos vários índices, com várias combinações de colunas, incluindo índices funcionais e condicionais. No entanto, a maioria dos índices revelou-se ineficaz ou não foram utilizados pelo plano de execução, ou não trouxeram melhorias significativas no desempenho da *query*.

Índices testados:

```
CREATE INDEX IF NOT EXISTS idx_library_added_date_price ON library (added_date,
buy_price);
CREATE INDEX IF NOT EXISTS idx_library_added_date_price_paid ON library (added_date,
buy_price) WHERE buy_price > 0;
CREATE INDEX IF NOT EXISTS idx_library_datebin_buyprice
ON library (
    date_bin('12:00:00', added_date, '2020-01-01'),
    buy_price
);
```

O único índice que se revelou efetivo e que foi utilizado pelo *planner*:

```
CREATE INDEX IF NOT EXISTS idx_library_added_date ON library (added_date);
```

Este índice permitiu que fosse utilizado um *Parallel Index Scan*, o que contribuiu para reduzir o tempo de leitura da tabela.

Novo plano: <https://explain.dalibo.com/plan/42c4gcf26d1244a4>

Novo tempo: 10s

### 3.4.3. 3ª Alteração

Após os testes com os índices, verificamos também alterações nos parâmetros de execução do PostgreSQL, com o intuito de melhorar o desempenho da *query* sem alterar a lógica da agregação.

**Parâmetros testados:**

```
SET max_parallel_workers
SET max_parallel_workers_per_gather
```

Tentamos forçar a utilização de mais *workers* em operações como *Parallel Index Scan*, *Gather Merge* e *Sort* para ajudar a *query* que lê várias linhas da **library**, e com o uso de mais *workers* a ideia era poder distribuir esta carga. Mas apesar de ativar mais *workers*, em alguns casos, o *overhead* de preparação dos *workers* e o custo de junção dos resultados (em *Gather Merge*) acabavam por anular os benefícios de ativar mais *workers*.

```
SET parallel_setup_cost = 0;
SET parallel_tuple_cost = 0;
```

Tentamos também reduzir artificialmente o custo estimado de preparar e transmitir tuplos entre *workers*, forçando o *planner* a escolher planos paralelos. Mas apesar de induzir mais paralelismo, nem sempre era o melhor para a *query*, pois acabava por usar *Gather Merge* com *external merge sort* em disco, o que tornava a execução mais lenta do que em planos não paralelos.

```
SET work_mem
```

Também testamos diferentes capacidades de memória disponíveis, como por exemplo, 256MB, 512MB, 1024MB, etc...

Com o objetivo de tentar evitar o uso do *external merge sort* e tentar forçar o uso de *quick sort* ou *in-memory aggregation*. Com mais memória o *planner* acabou por mudar para *HashAggregate*, mas acabava por causar *spikes* de uso de memória e de uso intenso do disco temporário, o que em vez de melhorar o tempo de execução, acabava por piorar.

```
SET enable_hashagg = off;
SET enable_sort = on;
```

Por fim também acabamos por experimentar forçar o *planner* a usar *GroupAggregate* em vez de *HashAggregate*, para evitar o uso massivo de memória em operações intermediárias em disco e também garantir que o *planner* não evita o *Sort* em favor de alternativas menos previsíveis. Sendo que nesta configuração que por *default* o PostgreSQL já vem por definição, acaba por ser o que dá melhores resultados pois mantém a consistência do *Sort* -> *GroupAggregate* e também mantém o uso de *Parallel Index Scan* com agregação em fluxo.

### 3.4.4. 4ª Alteração

Uma alternativa que testámos foi a utilização de CTEs para tentar melhorar a legibilidade e reuso de expressões, principalmente a função *date\_bin(...)*, aplicada repetidamente sobre a coluna *added\_date*.

```
WITH binned_library AS (
  SELECT
    date_bin('12 hours', added_date, '2020-01-01') AS bin,
    buy_price
  FROM library
  WHERE added_date >= '2020-01-01'
)
SELECT
  bin,
  count(*) FILTER (WHERE buy_price > 0) AS paid_copies,
  count(*) FILTER (WHERE buy_price = 0) AS free_copies,
  sum(buy_price) AS money_generated
FROM binned_library
GROUP BY bin
ORDER BY bin;
```

Plano: <https://explain.dalibo.com/plan/14d2bg930d12813g>

No entanto, o plano gerado mostra que o *planner* anulou a CTE, ou seja, tratou-a como parte da *query* principal, sem a materializar. Como resultado, a função *date\_bin(...)* continua a ser avaliada em tempo de execução dentro do *Sort* e do *GroupAggregate*, o que mantém o custo computacional e de disco praticamente igual ao que tínhamos antes.

Além disso, o plano evidencia o uso de *GroupAggregate* com *Gather Merge* e *Sort Method: external merge*, com uso intensivo de disco temporário (239MB por *worker*), tal como nas versões sem CTE.

Portanto, esta tentativa não trouxe melhorias de desempenho, e apenas resultou numa separação lógica sem efeito prático na eficiência da execução.

### 3.4.5. Impacto na carga transacional

Como temos a criação de apenas um índice afetamos ligeiramente a carga transacional. Analisando o *workload* com estes índices o impacto não é muito grande.

## 3.5. Query 5

```
SELECT u.id, u.username, u.country, (l.playtime / 60)::int AS hours, l.added_date
FROM game g
JOIN library l ON l.game_id = g.id
JOIN users u ON u.id = l.user_id
WHERE g.id = 730
      AND l.playtime > 0
      AND NOT u.vac_banned
ORDER BY 4 DESC, 5 DESC
LIMIT 1000;
```

Plano: <https://explain.dalibo.com/plan/hb4cedf5f860d638>

Analisando o plano deparamos que o maior gargalo é no *parallel scan* efetuado na **library** e também reparamos que a tabela **game** só é usado por causa do **game\_id**, que já é obtida na **library** por isso removemos a tabela **game** da operação.

```
CREATE INDEX IF NOT EXISTS index_library_game_playtime ON library(game_id, playtime
DESC) WHERE playtime > 0;
CREATE INDEX IF NOT EXISTS index_users_vac_banned ON users(id) WHERE NOT vac_banned;
```

```
SELECT u.id, u.username, u.country, (l.playtime / 60)::int AS hours, l.added_date
FROM library l
JOIN users u ON u.id = l.user_id AND NOT u.vac_banned
WHERE l.game_id = 730 AND l.playtime > 0
ORDER BY l.playtime DESC, l.added_date DESC
LIMIT 1000;
```

Novo plano: <https://explain.dalibo.com/plan/04c2a669048g4f42>

Com este plano o gargalo está num *gather merge*, isto acontece porque os *parallel workers* têm que juntar os seus resultados mas como a quantidade não compensa o uso de diferentes *workers* o *overhead* de ter vários é muito grande para esta *query*.

```
SET max_parallel_workers_per_gather = 0
```

Sem *parallel workers* não precisamos de um *gather* tirando o maior gargalo da *query*.

### **3.5.1. Impacto na carga transacional**

Como temos a criação de dois índices afetamos ligeiramente a carga transacional. Analisando o *workload* com estes índices o impacto não é muito grande sendo que as operações aumentam ligeiramente os seus tempos médios (cerca de 0.5ms).

## 4. Conclusão

O desenvolvimento deste relatório documenta o trabalho realizado durante a configuração, otimização e avaliação do *benchmark* baseado num excerto do *dataset* da loja online Steam. Ao longo deste processo foi possível consolidar os conceitos abordados nas aulas da UC de Administração de Bases de Dados e aplicar esses conhecimentos num caso prático e realista.

A aplicação de várias técnicas de otimização, como a criação de índices, a materialização de dados, a reformulação de *queries* e a parametrização do sistema, permitiu melhorar significativamente o desempenho das interrogações analíticas e das operações transacionais em PostgreSQL. Foram também exploradas alterações no código SQL e Java para potenciar o desempenho do sistema.

Em suma, o trabalho desenvolvido permitiu-nos aprofundar a compreensão sobre os desafios de administração e otimização de bases de dados, bem como adquirir experiência prática na avaliação do impacto de diferentes estratégias de melhoria de desempenho.