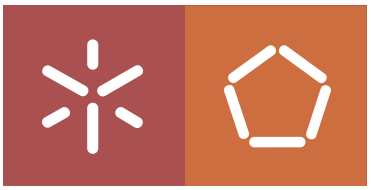

Universidade do Minho
Escola de Engenharia

# Administração de Bases de Dados

Relatório Trabalho Prático

Grupo 7

---

MEI - 1º Ano - 2º Semestre

**Trabalho realizado por:**

PG55945 - Gonçalo Marinho
PG55947 - Henrique Vaz
PG57887 - Luís Caetano
PG57891 - Maya Gomes
PG55987 - Mike Pinto

Braga, 13 de maio de 2025

# Índice

# 1. Introdução

Este relatório descreve o trabalho prático desenvolvido no âmbito da unidade curricular de Administração de Bases de Dados. O trabalho tem como principal objetivo configurar, otimizar e avaliar o desempenho de um sistema de base de dados suportando operações transacionais e analíticas, inspiradas no contexto de uma loja online semelhante à *Steam*. A execução e análise foram realizadas numa infraestrutura de referência, utilizando uma máquina virtual na *Google Cloud*, e focaram-se na melhoria de desempenho através de várias estratégias de otimização.

## 1.1. Contexto e Desafios

O trabalho incide sobre um sistema de base de dados que simula funcionalidades típicas de uma plataforma de distribuição digital de videojogos. O *benchmark* utilizado inclui tanto operações transacionais, como registo de novos utilizadores e jogos, compra de jogos, críticas e estatísticas de jogo, como consultas analíticas que envolvem agregações e análise temporal dos dados. Os principais desafios incluem:

- Simulação realista de carga: executar múltiplas transações em simultâneo (16 clientes) e consultas analíticas com elevado custo computacional.
- Equilíbrio entre desempenho e fidelidade funcional: garantir que as otimizações não comprometem a lógica original das operações.
- Limitações de recursos: trabalhar dentro dos limites de *CPU*, *RAM* e armazenamento definidos pela infraestrutura de referência.
- Impacto cruzado das otimizações: alterações feitas para melhorar o desempenho transacional podem impactar negativamente a componente analítica, e vice-versa.
- Gestão eficiente de recursos *cloud*: minimizar custos sem comprometer a validade dos resultados, explorando técnicas como *snapshots*, armazenamento externo, e instâncias *spot*.

## 1.2. Metodologia

A abordagem seguida neste trabalho pode ser dividida em várias etapas:

- Configuração do Ambiente: Instalação do PostgreSQL nativamente (sem recurso a Docker) numa máquina virtual N2 da *Google Cloud* e preparação do dataset (em formato CSV) e carga inicial na base de dados, utilizando os *scripts* fornecidos no repositório.
- Execução da Configuração de Referência: Avaliação inicial do sistema sem otimizações, para estabelecer uma linha de base de desempenho.
- Otimização da Carga Analítica: Identificação de consultas analíticas críticas e análise do plano de execução. Introdução de otimizações como partições, índices especializados e reformulação de queries.
- Otimização da Carga Transacional: Aplicação de estratégias de redundância (criação de índices apropriados, vistas materializadas quando viável, etc). Refatoração de código SQL/Java para melhorar a eficiência de execução e ajuste de parâmetros de configuração do *PostgreSQL*, como `buffers` e paralelismo.
- Automatização e Monitorização: Criação de *scripts* para automatizar a execução do *benchmark*. Recolha de métricas de desempenho. Repetição dos testes para garantir consistência e reprodutibilidade dos resultados.
- Análise Comparativa: Comparação dos resultados obtidos antes e depois das otimizações, com ênfase no débito transacional e tempo de resposta médio das consultas analíticas. Discussão do impacto de cada tipo de otimização e *trade-offs* observados.

# 2. Carga analítica no PostgreSQL

## 2.1. Interrogação analítica 1

```sql
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 análise tem como objetivo identificar o jogo mais vendido para cada ano presente na base de dados. Esta tarefa é realizada utilizando uma estrutura de *subqueries* aninhadas e um ranking baseado em vendas. Os resultados são então ordenados pelo ano em ordem decrescente, apresentando uma evolução temporal dos jogos mais populares.

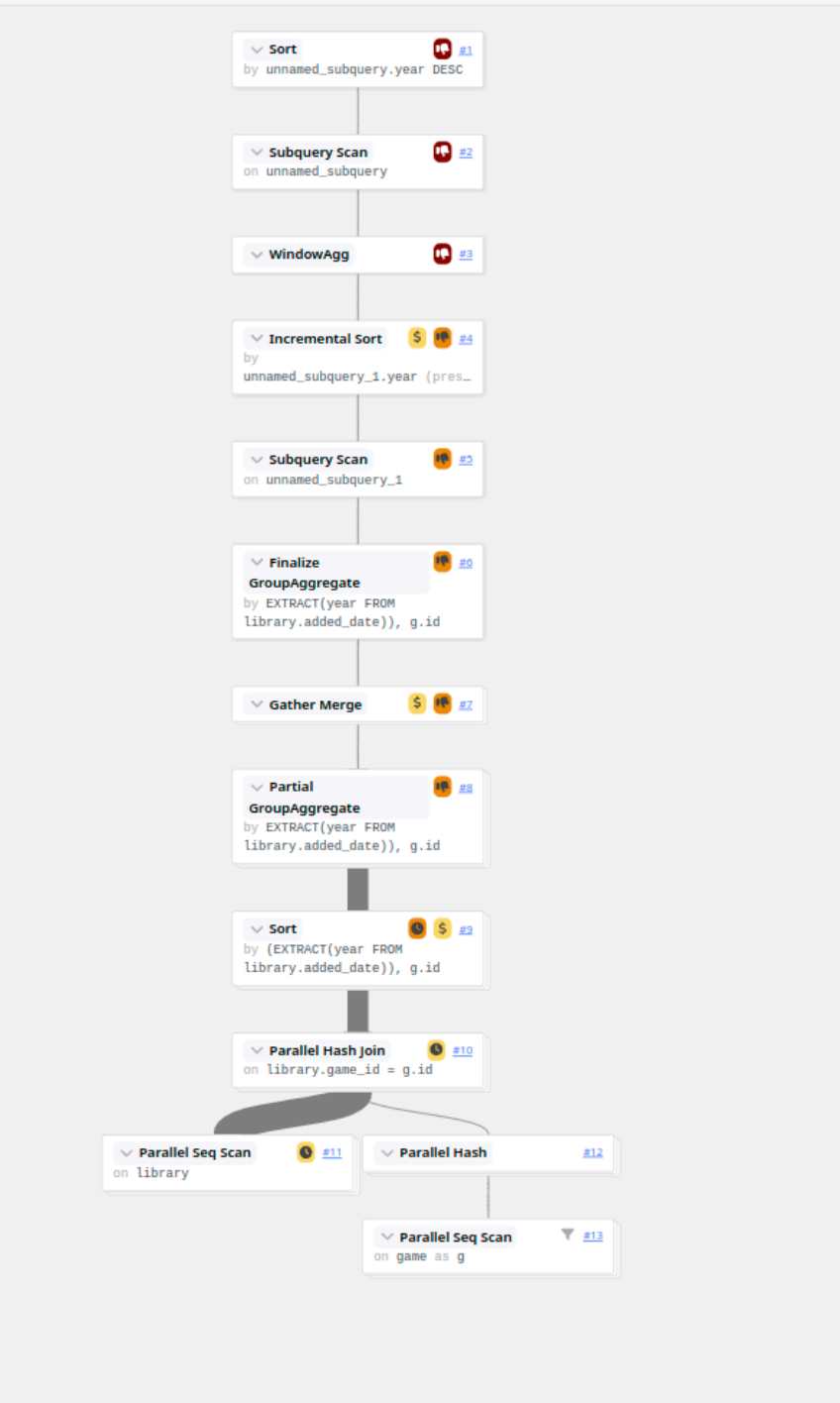


Figura 1: Plano de execução da *baseline* da *query* 1 sem otimizações

O plano de execução inicial revelou vários problemas críticos: o tempo de execução total era de 71.156 segundos (aproximadamente 1 minuto e 11 segundos), indicando uma *query* extremamente pesada e ineficiente. A expressão `EXTRACT(year FROM library.added_date)` era calculada repetidamente para cada registo, representando um processamento redundante que consumia recursos computacionais desnecessariamente. A junção entre as tabelas `library` e `game` através de `library.game_id = g.id` apresentava baixo desempenho devido à ausência de índices apropriados, forçando o sistema a realizar leituras sequenciais completas em ambas as tabelas. As operações de ordenação e agrupamento consumiam recursos significativos devido à grande quantidade de dados manipulados nestas operações, resultando num uso excessivo de memória e disco. O diagnóstico inicial apontou claramente para a necessidade de pré-computar valores recorrentes, adicionar índices adequados e simplificar a estrutura

da *query* para reduzir o processamento redundante e melhorar o fluxo de dados através do pipeline de execução.

#### 2.1.1. 1ª Fase de Otimização

Na primeira fase de otimização, foram implementadas três melhorias estruturais:

```
ALTER TABLE library
ADD COLUMN added_year INT GENERATED ALWAYS AS (EXTRACT(YEAR FROM added_date))
STORED;
CREATE INDEX idx_library_year_added ON library (added_year);
CREATE INDEX idx_library_game_id ON library (game_id);
```

E a *query* foi reformulada para:

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

A criação de uma coluna materializada `added_year` eliminou a necessidade de cálculos repetitivos da função `EXTRACT()`, transformando uma operação computacional intensiva numa simples leitura de dados já processados. A coluna gerada é mantida automaticamente pelo sistema, garantindo consistência com a coluna original `added_date`. O índice `idx_library_year_added` otimizou as operações de filtragem e agrupamento baseadas no ano, acelerando a cláusula `PARTITION BY l.added_year`. O índice `idx_library_game_id` melhorou o desempenho da operação de junção entre as tabelas `library` e `game`. A reestruturação da *query* eliminou uma *subquery* desnecessária.

O plano de execução após estas otimizações mostrou uma redução significativa no tempo de execução para 18.332 segundos, representando uma melhoria de aproximadamente 74% em relação ao valor inicial. Esta redução dramática confirma a eficácia das otimizações implementadas e valida a análise inicial dos pontos críticos.

No entanto, ainda foram identificados dois pontos críticos persistentes no plano de execução. O *Partial Hash Aggregate* na operação de agrupamento por `l.added_year, g.id` continuava a consumir recursos significativos, indicando que as operações de agrupamento ainda representavam um gargalo substancial. Este comportamento sugeria que o sistema poderia estar a utilizar espaço em disco para as operações de *hash* devido a limitações de memória disponível para processamento. O *Parallel Seq Scan* na tabela `library` ainda representava um custo elevado, indicando que os índices não estavam a ser utilizados de forma otimizada em todas as partes da query. Isto poderia ser devido a estatísticas desatualizadas ou insuficientes sobre a distribuição dos dados, levando o otimizador a fazer escolhas subótimas.

#### 2.1.2. 2ª Fase de Otimização

Para melhorar a estimativa estatística e a seleção de planos pelo otimizador, foram implementadas as seguintes alterações:

```
ALTER TABLE library ALTER COLUMN added_year SET STATISTICS 1000;
ALTER TABLE game ALTER COLUMN id SET STATISTICS 1000;
```

Estas alterações aumentaram a quantidade de estatísticas recolhidas pelo PostgreSQL para as colunas críticas, permitindo ao otimizador fazer melhores estimativas sobre a distribuição dos dados, resultando em planos de execução mais eficientes. O otimizador de *queries* do PostgreSQL depende fortemente das estatísticas sobre a distribuição de valores nas colunas para tomar decisões informadas sobre como executar uma *query*. Quando estas estatísticas são insuficientes ou imprecisas, o otimizador pode fazer escolhas subótimas, como não utilizar um índice disponível quando seria vantajoso fazê-lo. Ao incrementar o valor de estatísticas para 1000 (o padrão é normalmente 100), o sistema de gestão de base de dados recolhe amostras maiores e mais detalhadas das colunas, melhorando significativamente a precisão das estatísticas utilizadas no planeamento de *queries*.

O tempo de execução reduziu para 17.442 segundos, uma melhoria de aproximadamente 5% em relação à fase anterior. Esta melhoria, embora mais modesta, confirma que a qualidade das estatísticas disponíveis para o otimizador tem um impacto real no desempenho das *queries*. No entanto, os mesmos pontos críticos identificados anteriormente ainda persistiam, embora com ligeira melhoria, indicando que era necessário abordar outros aspetos do sistema para obter melhorias mais significativas.

#### 2.1.3. 3ª Fase de Otimização

Nesta fase, foram realizadas operações de manutenção nas tabelas para recolher estatísticas atualizadas e reorganizar fisicamente os dados:

```
VACUUM ANALYZE library;
VACUUM ANALYZE game;
CREATE INDEX idx_library_year_game ON library (added_year, game_id);
CLUSTER library USING idx_library_year_game;
```

A operação `VACUUM ANALYZE` remove linhas mortas e atualiza as estatísticas do otimizador, permitindo melhores decisões de planeamento. Ao longo do tempo, as operações de inserção, atualização e eliminação deixam "espaço morto" nas tabelas, que pode degradar o desempenho. Além disso, a distribuição dos dados pode mudar significativamente, tornando as estatísticas existentes menos representativas da realidade atual. O `VACUUM ANALYZE` resolve ambos os problemas, recuperando espaço e atualizando as estatísticas para refletir o estado atual dos dados.

A reordenação física dos dados na tabela `library` segundo o índice composto `idx_library_year_game` melhora significativamente a localidade dos dados, reduzindo o tempo necessário para aceder a registos relacionados. Quando os dados estão fisicamente organizados na mesma ordem em que são frequentemente acedidos, o sistema pode ler blocos contíguos de disco, aproveitando melhor a cache do sistema operativo e do hardware. Esta otimização é particularmente eficaz para tabelas de grande dimensão que são acedidas principalmente em determinadas sequências previsíveis, como é o caso.

O tempo de execução reduziu para 12.734 segundos, representando uma melhoria de aproximadamente 27% em relação à fase anterior. O plano de execução mostrou redução significativa no tempo de `Sort` e `HashAggregate`, mas o `Partial Hash Aggregate` ainda era um ponto crítico devido à limitação de memória.

#### 2.1.4. 4ª Fase de Otimização

Para resolver o problema de memória no processamento do `HashAggregate`, o parâmetro `work_mem` foi ajustado:

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

Aumentar o `work_mem` de seu valor padrão (tipicamente 4MB) para 256MB permite que as operações de *hash* e ordenação sejam realizadas em memória em vez de utilizar disco, o que é substancialmente mais rápido. Este ajuste é particularmente eficaz para operações de `HashAggregate` que processam grandes volumes de dados.

O tempo de execução reduziu para 10.181 segundos, uma melhoria de aproximadamente 20% em relação à fase anterior. O plano de execução revelou uma mudança significativa: as operações de `hash aggregate` agora eram realizadas completamente em memória, evitando a escrita para disco. Isto confirma a hipótese de que a falta de memória disponível para as operações de *hash* era um fator limitante no desempenho. No entanto, o `Parallel Seq Scan` na tabela `library` continuava a representar um custo elevado, indicando que havia ainda margem para melhoria através de otimizações adicionais.

#### 2.1.5. 5ª Fase de Otimização

Para melhorar o paralelismo e distribuir melhor a carga de trabalho entre os núcleos disponíveis, foram ajustados os parâmetros:

```
SET max_parallel_workers_per_gather = 4;
SET parallel_tuple_cost = 0.1;
SET parallel_setup_cost = 1000;
SET min_parallel_table_scan_size = '1MB';
SET min_parallel_index_scan_size = '1MB';
```

O tempo de execução final reduziu para 7.776 segundos, representando uma melhoria de aproximadamente 24% em relação à fase anterior e uma redução de aproximadamente 89% em relação ao tempo de execução original. O plano de execução mostrou um aumento no número de *workers* paralelos de 2 para 4, e uma distribuição mais uniforme do trabalho entre os mesmos.

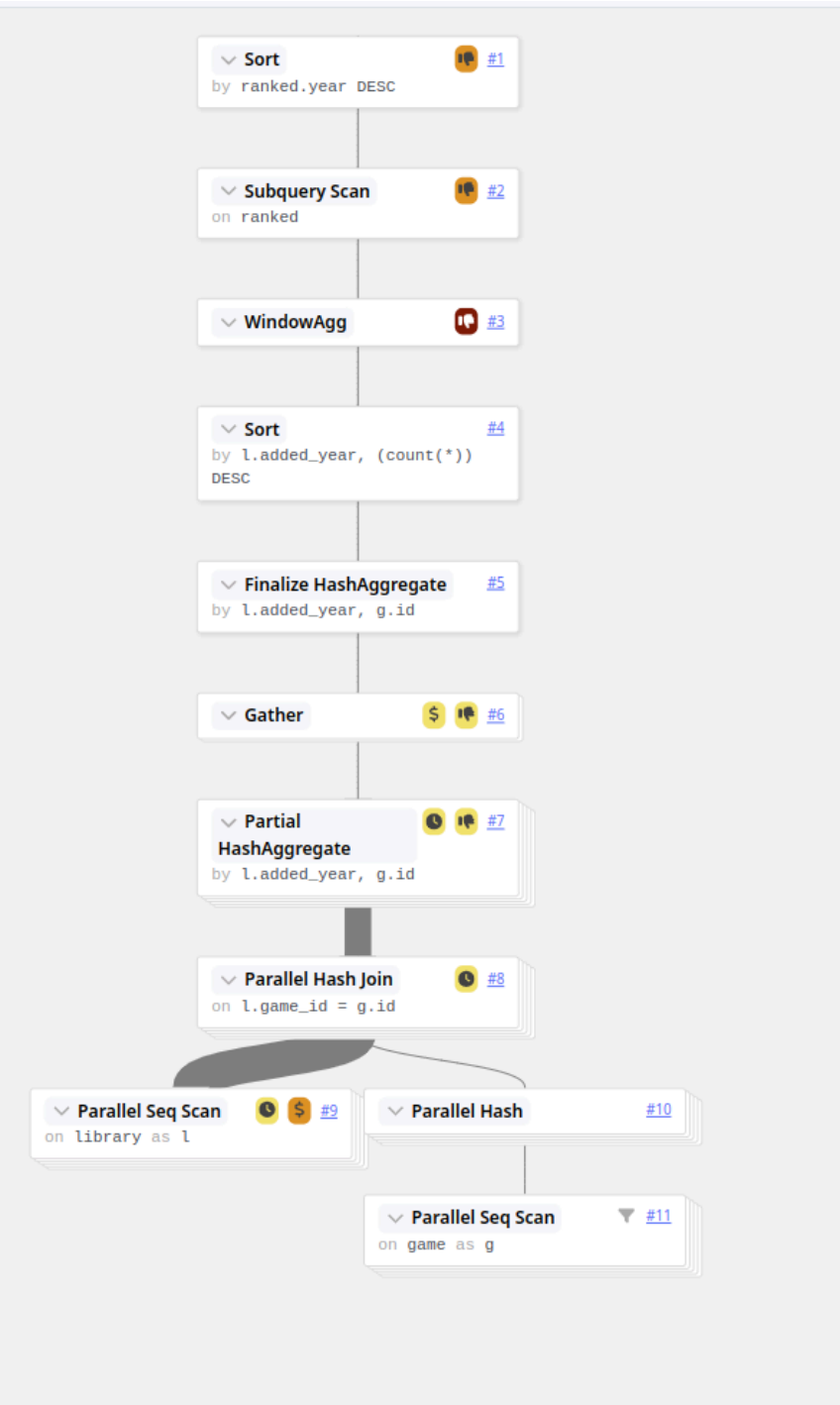


Figura 2: Plano de execução da *query* 1 após otimizações.

### 2.1.6. Resultados

Tabela 1: Resultados Finais e Análise Comparativa da Query 1

| Fase de Otimização | Tempo de Execução (s) | Melhoria Relativa | Melhoria Acumulada |
|---|---|---|---|
| Original | 71.156 | - | - |
| 1ª Fase | 18.332 | 74% | 74% |
| 2ª Fase | 17.442 | 5% | 75% |
| 3ª Fase | 12.734 | 27% | 82% |
| 4ª Fase | 10.181 | 20% | 86% |
| 5ª Fase | 7.776 | 24% | 89% |

## 2.2. Interrogação analítica 2

```sql
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;
```

### 2.2.1. Descrição da Query

Esta interrogação retorna os 15 jogos mais populares entre os amigos de um utilizador específico (*ID* 3331100), mas que este utilizador ainda não possui na sua biblioteca.

Assim, esta *query* tem como objetivo sugerir jogos para um utilizador, com base no que os seus amigos jogam. Primeiro, ela identifica todos os amigos deste utilizador consultando a tabela de amizades (*friendship*). Em seguida, verifica quais jogos estes amigos possuem nas suas bibliotecas (*library*). Depois, filtra estes jogos para garantir que o utilizador (alvo) ainda não os tenha. Por fim, agrupa os jogos encontrados, conta quantos amigos possuem cada um deles, ordena os resultados do mais popular para o menos popular e retorna os 15 primeiros. Assim, o utilizador recebe uma lista dos jogos mais populares entre os seus amigos, mas que ainda não fazem parte da sua biblioteca.

### 2.2.2. Análise do Plano de Execução

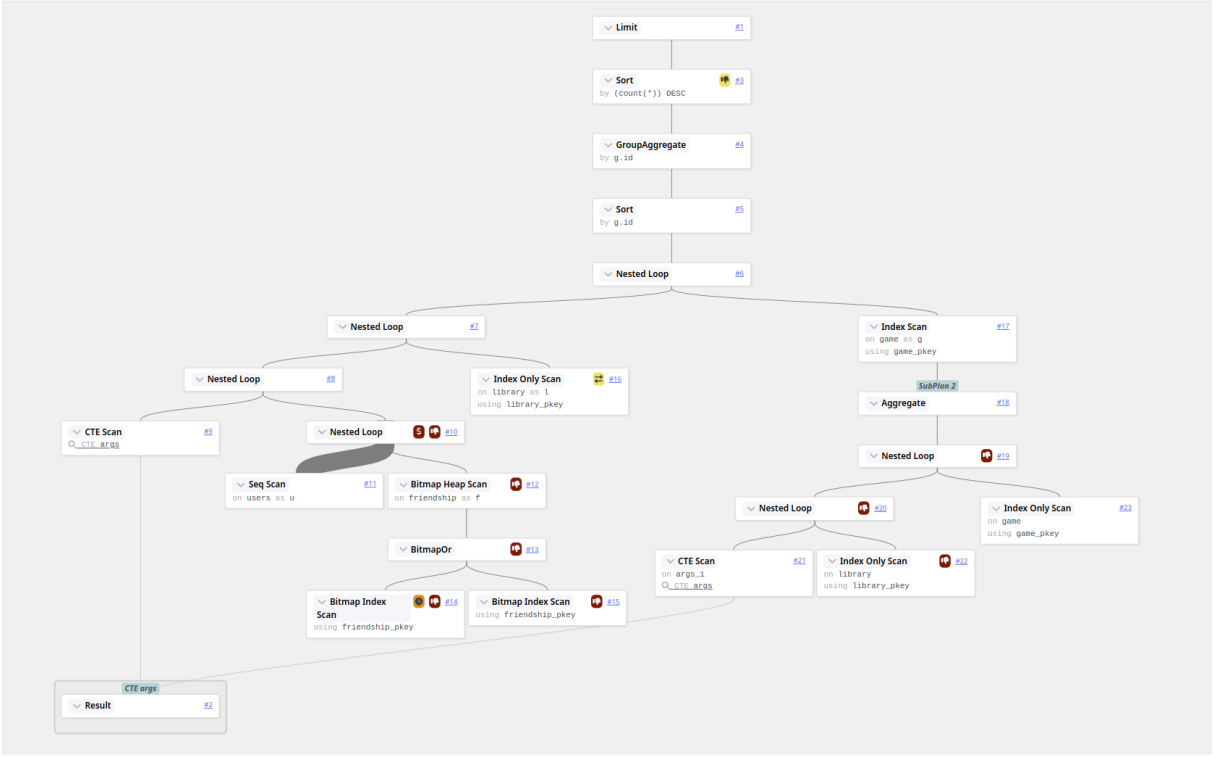


Figura 3: Plano de execução da *baseline* da *query* 2 sem otimizações

Ao analisar o plano de execução conseguimos notar uma demora significativa na parte do filtro das amizades, isto é do *Node* #10 ao #15. Notamos que o "OR" lógico nos `JOIN` da `CTE` (tanto `f.user_id_1 = u.id AND f.user_id_2 = args.id` como o inverso) não permite o uso eficiente de índices. Isto leva à construção de um *bitmap* combinado e a uma leitura pesada da *heap*, o que piora o desempenho.

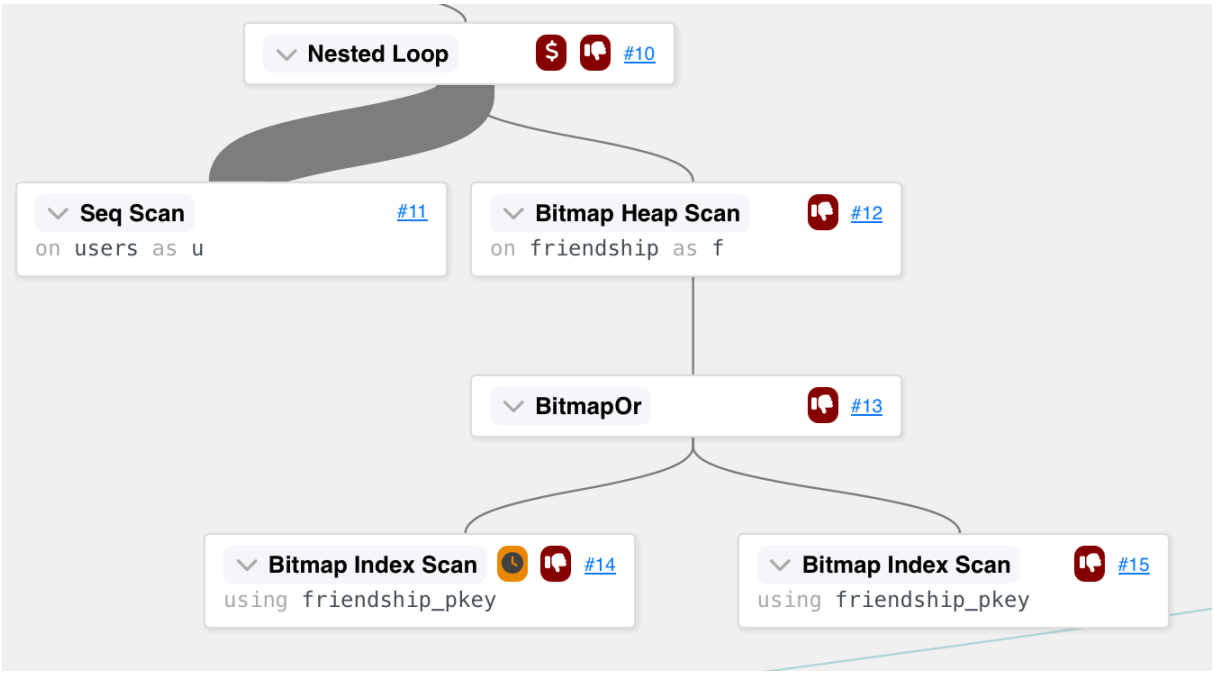


Figura 4: Plano de execução relativo ao *Nested Loop*.

### 2.2.3. Otimizações

#### 2.2.3.1. Primeira fase de otimização: Reestruturação da Consulta

```
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 args
    JOIN friendship f
        ON f.user_id_1 = args.id OR f.user_id_2 = args.id
),
friend_games AS (
    SELECT g.id, g.name
    FROM game g
    JOIN library l ON l.game_id = g.id
    JOIN friends f ON f.friend_id = l.user_id
    WHERE NOT EXISTS (
        SELECT 1
        FROM library l2
        WHERE l2.user_id = (SELECT id FROM args) AND l2.game_id = g.id
    )
)
SELECT g.id, g.name, COUNT(*) AS friend_count
FROM friend_games g
GROUP BY g.id, g.name
ORDER BY friend_count DESC
LIMIT 15;
```

#### 2.2.3.2. Alterações detalhadas

De forma a corrigir o `OR` em `JOIN` que força o *planner* a usar `BitmapOr` e `Heap Scans`, pois não pode usar um único índice eficientemente, decidimos melhorar o código *sql* removendo o `OR` lógico e substituindo-o por uma lógica determinística via `CASE`. Desta forma, o filtro `ON f.user_id_1 = args.id OR f.user_id_2 = args.id` ainda existe, mas é isolado no `CTE` *friends*, o que gera uma lista direta de *friend_id*, simplificando o plano de execução.

```
CASE
  WHEN f.user_id_1 = args.id THEN f.user_id_2
  ELSE f.user_id_1
END AS friend_id
```

Para além disso, notamos que a *subquery* com `count(*) = 0` é executada para cada linha do `game`, formando um gargalo. Assim, é executada uma vez para cada jogo de amigo, com `JOIN` internos e agregações, o que causa repetição de leituras nas tabelas `library` e game. Desta forma, substituímos a *subquery* pela que está abaixo. O `NOT EXISTS` pára na primeira ocorrência, evitando a agregação `count(*)`.

```
```

```
WHERE NOT EXISTS (
  SELECT 1
  FROM library l2
  WHERE l2.user_id = (SELECT id FROM args) AND l2.game_id = g.id
)
```

No início da *query* notamos um `JOIN users u ON true` desnecessário, pois fazia um *scan* completo na tabela `users` o `JOIN` é redundante com a cláusula `ON true`, o que impede otimizações. Desta forma, retiramos esta parte ficando sem `JOIN` com `users` e tendo agora o `friends` a entregar diretamente `friend_id`.

#### 2.2.3.3. Segunda fase de otimização: Criação de índices

De forma a melhorar ainda mais a `Bitmap Index Scan` (node #14) criamos o índice abaixo. Assim, enquanto antes o *planner* usava `BitmapOr` entre *user_id_1* e *user_id_2* agora, permitimos usar *scan* direto via índice para *user_id_1*, reduzindo o custo de leitura e gerindo um plano mais seletivo e mais rápido ao extrair amizades onde o utilizador é o *user_id_1*.

```
CREATE INDEX idx_friendship_user_ids ON friendship (user_id_1, user_id_2);
```

Para a outra metade do `BitmapOr` (node #15) criamos outro índice de forma a que o *planner* possa utilizar dois `Bitmap Index Scans` eficientes, um por cada direção da amizade. Assim, evitamos o custo de `BitmapOr` ao dividir a lógica de amizade.

```
CREATE INDEX idx_friendship_user_ids_reversed ON friendship (user_id_2,
user_id_1);
```

Para otimizar a seguinte parte da *query*:

```
JOIN library l ON l.game_id = g.id
JOIN friends f ON f.friend_id = l.user_id
```

Criamos outro índice de forma a permitir um *lookup* eficiente por *user_id* (amigos). Este índice permite um acesso por *user_id* (amigo) direto e eficiente.

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

Finalmente, de forma a facilitar a condição inversa `WHERE l2.game_id = g.id AND l2.user_id = ...` e melhorar a *subquery* `NOT EXISTS` que filtra jogos já possuídos, criamos um último índice. Com o mesmo, garantimos que as buscas que são realizadas primeiro por jogo e depois por *user_id*, também sejam eficientes.

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

#### 2.2.3.4. Análise da Evolução do Plano de Execução

A comparação entre o plano de execução original e o plano otimizado revela uma melhoria substancial na eficiência da consulta, tanto em termos de estrutura lógica como de custo computacional. A diferença estrutural entre os planos permite inferir uma melhoria expressiva na performance geral, resultado da reformulação da consulta *SQL* e da criação de índices direcionados.

O plano original apresenta uma complexidade significativa, com profundas cadeias de `Nested Loops`, múltiplos `Bitmap Heap Scans`, e avaliações repetidas de *subqueries* através de operações de agregação

`(count(\*) = 0)`. Esta abordagem implicava um grande volume de leituras não indexadas, especialmente sobre as tabelas *friendship* e *library*, obrigando a recorrer a estratégias de *scanning* de dados menos eficientes, como `Seq Scan` e `Bitmap Heap Scan`. Além disso, a filtragem de amigos do utilizador era feita *inline*, com uma condição `OR` complexa, dificultando a utilização de índices compostos e aumentando o custo da execução.

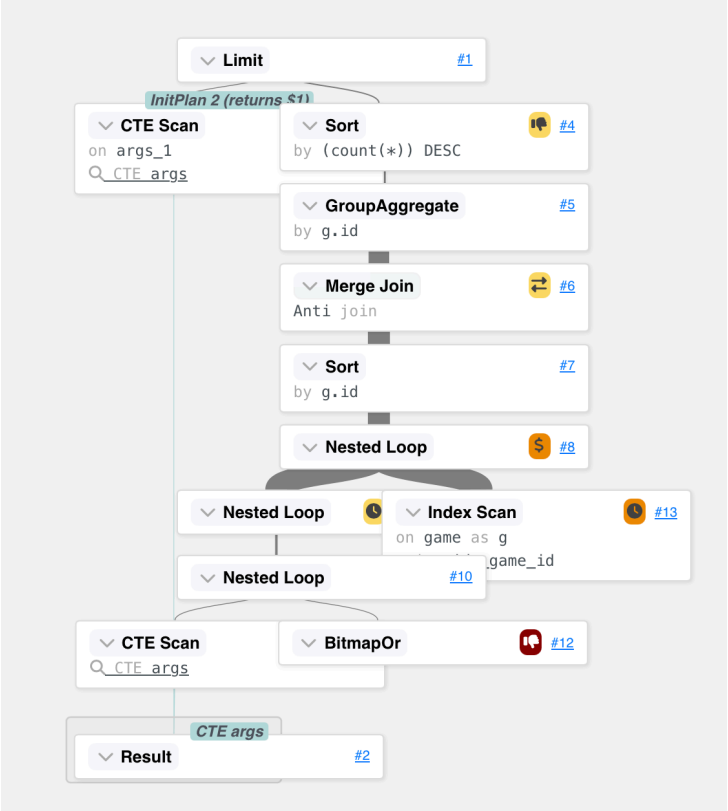


Figura 5: Plano de execução da *query 2* após otimizações.

Em contraste, o plano otimizado adota uma estrutura significativamente mais racional e modular, com a utilização de *Common Table Expressions (CTEs)* como *friends* e *friend_games*, que contribuem para a clareza lógica e permitem ao otimizador *SQL* aplicar estratégias mais eficientes. A reformulação da *subquery* de exclusão (substituindo `count(\*) = 0` por `NOT EXISTS`) representa uma melhoria crítica, eliminando a necessidade de agregações dispendiosas. Esta alteração permitiu a introdução de um `Merge Join`, altamente performático, em substituição de vários `Nested Loops`.

A melhoria mais significativa é a utilização eficaz dos índices criados (`idx_friendship_user_ids`, `idx_library_user_game`, entre outros). O plano mostra agora o uso recorrente de `Index Scans` e `Index Only Scans`, reduzindo drasticamente a quantidade de dados lidos em disco. Estas alterações otimizam tanto a leitura de amizades como a verificação de jogos já possuídos pelo utilizador.

Apesar de ainda se verificarem algumas ocorrências de `Sort` e operações intermediárias de ordenação, estas são muito mais localizadas e ocorrem em conjuntos de dados substancialmente reduzidos graças à filtragem inicial mais eficaz. A quantidade de nós no plano também foi significativamente reduzida, tornando o plano mais previsível.

Em termos funcionais, o número de linhas processadas mantém-se consistente, o que indica que as alterações estruturais não afetaram a semântica da consulta nem a qualidade dos resultados, mas apenas a forma como são obtidos.

#### 2.2.4. Resultados

Tabela 2: Resultados Finais e Análise Comparativa da *query* 2

| Fase de Otimização | Tempo de Execução (s) | Melhoria Relativa | Melhoria Acumulada |
|---|---|---|---|
| Original | 209.27 | - | - |
| 1ª Fase | 169.28 | 19.12% | 19.12% |
| 2ª Fase | 45.16 | 73.33% | 78.41% |
| 3ª Fase | 0.04 | 99.91% | 99.98% |

### 2.3. Interrogação analítica 3

```sql
SELECT count(*) AS total, count(DISTINCT id) AS unique
FROM (
    (
        SELECT p.name, u.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
        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';
```

#### 2.3.1. Descrição da Query

Esta interrogação tem como objetivo contar o número total de registos e o número de utilizadores distintos que possuem na sua biblioteca jogos associados à *Ubisoft*, seja enquanto publicadora ou enquanto desenvolvedora. O critério de seleção é realizado com base nos nomes das entidades (*publisher* ou *developer*) que sejam iguais a *Ubisoft*.

A estrutura da *query* baseia-se na união (`UNION ALL`) de dois conjuntos de dados:

O primeiro conjunto corresponde aos jogos associados à `Ubisoft` como publicadora, fazendo a junção entre as tabelas `publisher`, `games_publishers`, `game`, `library` e `users`, obtendo o nome da publicadora e o identificador do utilizador que possui o jogo.

O segundo conjunto segue uma lógica idêntica, mas refere-se aos jogos em que a *Ubisoft* atua como desenvolvedora, utilizando as tabelas *developer* e *games_developers*.

Após esta união, é aplicada uma filtragem sobre o campo `name`, selecionando apenas os registos em que o nome da publicadora ou desenvolvedora é igual a *Ubisoft*. A *query* retorna duas métricas principais, o *total*, que corresponde ao número total de registos encontrados (incluindo possíveis repetições de utilizadores), e o *unique* que corresponde ao número de utilizadores distintos (`count(DISTINCT id)`) que possuem pelo menos um desses jogos na biblioteca.

### 2.3.2. Análise do Plano de Execução

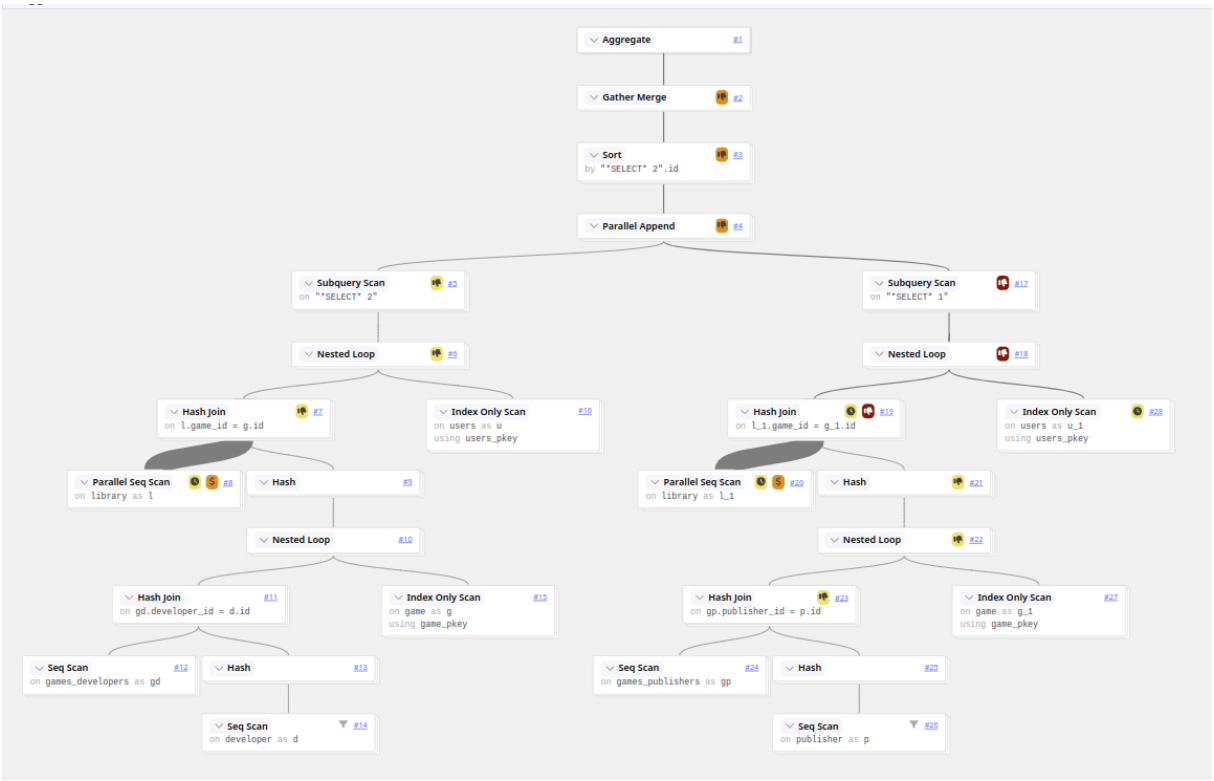


Figura 6: Plano de execução da *query 3* antes das otimizações

A consulta apresentada evidencia um tempo de execução elevado, cerca de 29 segundos, o que reflete diversas limitações de desempenho relacionados principalmente ao volume dos dados e à estrutura da *query*. A operação realiza duas *subconsultas* paralelas com junções entre várias tabelas — como `library`, `game`, `developer`, `publisher` e `users` — e posteriormente realiza uma agregação sobre o conjunto resultante de quase 3,8 milhões de linhas. Mesmo com o uso de paralelização, que envolveu dois *workers*, o tempo total de execução permanece alto, o que indica que a estratégia de paralelismo não está a ser suficiente para mitigar o custo computacional da consulta.

Um dos principais pontos críticos identificados é a ausência de índices adequados nas colunas utilizadas com filtros `LIKE`. Tanto a tabela `developer` quanto a tabela `publisher` são acedidas através de `Sequential Scans`, apesar de conterem mais de 50 mil e 42 mil registros, respetivamente. A filtragem pelo nome *Ubisoft* acaba por forçar uma leitura completa destas tabelas, o que aumenta significativamente o tempo de leitura.

Além disso, a ordenação dos resultados para o `Gather Merge` foi realizada através de `external merge sort`, o que indica que o sistema não conseguiu realizar a operação inteiramente em memória. A necessidade de utilizar mais de 20 MB de espaço em disco por `worker` para ordenar os dados reflete uma limitação da configuração de `work_mem`, e evidencia a ineficiência deste processo.

Outro aspecto relevante é o uso intensivo de `Nested Loops` e `Hash Joins`, em especial nas operações envolvendo a tabela `library`, que chega a ser percorrida mais de 60 milhões de vezes em cada uma das *subconsultas* paralelas. Isso resulta num número extremamente alto de blocos acedidos (mais de

13 milhões de blocos em cache e 2,4 milhões lidos do disco). A utilização de estratégias como particionamento da tabela `library`, criação de índices compostos mais seletivos, ou mesmo reformulação da lógica da *query* para reduzir o número de registos intermediários, pode gerar ganhos substanciais de desempenho.

Apesar da utilização de `Index Only Scans` nas tabelas `game` e `users`, o volume de acessos aos índices ainda é elevado — no caso de `users`, foram mais de 3,5 milhões de leituras indexadas, o que, mesmo sem a necessidade de acessar a `heap`, representa um custo elevado em termos de leitura de blocos.

Por fim, é importante mencionar o uso de *JIT compilation (Just-In-Time)*, que foi ativado para esta consulta e envolveu tempo de geração de código e otimização superior a 1 segundo. Embora o *JIT* possa trazer benefícios em consultas repetidas ou extremamente complexas, o seu custo de ativação pode não compensar em casos isolados como este, principalmente se a consulta não for recorrente.

#### 2.3.3. 1ª Fase de Otimização

Na primeira fase do processo de optimização foi feita uma reescrita da query.

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

A reescrita da consulta introduz uma abordagem substancialmente mais eficiente em relação à versão original. Esta reformulação melhora o desempenho por várias razões fundamentais que se refletem diretamente na execução do plano e na utilização de recursos do sistema.

Em primeiro lugar, a nova versão estrutura a lógica em blocos utilizando `Common Table Expressions (CTEs)`, o que melhora a modularidade e facilita a compreensão do plano de execução. Cada `CTE` encapsula um conjunto de dados delimitado — jogos publicados ou desenvolvidos pela *Ubisoft* — e aplica os filtros de forma antecipada evitando o custo de transportar colunas desnecessárias (como `name`) até a etapa final de agregação.

Além disso, a utilização explícita das relações de junção com filtros aplicados o mais cedo possível (nomeadamente `LIKE 'Ubisoft'`) reduz significativamente o volume de dados intermediários. Na versão anterior, a cláusula `WHERE` era aplicada somente após a junção completa e a união dos conjuntos,

o que exigia a leitura e materialização de milhões de linhas antes de qualquer filtragem ser efetuada. Na versão atual, os filtros são aplicados no momento das junções com `publisher` e `developer`, evitando leituras desnecessárias e agilizando a eliminação de registos irrelevantes.

Outro ponto crucial de otimização diz respeito ao uso de `UNION ALL` apenas após as seleções filtradas. Embora o `UNION ALL` continue presente, opera agora sobre subconjuntos já reduzidos de dados, o que minimiza o seu impacto na memória e no custo de *CPU*. Esta abordagem evita a criação de um conjunto monolítico e repetitivo contendo múltiplas ocorrências de jogos por utilizador antes da agregação final.

Adicionalmente, ao focar apenas nas colunas necessárias (`user_id`, `game_id`) e evitar o transporte de dados não utilizados, a consulta permite ao *PostgreSQL* elaborar planos de execução mais eficientes.

Por fim, a separação clara entre a lógica dos `publishers` e `developers`, seguida de uma unificação posterior, permite um controlo mais granular sobre a execução de cada parte.

O plano de execução após estas otimizações mostrou uma redução significativa no tempo de execução para 12.944 segundos, representando uma melhoria de aproximadamente 56% em relação ao valor inicial.

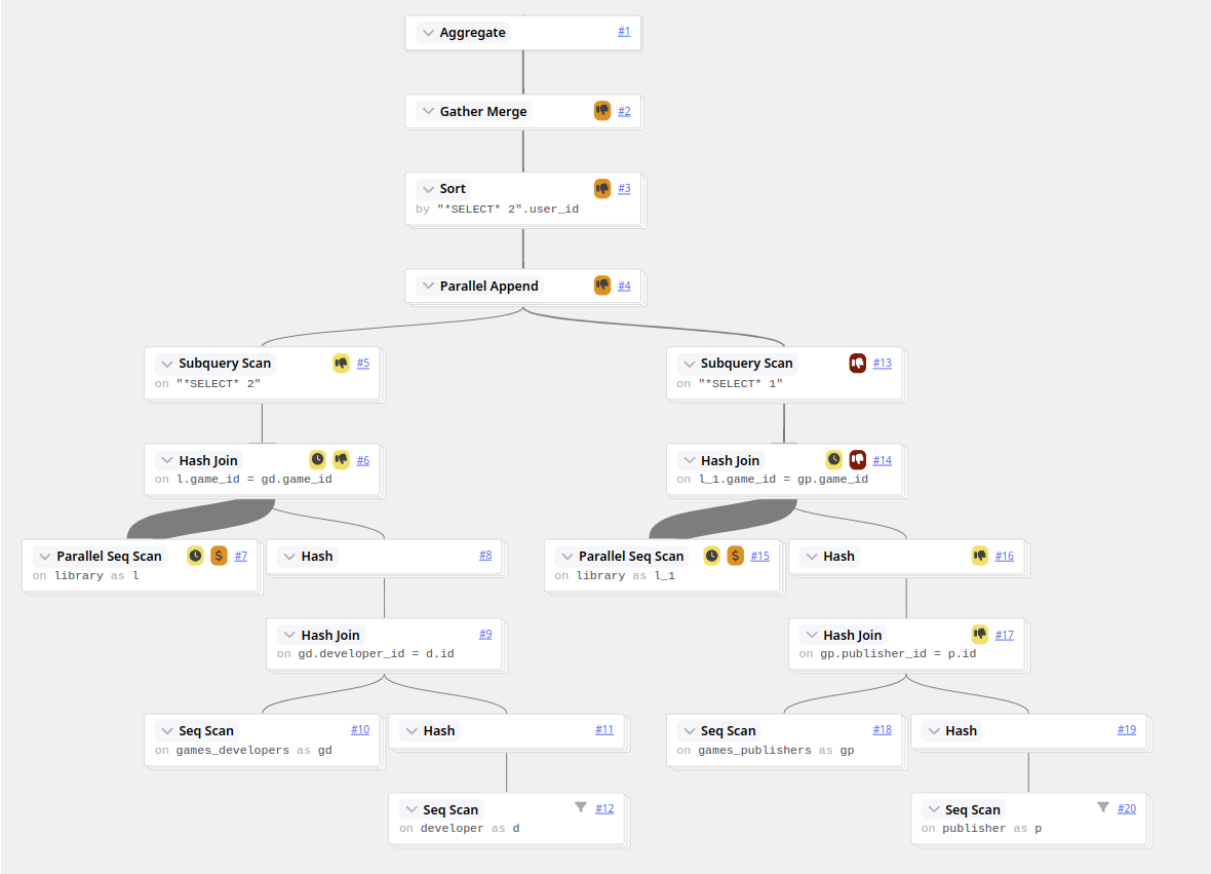


Figura 7: Plano de execução da *query 3* após 1ª fase de otimizações

### 2.3.4. 2ª Fase de Otimização

Com o objetivo de otimizar a performance da consulta, nesta 2ª fase de optimização foram criados índices direcionados aos principais pontos de acesso e filtragem das tabelas envolvidas. Cada índice foi concebido de forma a acelerar as operações mais frequentes no plano de execução, nomeadamente os `JOIN`s, os filtros por prefixo de nome e o acesso eficiente à tabela `library`, que contém o maior volume de dados.

```
CREATE INDEX idx_publisher_name ON publisher(name);
```

Este índice foi criado para acelerar o filtro textual aplicado sobre o nome das editoras (`publisher.name LIKE 'Ubisoft'`).

```
CREATE INDEX idx_developer_name ON developer(name);
```

Semelhante ao anterior, este índice acelera o filtro `developer.name LIKE 'Ubisoft'`.

```
CREATE INDEX idx_games_publishers_pub_game ON games_publishers(publisher_id,
game_id);
```

Este índice composto melhora a performance dos `JOIN` entre `games_publishers` e `publisher`, bem como entre `games_publishers` e `library`. A ordenação dos campos (`publisher_id`, `game_id`) reflete a sequência lógica dos `JOIN`, permitindo ao otimizador resolver as junções de forma eficiente com base em chaves estrangeiras.

```
CREATE INDEX idx_games_developers_dev_game ON games_developers(developer_id,
game_id);
```

Da mesma forma que o índice anterior, este índice otimiza os `JOINs` entre `games_developers`, `developer` e `library`. É especialmente útil quando existem muitos jogos com múltiplos `developers`, reduzindo significativamente o custo de acesso a estas relações.

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

Este índice foi criado para otimizar o acesso à tabela `library`, a qual representa os jogos pertencentes aos utilizadores. Como os `JOIN` com `games_publishers` e `games_developers` se baseiam em `game_id`, este índice permite resolver rapidamente essas associações. A inclusão de `user_id` na sequência também melhora a performance da etapa de agregação final.

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

Este índice adicional serve como complemento ao anterior, otimizando operações que agrupam ou filtram por `user_id`. É especialmente útil na contagem de utilizadores únicos (`COUNT(DISTINCT user_id)`), pois permite a leitura direta de todos os jogos pertencentes a um utilizador, sem necessidade de reordenação.

O tempo de execução reduziu para 2.992 segundos, uma melhoria de aproximadamente 76% em relação à fase anterior e de 89% em relação ao valor inicial, ou seja, a execução da consulta tornou-se quase 10 vezes mais rápida ao longo do processo de otimização.

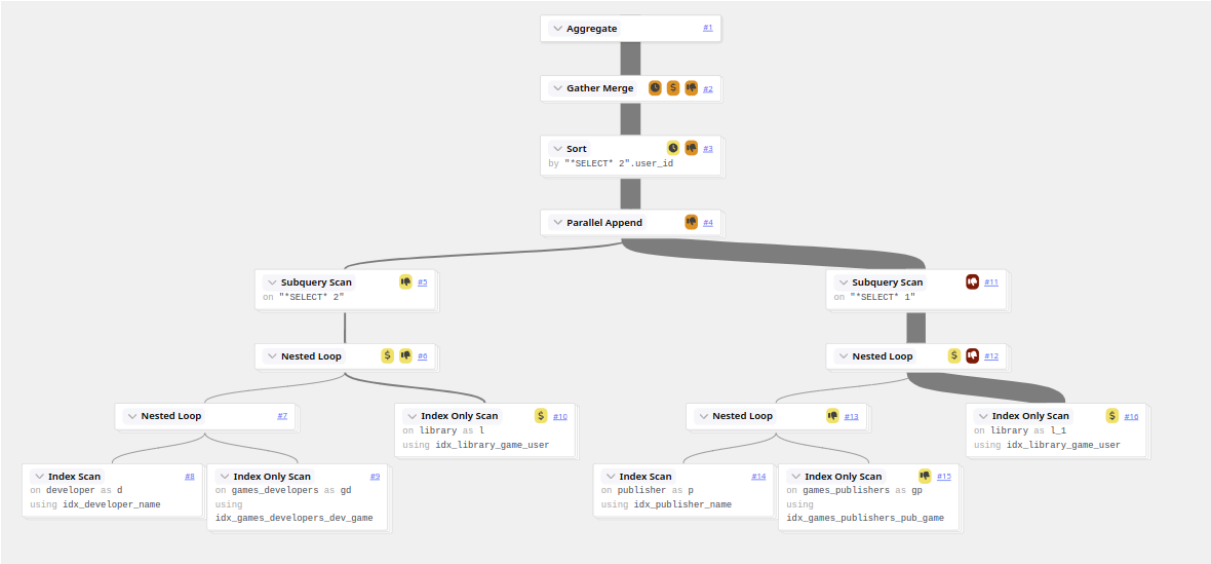


Figura 8: Plano de execução da *query 3* após 2ª fase de otimizações

### 2.3.5. 3ª Fase de Otimização

Nesta 3ª fase iremos testar uma abordagem diferente das duas anteriores. A ideia será manter uma tabela separada com estatísticas agregadas por empresa (`publisher` ou `developer`), de forma a evitar recalcular os valores totais e distintos de utilizadores em cada consulta. O processo será dividido em três passos principais:

- Criação da Tabela de Estatísticas;
- Povoamento Inicial da Tabela;
- Atualização Automática com *Trigger*

#### 2.3.5.1. Criação da Tabela de Estatísticas

Vamos criar a tabela `company_user_game_stats` que irá armazenar, por cada empresa, o número total de jogos nas bibliotecas dos utilizadores (total) e o número de utilizadores distintos que possuem pelo menos um jogo dessa empresa (`unique_users`).

```
CREATE TABLE company_user_game_stats (
    company_name TEXT PRIMARY KEY,
    total INT,
    unique_users INT,
    last_updated TIMESTAMP DEFAULT NOW()
);
```

#### 2.3.5.2. Povoamento Inicial da Tabela

A tabela será inicialmente populada com base nos dados existentes nas tabelas `library`, `games_publishers`, `games_developers`, `publisher` e `developer`. Serão agregadas as estatísticas por empresa (tanto como `publishercomo developer`), somando os valores de forma acumulada, agrupando por nome da empresa.

```
-- Atualiza estatísticas para todas as empresas (publishers e developers)
WITH publisher_user_games AS (
    SELECT p.name AS company_name, l.user_id, l.game_id
    FROM games_publishers gp
    JOIN publisher p ON p.id = gp.publisher_id
```

```
    JOIN library l ON l.game_id = gp.game_id
),
developer_user_games AS (
    SELECT d.name AS company_name, l.user_id, l.game_id
    FROM games_developers gd
    JOIN developer d ON d.id = gd.developer_id
    JOIN library l ON l.game_id = gd.game_id
),
combined_user_games AS (
    SELECT * FROM publisher_user_games
    UNION ALL
    SELECT * FROM developer_user_games
),
aggregated_stats AS (
    SELECT
        company_name,
        COUNT(*) AS total,
        COUNT(DISTINCT user_id) AS unique_users
    FROM combined_user_games
    GROUP BY company_name
)
INSERT INTO company_user_game_stats (company_name, total, unique_users,
last_updated)
SELECT
    company_name, total, unique_users, NOW()
FROM aggregated_stats
ON CONFLICT (company_name) DO UPDATE
SET total = EXCLUDED.total,
    unique_users = EXCLUDED.unique_users,
    last_updated = NOW();
```

#### 2.3.5.3. Atualização Automática com *Trigger*

Para manter os dados sincronizados com novas inserções na `library`.

```
CREATE TRIGGER trg_update_stats_after_insert
AFTER INSERT ON library
FOR EACH ROW
EXECUTE FUNCTION update_company_game_stats_on_insert();
```

Sempre que um novo jogo for adicionado à biblioteca de um utilizador, a função `update_company_game_stats_on_insert` será executada automaticamente e irá:

- Identificar as empresas associadas ao jogo.
- Incrementa o valor total para o `publisher` e `developer`.
- Garante que o número de utilizadores distintos não é incrementado se o jogador já tiver na sua `library` outro jogo do mesmo *developer*/*publisher*.

Este método visa melhorar a performance em consultas frequentes, movendo o custo do cálculo estatístico para o momento da inserção de dados, o que é particularmente útil quando o volume de leituras supera o de escritas.

```

```

```
CREATE OR REPLACE FUNCTION update_company_game_stats_on_insert()
RETURNS TRIGGER AS $$
BEGIN
    -- Publishers
    INSERT INTO company_user_game_stats (company_name, total, unique_users,
last_updated)
    SELECT p.name, 1, 1, NOW()
    FROM games_publishers gp
    JOIN publisher p ON p.id = gp.publisher_id
    WHERE gp.game_id = NEW.game_id
    ON CONFLICT (company_name) DO UPDATE
    SET total = company_user_game_stats.total + 1,
        unique_users = company_user_game_stats.unique_users +
            CASE
                WHEN EXISTS (
                    SELECT 1 FROM library l
                    JOIN games_publishers gp2 ON l.game_id = gp2.game_id
                    JOIN publisher p2 ON p2.id = gp2.publisher_id
                    WHERE l.user_id = NEW.user_id
                      AND gp2.game_id != NEW.game_id
                      AND p2.name = company_user_game_stats.company_name
                ) THEN 0
                ELSE 1
            END,
        last_updated = NOW();

    -- Developers
    INSERT INTO company_user_game_stats (company_name, total, unique_users,
last_updated)
    SELECT d.name, 1, 1, NOW()
    FROM games_developers gd
    JOIN developer d ON d.id = gd.developer_id
    WHERE gd.game_id = NEW.game_id
    ON CONFLICT (company_name) DO UPDATE
    SET total = company_user_game_stats.total + 1,
        unique_users = company_user_game_stats.unique_users +
            CASE
                WHEN EXISTS (
                    SELECT 1 FROM library l
                    JOIN games_developers gd2 ON l.game_id = gd2.game_id
                    JOIN developer d2 ON d2.id = gd2.developer_id
                    WHERE l.user_id = NEW.user_id
                      AND gd2.game_id != NEW.game_id
                      AND d2.name = company_user_game_stats.company_name
                ) THEN 0
                ELSE 1
            END,
        last_updated = NOW();

    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Com isto o tempo de execução passa a ser de cerca de 0.08s (basicamente instantâneo). Em contrapartida as inserções serão um pouco mais lentas, demorando cerca de 3ms a mais do que as inserções sem o *trigger*.

### 2.3.6. Resultados

Tabela 3: Resultados Finais e Análise Comparativa da *query* 3

| Fase de Otimização | Tempo de Execução (s) | Melhoria Relativa | Melhoria Acumulada |
|---|---|---|---|
| Original | 29.603 | - | - |
| 1ª Fase | 12.944 | 56% | 56% |
| 2ª Fase | 2.992 | 77% | 89% |
| 3ª Fase | 0.08 | 97% | 99% |

## 2.4. Interrogação analítica 4

```sql
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;
```

### 2.4.1. Descrição da Query

Esta *query* tem como objetivo analisar a atividade de aquisição de jogos na plataforma ao longo do tempo, dividindo os eventos em intervalos regulares de 12 horas (*bins* temporais). Para isso, é utilizada a função `generate_series` para criar uma sequência de intervalos horários e a função `date_bin` para alinhar as datas dos registos com os intervalos definidos. A cláusula `JOIN` associa cada registo da tabela `library` ao respetivo intervalo, e a contagem é realizada com base nos grupos definidos por cada valor de `bins.bin`.

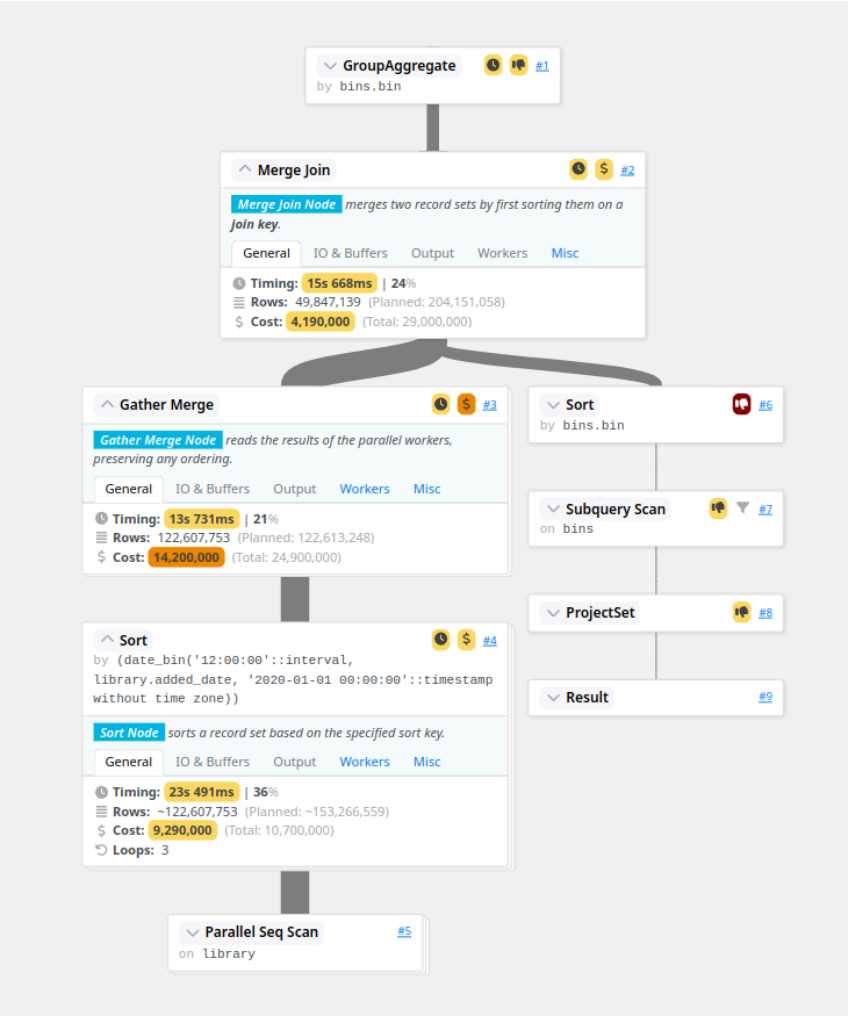


Figura 9: Plano de execução da *baseline* da *query* 4 antes das otimizações.

O plano de execução da *query*, obtido com `EXPLAIN ANALYZE`, revelou importantes características de desempenho:

- **Tempo total de execução**: aproximadamente **65 segundos** (65163.981 ms), o que indica uma operação intensiva, especialmente considerando que apenas se está a contar registos por intervalo.
- O nó principal da execução foi um `GroupAggregate`, que recebeu dados de um `Merge Join` entre a tabela `library` (após aplicação de `date_bin`) e a sequência gerada por `generate_series`.
- A operação mais pesada foi o `Parallel Seq Scan` na tabela `library`, indicando que todo o conteúdo da tabela foi lido sequencialmente por múltiplos *workers*, o que justifica o volume de dados processado (mais de **122 milhões de linhas**).
- A ordenação necessária para o `Merge Join` foi realizada com `external merge sort`, usando mais de **1.3 GB de espaço em disco por worker**, o que sugere que os dados não cabiam na memória (`work_mem`) disponível.
- A subquery de `bins` gerou cerca de 15.815 entradas, mas apenas 3.905 foram mantidas após o `Filter`, implicando que muitos intervalos gerados foram descartados tardiamente, já durante a execução da *query*.

### 2.4.2. 1ª Fase de Otimização

A *query* original aplicava a função `date_bin('12 hours', added_date, '2020-01-01')` diretamente sobre milhões de linhas da tabela `library`. Esta abordagem resultava num **custo computacional elevado** devido a:

- Leitura sequencial completa da tabela (*Sequential Scan*);
- Ordenação externa em disco para permitir o `Merge Join`;
- Impossibilidade de utilizar índices sobre `added_date`, uma vez que a função `date_bin()` é aplicada diretamente à coluna na cláusula `JOIN`.

Estas operações tornavam a execução da *query* lenta e ineficiente, especialmente para volumes de dados elevados.

Desta forma foi adotada uma abordagem de materialização da função `date_bin()` numa nova coluna, de forma a evitar o seu cálculo em tempo de execução.

Para tal foram utilizadas as seguintes diretivas:

```
ALTER TABLE library ADD COLUMN binned_added_date timestamp;
UPDATE library SET binned_added_date = date_bin('12 hours', added_date,
'2020-01-01');
CREATE INDEX idx_library_binned_added_date ON library(binned_added_date);
```

Com estas alteração foi possível realizar a escrita da *query* para utilizar a nova coluna:

```
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 library.binned_added_date = bin
WHERE bin >= '2020-01-01'
GROUP BY bin
ORDER BY bin;
```

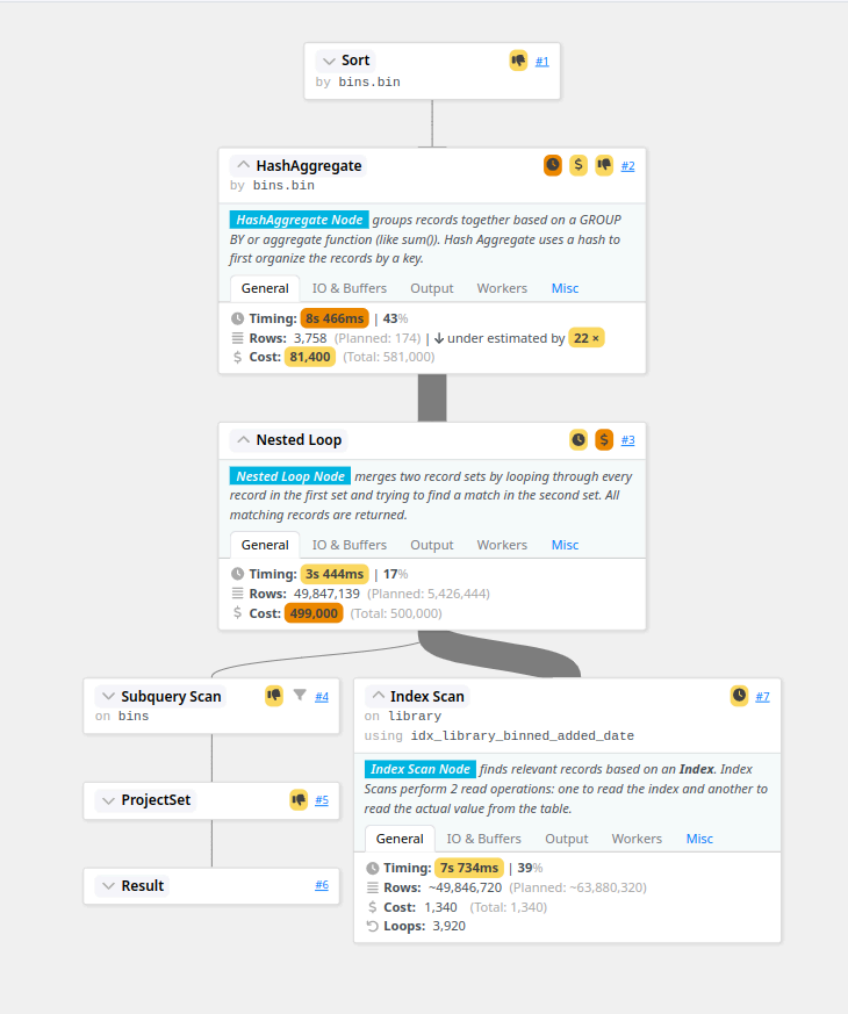


Figura 10: Plano de execução da *baseline* da *query* 4 após a 1ª fase de otimização

Após a otimização, o plano de execução da *query* passou a beneficiar de **acesso indexado** (`Index Scan`), evitando a leitura sequencial da tabela e a aplicação da função `date_bin()` (evitando o *full sort* e *scan* desnecessários). O tempo de execução da *query* reduziu para cerca de 19 segundos (19.73044 segundos), tendo reduzido significativamente, proporcionando ganhos expressivos em eficiência e escalabilidade.

#### 2.4.2.1. Considerações adicionais

Para garantir que os novos registos fiquem sincronizados com esta otimização, foi também implementado um **trigger** que atualiza automaticamente a coluna `binned_added_date` sempre que há inserções ou atualizações:

```
CREATE FUNCTION update_binned_added_date() RETURNS trigger AS $$
BEGIN
  NEW.binned_added_date := date_bin('12 hours', NEW.added_date, '2020-01-01');
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_update_binned_added_date
BEFORE INSERT OR UPDATE ON library
FOR EACH ROW
EXECUTE FUNCTION update_binned_added_date();
```

### 2.4.3. 2ª Fase de Otimização

Apesar da introdução da colina `binned_added_date` e do seu respetivo índice ter reduzido significativamente o custo computacional, é possível realizar melhorias adicionais através da **redução do volume de dados processados** e da **eliminação de operações desnecessárias na geração de *bins***.

A `CTE bins` gera todos os intervalos de 12h desde *2003-09-12* até ao `now()`, dos quais apenas uma fração é efetivamente utilizada. Essa abordagem:

- Cria milhares de `bins` irrelevantes, como mostrado pelo `Rows Removed by Filter: 11910`;
- Ocasiona **sobrecarga de memória e tempo de ordenação** desnecessários;
- Agrava o custo do `Merge Join`, pois todos esses `bins` são considerados antes da filtragem.

Desta forma, é possível reduzir o intervalo gerado pela `CTE (bins)` para começar diretamente de `2020-01-01`, eliminando a necessidade de aplicar `Where bin >= '2020-01-01'` posteriormente.

```
WITH bins AS (
    SELECT generate_series('2020-01-01', 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 library.binned_added_date = bin
GROUP BY bin
ORDER BY bin;
```

Com esta alteração, a geração de `bins` tornou-se **mais eficiente**, reduzindo o volume total de linhas processadas e acelerando a fase de **join** e **grouping**.

### 2.4.4. 3ª Fase de Otmização

Após as otimizações anteriores, ainda se observava um custo elevado no plano de execução, principalmente causado por:

- O `Nested Loop` gerado ao combinar `generate_series` com a tabela `library`;
- O `Index Scan` executado repetidamente para cada `bin`;
- Um `HashAggregate` custoso para consolidar os resultados;

Para superar estas limitações, foi aplicada uma otimização **estrutural: eliminar completamente o uso da função `generate_series()`** e utilizar uma agregação direta com `GROUP BY` sobre a coluna `binned_added_date`, previamente materializada.

Desta forma foi alterada a estrutura da *query*:

```
SELECT binned_added_date 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 binned_added_date >= '2020-01-01'
GROUP BY binned_added_date
ORDER BY binned_added_date;
```

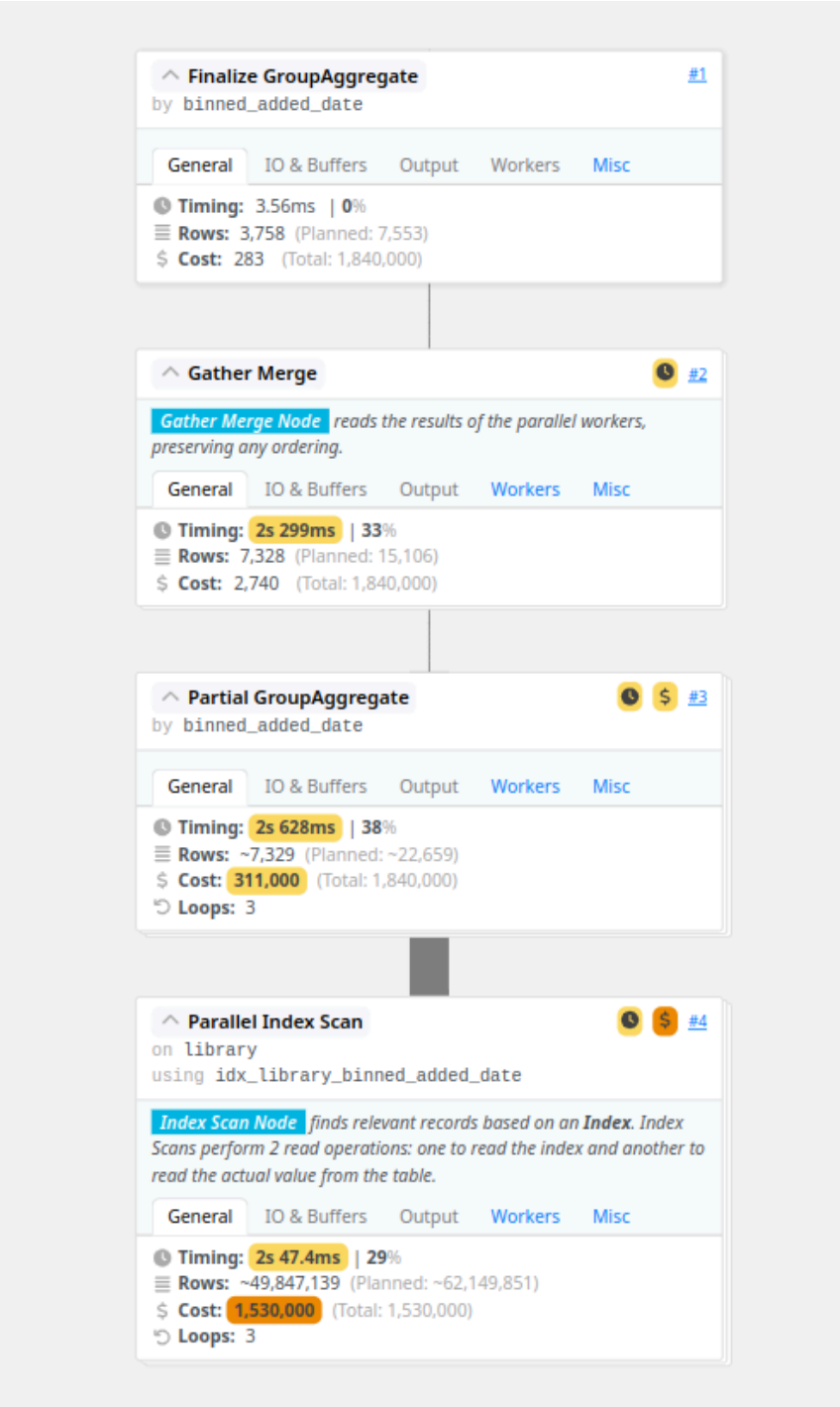


Figura 11: Plano de execução da *baseline* da *query* 4 após a 3ª fase de otimização

Esta nova *query* assume que **bins vazios (sem dados)** não precisam ser incluídos. Isso permite uma abordagem muito mais eficiente, com agregação direta. Podemos verificar que o tempo de execução da *query* foi reduzido para cerca de 6 segundos (6978.824 ms), além disso foi possível reduzir o custo de execução evitando o uso do *Nested Loop*, *Hash Aggegate* e paralelizar o *Index Scan.*

#### 2.4.5. 4ª fase de otimização

Por fim, de forma a remover o acesso desnecessário à *heap*, durante o *Index Scan* com o objetivo de tornar a consulta mais eficiente em termos de I/O foi necessário reescrever o `Index` criado na 1ª fase de otimização.

Mesmo com um índice existente sobre a coluna `binned_added_date`, o *PostgreSQL* ainda precisa acessar a tabela original para recuperar outras colunas não presentes no índice (*heap fetch*). Ao criarmos um *covering index*, adicionamos explicitamente as colunas necessárias dentro do *Index*, isto permite executar a *query* apenas com os dados do *Index* sem a necessidade de aceder à tabela.

```
DROP INDEX IF EXISTS idx_library_binned_added_date;
CREATE INDEX idx_library_binned_added_date_covering
  ON library (binned_added_date)
  INCLUDE (buy_price);
```

Esta melhoria tira proveito da funcionalidade `INCLUDE` do *PostgreSQL* para criar um índice coberto (*covering index*), ou seja, um índice que contém todas as colunas necessárias para satisfazer a *query*, mesmo que algumas não façam parte da ordenação principal do índice.

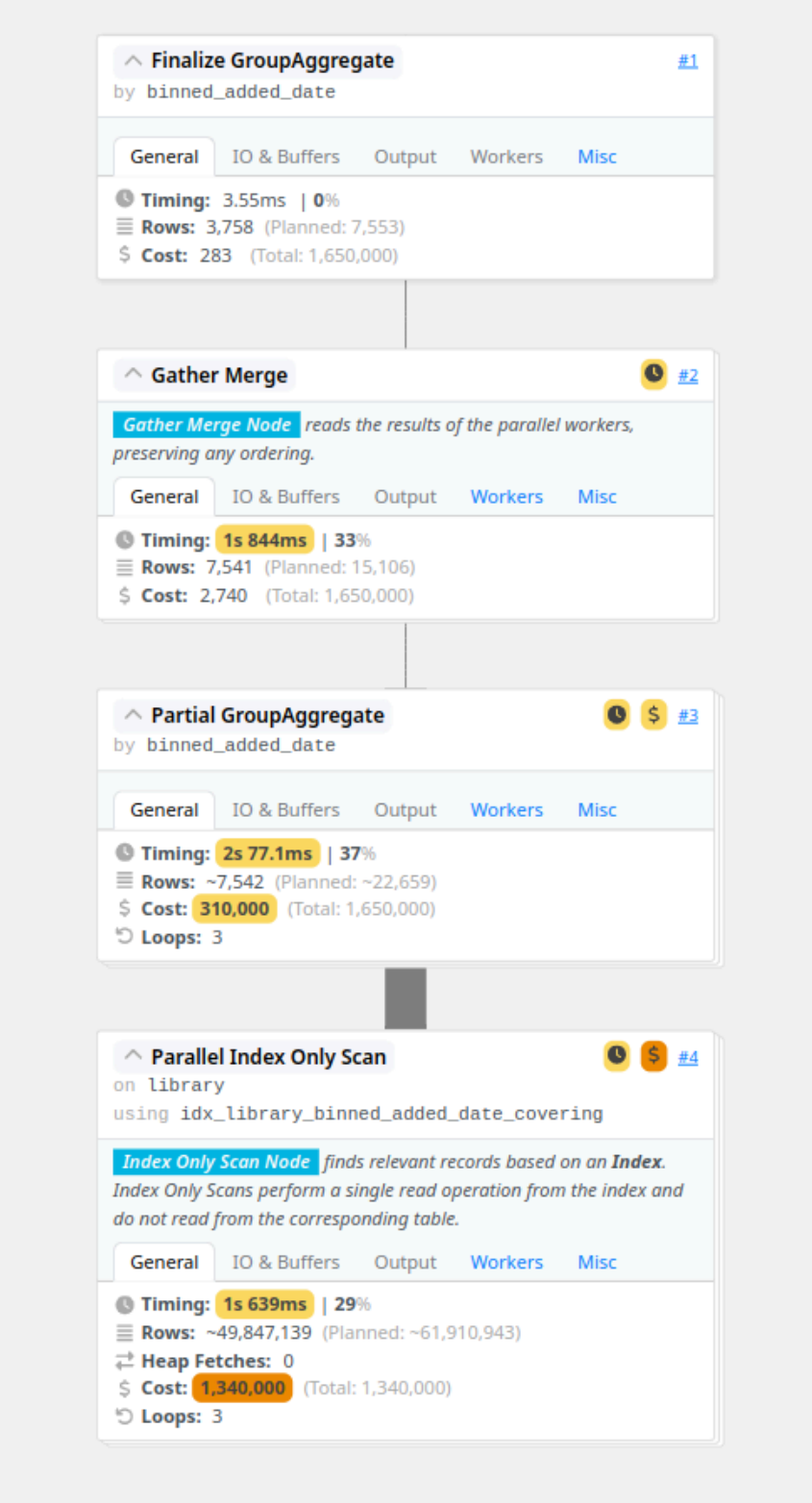


Figura 12: Plano de execução da *baseline* da *query* 4 após a 4ª fase de otimização

O resultado direto desta alteração é a substituição do *Index Scan* por um *Index Only Scan*, conforme confirmado com `EXPLAIN ANALYZE`. Com isso, eliminam-se os *heap fetches* (acessos adicionais à tabela base), o que resulta em:

- Menor carga de leitura em disco;
- Menor utilização de CPU para reconstrução de tuplos;

- Melhoria significativa de desempenho, especialmente em cenários com grandes volumes de dados ou acesso concorrente.

Esta fase é particularmente eficaz porque atua sobre os detalhes de acesso físico à tabela, sendo uma otimização de baixo nível, mas com alto impacto.

#### 2.4.6. Resultados

Tabela 4: Resultados Finais e Análise Comparativa da *query* 4

| Fase de Otimização | Tempo de Execução (s) | Melhoria Relativa | Melhoria Acumulada |
|---|---|---|---|
| Original | 65.163 s | - | - |
| 1ª Fase | 19.730 s | 69.73% | 69.73% |
| 2ª Fase | 17.885 s | 9.36% | 72.55% |
| 3ª Fase | 6.978 s | 60.96% | 89.29% |
| 4ª Fase | 5.564 s | 20.26% | 91.46% |

### 2.5. Interrogação analítica 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;
```

#### 2.5.1. Descrição da Query

Esta interrogação tem a finalidade de devolver uma lista de 1000 utilizadores com mais tempo acumulado (em horas) num determinado jogo, identificado pelo ID 730. Esta extração é feita com base nos dados armazenados nas tabelas `game`, `library` e `users`.

A interrogação começa por selecionar da tabela `users` os campos `id`, `username` e `country`, seleciona também o tempo de jogo (dividido por 60 e convertido para inteiro) e a data em que o jogo foi adicionado à biblioteca.

Através de `JOIN` entre as tabelas game (que contém os jogos disponíveis), `library` (que relaciona os utilizadores com os jogos que têm na sua biblioteca e com o tempo jogado) e `users` (que contém a informação dos utilizadores) a `query` obtém os dados acima referidos.

Com a cláusula `WHERE` são impostas três restrições fundamentais:

- Para filtrar a procura para um jogo, existe a condição g.id = 730;
- Para garantir que apenas são considerados utilizadores que efetivamente jogaram o jogo, existe a condição l.playtime > 0;
- Para excluir que utilizadores que tenham sido punidos com o sistema de detenção de *cheats* (*VAC*), existe a condição `NOT u.vac_banned`.

Por fim, os dados são ordenados por tempo de jogo de forma decrescente e, em caso de empate, pela data de adição do jogo à biblioteca, também de forma decrescente. Finalmente, a cardinalidade do resultado é restringida a no máximo os 1000 utilizadores mais relevantes consoante os critérios de ordenação.

### 2.5.2. Análise do Plano de Execução

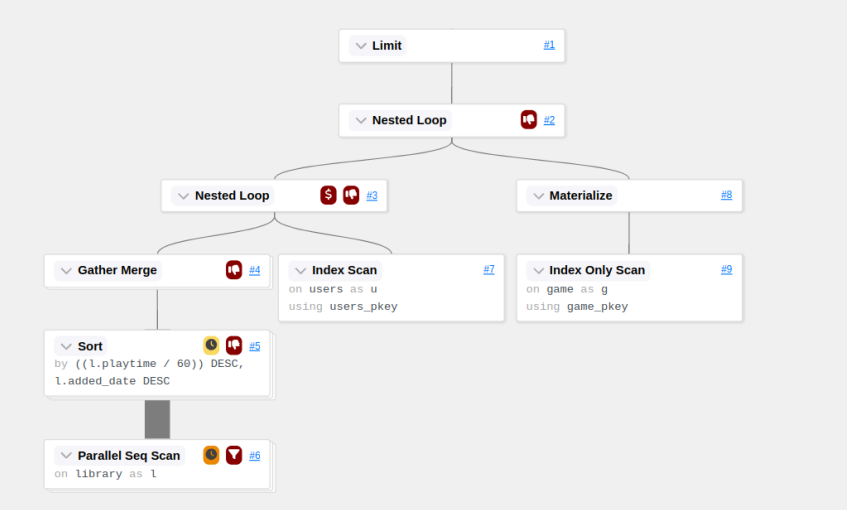


Figura 13: Plano de execução da *baseline* da *query* 5 antes das otimizações.

A *query* apresenta um plano de execução com cerca de 7s881ms, sendo este tempo devido, em grande parte, ao volume de dados que está a ser processado e pela complexidade das operações que estão envolvidas. Neste caso, a operação de ordenação `Sort` é a principal fonte de atraso, uma vez que é aplicada aos dados da tabela `library`, que foram previamente filtrados com base nas condições `playtime > 0` e `game_id = 730`.

Esta tabela `library` foi percorrida pelo `Parallel Sequential Scan`, sendo dividido o trabalho com 2 *workers*. Mesmo que exista o paralelismo, o número de linhas processado foi muito elevado, cerca de 40 milhões, das quais apenas 2,7 milhões foram satisfizeram o filtro. Estes dados, que posteriormente foram ordenados com base no tempo de jogo, em minutos, e na data de adição, implicaram que fosse utilizado o método de `external merge`. Isto diz-nos que a ordenação excedeu a memória disponível e recorreu ao disco temporário, o que faz com o que o tempo de execução aumente.

Após o `Sort`, foi utilizada a operação `Gather Merge` de modo a combinar os resultados ordenados das várias *threads* sequenciais. Embora que para preservar a ordenação antes de aplicar o `LIMIT`, esta operação é necessária, torna-se também uma fonte de latência adicional.

Finalmente são efetuadas duas junções através do `Nested Loop`. O primeiro, utilizando um `Index Scan` na chave primária e aplicando um filtro adicional para excluir utilizadores banidos, junta os dados da `library` com a tabela dos `users`.

A segunda, usando um `Index Only Scan`, é uma junção com a tabela game. No entanto, estas junções são aplicadas após a ordenação dos dados, o que significa que a maior parte do custo da *query* já foi incorrido à cabeça.

Concluindo, o tempo de execução da *query* é dominado por dois fatores principais: o *scan* paralelo e a filtragem da tabela *library*, e a subsequente ordenação externa dos dados filtrados.

### 2.5.3. Otimizações

#### 2.5.3.1. 1ª Fase de Otimização

Originalmente, a consulta que converte o tempo de jogo de minutos para horas era feita diretamente na *query SQL* através da expressão:

```
(l.playtime / 60)::int AS hours
```

No entanto esta expressão não pode indexada uma vez que o *PostgreSQL* não consegue aplicar automaticamente um índice a uma transformação de coluna feita *on-the-fly*. O *PostgreSQL* faz o cálculo para cada linha no momento da execução da consulta. Como o valor não existe como uma coluna real na tabela, não há como criar um índice sobre ele.

Assim, para melhorar a legibilidade e a eficiência das consultas relativas ao tempo de jogo, foi adicionada à tabela *library* uma nova coluna computada chamada *playtime_hours*.

```
ALTER TABLE library
ADD COLUMN playtime_hours INT GENERATED ALWAYS AS ((playtime / 60)::int) STORED;
```

Esta coluna é do tipo `INT` e é gerada automaticamente com base no valor existente em `playtime`, convertendo os minutos registados para horas através da expressão `(playtime / 60)::int`.

Além disso, como se trata de uma coluna `STORED`, o valor é fisicamente armazenado na base de dados. Com isto, consultas que necessitem de filtrar, agrupar ou ordenar por horas jogadas podem beneficiar de índices, aumentando o desempenho geral.

#### 2.5.3.2. 2ª Fase de Otimização

Para otimizar o desempenho das consultas mais frequentes sobre as tabelas `library` e `users`, foram criados novos índices direcionados às condições de filtragem e ordenação utilizadas.

A primeira melhoria consistiu na criação de um índice composto sobre a tabela `users`, abrangendo as colunas `id` e `vac_banned`.

```
CREATE INDEX CONCURRENTLY idx_users_id_vac_banned ON users (id) INCLUDE
(vac_banned);
```

Este índice (`idx_users_id_vac_banned`) permite que a cláusula `WHERE NOT u.vac_banned`, combinada com a junção por `u.id`, seja resolvida de forma mais eficiente, evitando a leitura completa da tabela.

A segunda melhoria foi feita sobre a tabela `library`, com a criação do índice `idx_library_fast`, foi criado um índice composto que inclui as colunas `game_id`, `playtime_hours DESC`, `added_date DESC` e `user_id`.

```
CREATE INDEX CONCURRENTLY idx_library_fast ON library (game_id, playtime_hours
DESC, added_date DESC, user_id) WHERE playtime > 0;
```

Este índice foi criado para acelerar consultas que filtram por `game_id`, verificam se `playtime > 0` e ordenam os resultados por tempo de jogo decrescente e data de adição mais recente.

Este índice permite que a consulta seja satisfeita utilizando exclusivamente o índice, recorrendo a um Index Only Scan, o que evita acessos ao disco para ler as linhas completas da tabela.

#### 2.5.3.3. 3ª Fase de Otimização

De seguida o último passo deste processo foi reescrever a *query* para utilizar a nova coluna `playtime_hours`.

```
SELECT u.id, u.username, u.country, l.playtime_hours 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 l.playtime_hours DESC, l.added_date DESC
LIMIT 1000;
```

Esta modificação permite que o plano de execução use o novo índice. Assim evita cálculos em tempo de execução que impediriam a sua utilização.

Com esta alteração, a ordenação pode ser feita diretamente com base na estrutura do índice, o que permite aplicar o `LIMIT` de forma muito mais eficiente.

### 2.5.4. Resultados

Tabela 5: Resultados Finais e Análise Comparativa da *query* 5

| Fase de Otimização | Tempo de Execução (s) | Melhoria Relativa | Melhoria Acumulada |
|---|---|---|---|
| Original | 3.858 s | - | - |
| 1ª e 2ª Fase | 2.187 s | 43.3% | 43.3% |
| 3ª Fase | 0.017 s | 99.2% | 99.6% |

# 3. Carga Transacional

## 3.1. Otimização através de índices

### 3.1.1. gameReviews()

A *query* `gameReviews` divide-se em duas *queries* principais: `getGameScore` e `getGameRecentReviews`. O processo de otimização seguiu uma abordagem metódica, analisando os planos de execução iniciais, implementando otimizações específicas, e posteriormente avaliando o impacto das alterações realizadas. Para os testes de otimização, utilizamos o jogo com identificador 783770.

#### 3.1.1.1. getGameRecentReviews

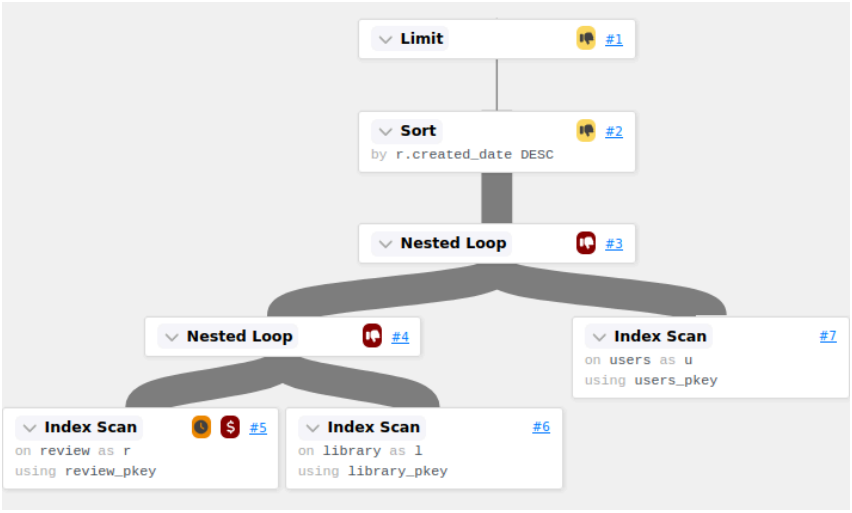


Figura 14: Plano de execução da *baseline* da *query* `getGameRecentReviews` antes das otimizações

A *query* `getGameRecentReviews` tem como objetivo apresentar as 25 avaliações mais recentes de um determinado jogo, incluindo informações do utilizador e do seu tempo de jogo. Ao analisar o plano de execução inicial, identificamos várias ineficiências: O plano inicial um tempo de execução de 211.267 ms. A operação mais dispendiosa era o `Index Scan` na tabela `review`, seguido de `Nested Loops` para junção com as tabelas `library` e `users`. O ordenamento por data de criação estava a ser realizado após a recuperação de todos os registos correspondentes ao jogo, o que significa que primeiro todos os registos eram lidos e só depois ordenados. Esta abordagem é ineficiente quando apenas as 25 entradas mais recentes são necessárias.

Considerando estas observações, implementamos um índice composto que abrange as colunas `game_id` e `created_date`:

```
CREATE INDEX idx_review_game_date ON review(game_id, created_date DESC);
```

Este índice foi concebido com o intuito de otimizar simultaneamente a filtragem pelo identificador do jogo e a ordenação pela data de criação em ordem descendente. A inclusão da ordenação `DESC` no índice é crucial, pois corresponde diretamente à necessidade da *query* de apresentar as avaliações mais recentes primeiro.

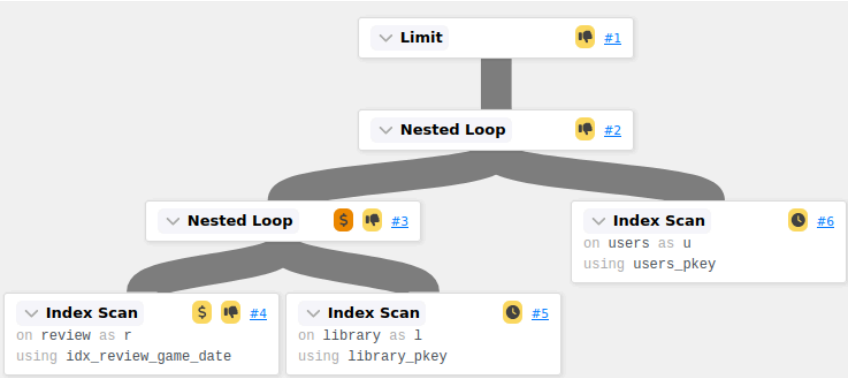


Figura 15: Plano de execução da *baseline* da *query* `getGameRecentReviews` após otimizações

O tempo de execução diminuiu de 211.267 ms para apenas 0.266 ms, representando uma melhoria de aproximadamente 99.9%. O número de *buffers* acedidos reduziu drasticamente, de 48919 (7367 *hit* + 41552 *read*) para apenas 253 *hits*, sem qualquer leitura direta do disco.

#### 3.1.1.2. getGameScore

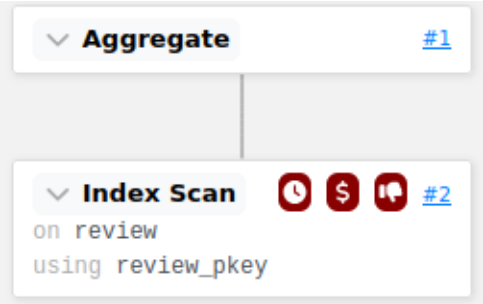


Figura 16: Plano de execução da *baseline* da *query* `getGameScore` antes das otimizações

A *query* `getGameScore` calcula a percentagem de avaliações positivas para um determinado jogo. O plano de execução inicial apresentava um tempo de execução de 167.981 ms, com um *scan* de índice na tabela `review`, mas ainda assim com um custo elevado.

Para otimizar esta *query*, criamos um índice que incluía apenas a coluna necessária para o cálculo:

```
CREATE INDEX idx_review_game_recommend ON review(game_id) INCLUDE(recommend);
```

Este índice utiliza a funcionalidade `INCLUDE`, que permite adicionar colunas não-chave ao índice. Desta forma, o PostgreSQL pode realizar um `Index Only Scan`, evitando o acesso à tabela base para obter os valores da coluna `recommend`. Após criar o índice, notamos que havia um número significativo de *heap fetches*, o que indicava que o índice não estava a ser utilizado de forma totalmente eficiente. Executamos então um `VACUUM ANALYZE` na tabela `review`. Esta operação atualizou as estatísticas da tabela e limpou os registos obsoletos, permitindo que o *PostgreSQL* utilizasse o índice de forma mais eficaz.

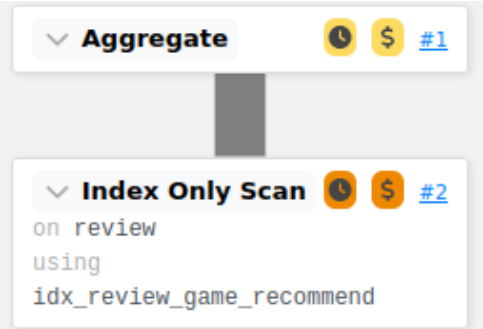


Figura 17: Plano de execução da *baseline* da *query* `getGameScore` após otimizações

O tempo de execução diminuiu de 167.981 ms para 0.250 ms, uma melhoria substancial. O plano de execução passou a mostrar um `Index Only Scan`, confirmando que todas as informações necessárias estavam a ser obtidas diretamente do índice, sem necessidade de aceder à tabela base (*heap fetches*: 0).

#### 3.1.1.3. Solução Final

Após a implementação dos dois índices, observamos um efeito colateral não desejado: a *query* `getGameRecentReviews` deixou de utilizar o índice criado especificamente para ela, passando a utilizar o índice `idx_review_game_recommend`. Isto resultou num aumento do tempo de execução para 13 ms, significativamente superior aos 0.266 ms conseguidos anteriormente.

Para resolver este problema e otimizar ambas as *queries* simultaneamente, criamos um índice único que satisfizesse os requisitos de ambas, apagando os índices anteriores:

```
  CREATE INDEX idx_review_game_date_recommend ON review(game_id, created_date
DESC) INCLUDE (recommend, text);
```

Este índice combina o melhor dos dois mundos: mantém a estrutura de ordenação necessária para a *query* `getGameRecentReviews`, inclui as colunas adicionais necessárias para a *query* `getGameScore` e adiciona também a coluna `text` para evitar acessos à tabela base na primeira *query*.

#### 3.1.1.4. Resultados

Para a *query* `getGameRecentReviews`, o tempo de execução manteve-se excelente, em 0.279 ms, ligeiramente superior ao resultado da primeira otimização, mas ainda assim uma melhoria significativa em relação ao valor inicial. Para a *query* `getGameScore`, o tempo de execução foi de 0.329 ms, um pouco mais elevado que o resultado da segunda otimização, mas ainda assim muito inferior ao valor original. O mais importante foi o resultado do *benchmark* completo para a função `gameReviews()`, que combina ambas as *queries*.

Tabela 6: Resultados Finais e Análise Comparativa das Otimizações de *query* transacional `getGameReviews`

| Fase de Otimização | Tempo de Execução (s) | Melhoria Relativa | Melhoria Acumulada |
|---|---|---|---|
| Baseline sem otimizações | 1.70 | - | - |
| Com o 1º índice | 0.04 | 97.70% | 97.70% |
| Com os 2 índices separados | 0.06 | −65.11% | 96.21% |
| Com o índice unificado | 0.01 | 76.90% | 99.12% |

### 3.1.2. userInfo()

A *query* `userInfo` divide-se em duas *queries* principais: `getUserInfo` e `getUserTopGames`. O processo de otimização seguiu uma abordagem metódica, analisando os planos de execução iniciais, implementando otimizações específicas, e posteriormente avaliando o impacto das alterações realizadas.

#### 3.1.2.1. getUserInfo

A *query* `getUserInfo` tem como objetivo obter os dados de perfil de um utilizador a partir do seu id. Trata-se de uma operação simples e frequente, com impacto direto na experiência de utilização da aplicação, nomeadamente na construção de perfis e *dashboards* personalizados.

Sendo esta *query* um simples `SELECT`, decidimos testar a criação do seguinte índice:

```
CREATE INDEX idx_user_profile_summary
ON users(id, username, created_date, vac_banned, profile_description, country);
```

A ideia seria permitir um `Index Only Scan`, onde toda a informação necessária já se encontrasse diretamente no índice, evitando assim acessos à tabela base (`Heap Fetches`), o que poderia resultar numa redução do tempo de execução — sobretudo em ambientes de alta carga ou com muitas leituras concorrentes.

No entanto, após análise do plano de execução, verificamos que o *PostgreSQL* continuou a optar por utilizar o índice primário (users_pkey) em vez do novo índice composto. Este comportamento deve-se a vários fatores:

- O índice primário (`id`) é mais pequeno e mais eficiente para buscas por chave única.
- A consulta retorna apenas uma linha, o que reduz o impacto da leitura da tabela.
- O *PostgreSQL* pode não ter considerado o índice composto mais vantajoso.

Desta forma, concluímos que, para esta *query* em específico, a criação do índice `idx_user_profile_summary` não traz benefícios práticos de desempenho. A estrutura atual já é suficientemente eficiente para este tipo de acesso pontual.

#### 3.1.2.2. getUserTopGames

A *query* `getUserTopGames` tem como objetivo apresentar os 5 jogos mais jogados por um determinado utilizador, ordenados por tempo de jogo (*playtime*). Esta informação é frequentemente utilizada em perfis de utilizador e *dashboards* personalizados.

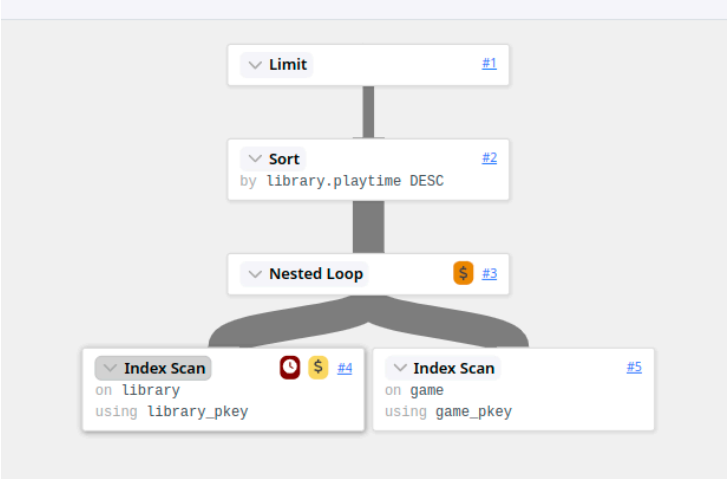


Figura 18: Plano de execução da *baseline* da *query* `getUserTopGames` antes das otimizações

Inicialmente, a execução da *query* apresentava um custo considerável devido ao uso de `Sort` e junções em `Nested Loop` após a leitura de todos os registos da `library` daquele utilizador. O plano de execução incluía:

Um `Index Scan` por `user_id` na `library`

Um ordenamento (`Sort`) para aplicar o `ORDER BY playtime DESC`

Junção com a tabela game para obter o nome do jogo

Para otimizar a *query*, foi criado o seguinte índice composto:

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

Este índice foi desenhado para suportar diretamente o filtro por `user_id`, a ordenação por `playtime DESC`, e também incluir o `game_id` necessário para a junção com a tabela game. O objetivo era permitir a utilização de um `Index Only Scan`, evitando a leitura completa da tabela e o custo adicional de ordenação.

Após aplicar o índice, verificou-se uma melhoria significativa no plano de execução:

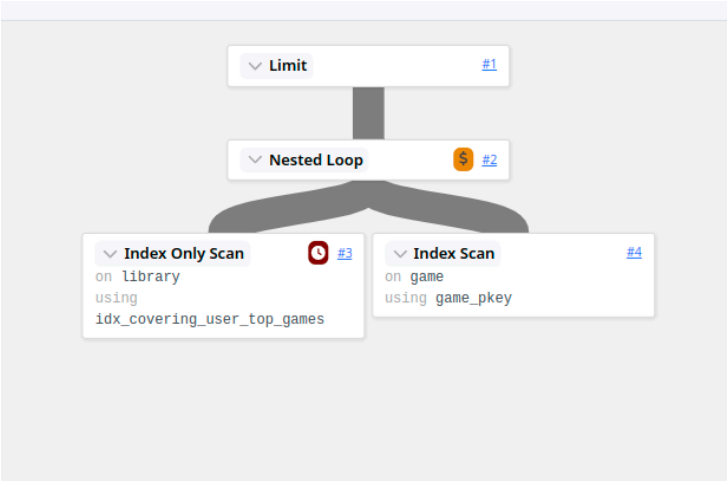


Figura 19: Plano de execução da *baseline* da *query* `getUserTopGames` após otimizações

O *PostgreSQL* passou a usar um `Index Only Scan`, com `Heap Fetches: 0`

O custo total da *query* foi reduzido substancialmente (de 3.8 ms para 1.2 ms)

A quantidade de blocos lidos e acessos a disco diminuiu consideravelmente.

Com este índice, a *query* tornou-se consideravelmente mais eficiente, aproveitando ao máximo as capacidades do *PostgreSQL* em otimizações baseadas em índices. Assim, a criação do índice `idx_covering_user_top_games` revelou-se altamente benéfica para esta operação, reduzindo tempos de resposta e carga de *I/O*.

### 3.1.3. buyGame

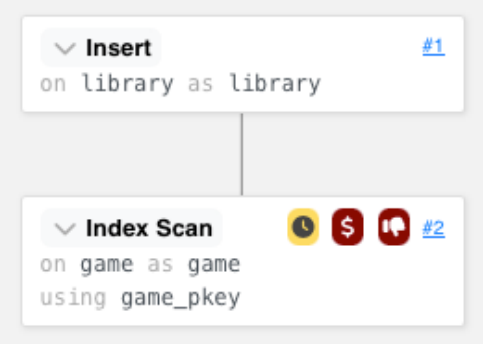


Figura 20: Plano de execução da *baseline* da *query* `buyGame` antes das otimizações

Com base no plano da *query*, conseguimos perceber que a *query* faz um `Index Scan` no game usando `game_pkey` (que é o índice implícito da `PK id`) de forma a recuperar a coluna *price*.

Assim, temos que o índice atual (`game_pkey`) permite encontrar a linha, mas como o *price* não está incluído diretamente no índice, o *PostgreSQL* acaba por fazer um acesso adicional à tabela base (*heap*) para buscar *price*. Desta forma, criamos o seguinte índice:

```
CREATE INDEX idx_game_id_include_price ON game(id) INCLUDE (price);
```

Este índice permite que o *price* venha diretamente do índice, sem aceder à tabela base. Permitimos ao PostgreSQL fazer um `Index-Only Scan` (em vez de `Index Scan` + *heap lookup*).

#### 3.1.3.1. Resultados

Para a *query* `getUserInfo`, o tempo de execução manteve-se igual ao inicial, em cerca de 0.8 ms. Para a *query* `getUserTopGames`, o tempo de execução passou de cerca de 3.8ms para 1.2ms. O mais importante foi o resultado do *benchmark* completo para a função `userInfo()`, que combina ambas as *queries*.

Tabela 7: Resultados Finais e Análise Comparativa das Otimizações de *query* transacional `userInfo`

| Fase de Otimização | Tempo de Execução (ms) | Melhoria Relativa | Melhoria Acumulada |
|---|---|---|---|
| Baseline sem otimizações | 6.334 | - | - |
| Com o índice | 2.430 | 61% | 61% |

### 3.1.4. recentGamesPerTag()

#### 3.1.4.1. getRecentGamesPerTag

Esta *query* tem como objetivo de recuperar informações sobre jogos: o id, o nome e a data de lançamento; que estarão associados a uma determinada `tag`, ordená-los pela data de lançamento de forma decrescente e limitá-los a 25 resultados. A estrutura da *query* envolve três tabelas: `tag`, `games_tags` e `game`. A relação entre estas tabelas é estabelecida através de `joins`, sendo que a tabela `games_tags` funciona como uma tabela intermediária entre `tag` e `game`. O filtro é aplicado através do `WHERE`, retornando todos os jogos associados a uma `tag` específica.

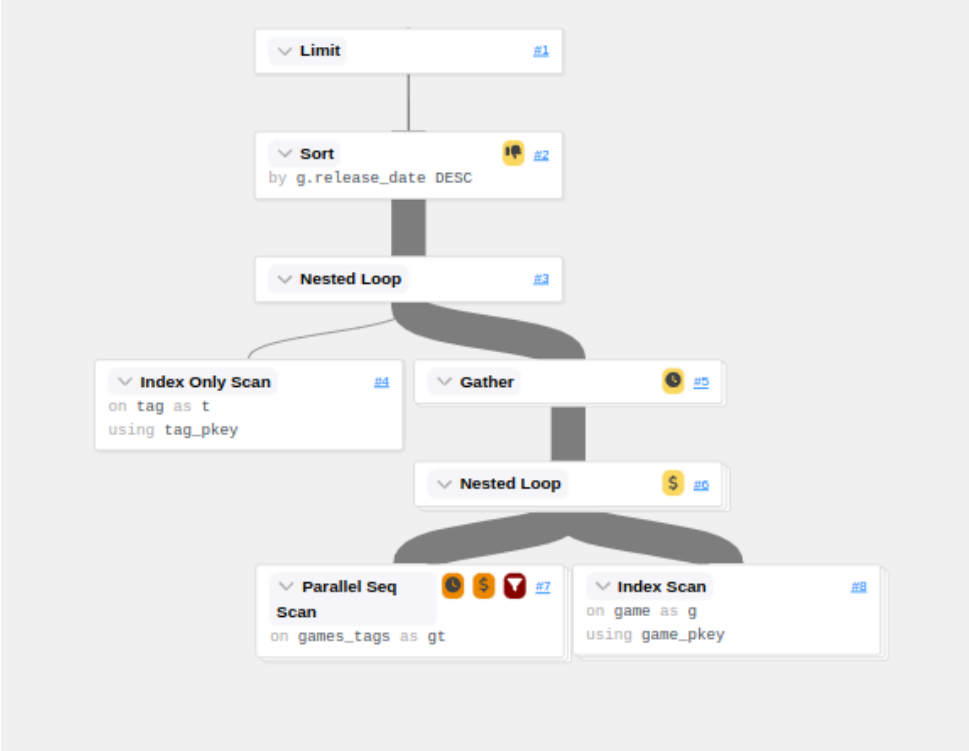


Figura 21: Plano de execução da *baseline* da *query* `getRecentGamesPerTag` antes das otimizações.

O plano inicial tem um tempo de execução de 19.059ms. A operação mais dispendiosa é o `Parallel Seq Scan`, na tabela `games_tag`. A ordenação pela data em que foram adicionados em ordem decrescente está também a ser realizada após a recuperação de todos os registos. Isto torna-se eficiente quando são apenas necessárias as 25 entradas mais recentes.

Tendo isto em conta, foi implementado o índice que abrange a coluna `release_date`.

```
CREATE INDEX idx_game_release_date ON game(release_date DESC);
```

Este índice tem como intuito otimizar a filtragem pela data de lançamento. Isso permite ao *PostgreSQL* realizar a ordenação sem precisar de carregar todos os dados para a memória, otimizando o processo de ordenação.

Outra otimização realizada foi a criação de um índice composto nas tabelas envolvidas nos `joins`. Pela *query* original, o *PostgreSQL* poderia estar realizar pesquisas completas nas tabelas. Para resolver isso, foi criado um índice composto na tabela `games_tags` para otimizar a junção entre `tag` e `game`, considerando os campos `tag_id` e `game_id` simultaneamente. Esta alteração permite que a base de dados encontre mais rapidamente os pares de `tags` e `games` associados.

```
CREATE INDEX idx_games_tags_tag_id_game_id  ON games_tags(tag_id, game_id);
```

Assim, a necessidade de realizar *scans* completos nas tabelas é reduzida.

#### 3.1.4.2. Resultados

Para a *query* `getRecentGamesPerTag()`, o tempo de execução reduziu para 4.552ms. Sendo que a função `RecentGamesPerTag` apenas utiliza uma *query*, o *benchmark* completo tornou-se simples de realizar, realizando testes para a *query default*, depois realizando testes após a inserção de índices e finalmente após a restruturação de código.

Tabela 8: Resultados Finais e Análise Comparativa das Otimizações de *query* transacional `recentGamesPerTag`

| Fase de Otimização | Tempo de Execução (ms) | Melhoria Relativa | Melhoria Acumulada |
|---|---|---|---|
| Baseline sem otimizações | 18.803 | - | - |
| Com os índices | 5.687 | 69.75% | 69.75% |

### 3.1.5. searchGames()

#### 3.1.5.1. getGamesByTitle

Esta *query* tem como objetivo recuperar o ID, nome e data de lançamento dos jogos cujo nome corresponde a uma expressão de pesquisa textual, utilizando os recursos de `Full-Text Search` do *PostgreSQL*. A busca é realizada com base em `to_tsvector` e `to_tsquery`, com suporte à língua inglesa. O filtro de correspondência textual é aplicado diretamente no `WHERE`, limitando os resultados a 25 jogos.

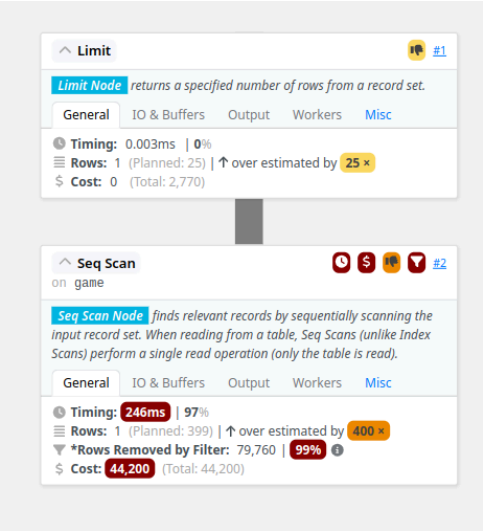


Figura 22: Plano de execução da *baseline* da *query* `getGamesByTitle` antes das otimizações.

Inicialmente, o plano de execução recorria a um `Seq Scan` na tabela game, analisando todas as 80.000 entradas para encontrar possíveis correspondências.

```
SELECT id, name, release_date
FROM game
WHERE to_tsvector('english', name) @@ to_tsquery('english', ?)
LIMIT 25;
```

O plano inicial teve um tempo de execução de 253.364 ms, devido à ausência de índices adequados, resultando numa varredura completa da tabela e aplicação do filtro `to_tsvector` linha a linha.

Para otimizar esta situação, foi criado um índice `GIN` sobre a coluna `name`, usando `to_tsvector`, o que permite acelerar significativamente este tipo de pesquisa textual.

```
CREATE INDEX idx_game_name_fts ON game USING GIN (to_tsvector('english', name));
```

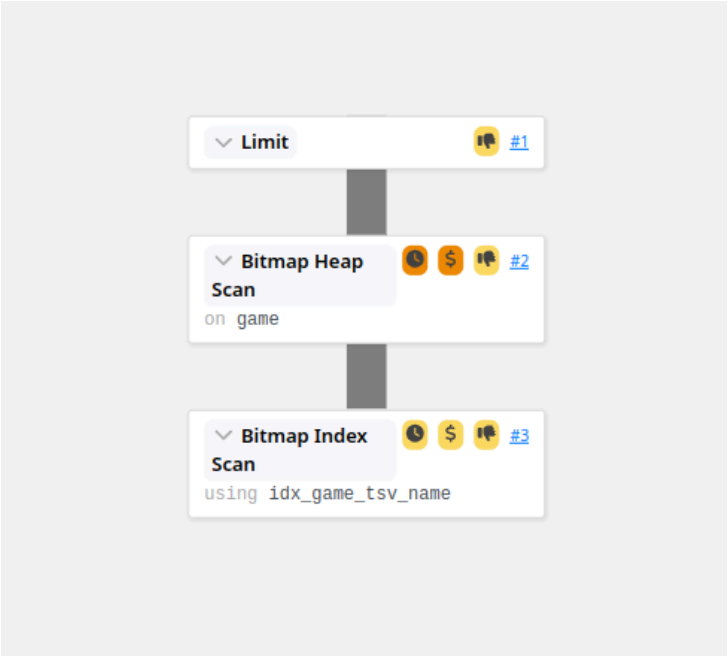


Figura 23: Plano de execução da *baseline* da *query* `getRecentGamesPerTag` após as otimizações.

Após a criação do índice, o plano de execução passou a utilizar um `Bitmap Index Scan`, com reaproveitamento eficiente do índice, reduzindo o tempo de execução para **0.029 ms**.

#### 3.1.5.2. getGamesSemantic

Esta *query* realiza uma procura **semântica** por jogos, ordenando os resultados pela **proximidade vetorial** entre os `embeddings` do jogo e um vetor de consulta armazenado na tabela `query_embedding_sample`.

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

Esta operação utiliza a extensão `pgvector` para calcular a distância vetorial (), normalmente a distância Euclidiana (`vector_l2_ops`), ordenando os jogos mais semanticamente similares ao vetor de consulta.

O tempo de execução inicial era 1.670 ms — já bastante eficiente. Como tentativa de melhoria, foi criado um índice vetorial com `ivfflat` para acelerar a ordenação por proximidade:

```
CREATE INDEX ON game_search_embedding USING ivfflat (embedding vector_l2_ops)
WITH (lists = 200);
```

No entanto, após testes, verificou-se que **o uso do índice `ivfflat` não teve impacto no desempenho da query**. Isto deve-se ao facto de que índices `ivfflat` só são utilizados quando o operador `<->` é combinado com um **filtro explícito** no vetor de comparação. Como o `ORDER BY` é aplicado sem `WHERE` (i.e., sem restringir previamente o conjunto de *embeddings* a comparar), o PostgreSQL continua a optar por um `Seq Scan` ou ordenação na totalidade dos dados.

#### 3.1.5.3. Resultados

A função `searchGames()` é composta por duas *subqueries* que representam cenários distintos de otimização.

Tabela 9: Resultados Finais e Análise Comparativa das Otimizações de *query* transacional `searchGames`

| Query | Tempo Inicial (ms) | Tempo Após Otimização (ms) | Melhoria Relativa |
|---|---|---|---|
| getGamesByTitle | 253.364 | 0.029 | 99.99% |
| getGamesSemantic | 1.670 | ≈1.6 | 0% |

### 3.1.6. Resultados Obtidos

Tabela 10: Resultados Finais e Análise Comparativa das otimizações realizadas

| Métrica | Sem índices | Apenas índices Analíticos | Apenas índices Transacionais | Com Índices Analíticos e Transacionais |
|---|---|---|---|---|
| AddPlaytime (ms) | 2.335 | 7.360 | 3.581 | 7.332 |
| ReviewGame (ms) | 2.099 | 2.269 | 4.173 | 4.098 |
| BuyGame (ms) | 1.779 | 11.038 | 2.852 | 10.484 |
| NewFriendship (ms) | 2.795 | 6.330 | 3.209 | 5.929 |
| NewGame (ms) | 4.024 | 3.927 | 3.889 | 3.947 |
| NewUser (ms) | 2.786 | 2.915 | 2.698 | 2.767 |
| GameInfo (ms) | 1.925 | 2.164 | 1.865 | 1.918 |
| GameReviews (ms) | 1618.161 | 961.483 | 16.548 | 13.774 |
| UserInfo (ms) | 6.418 | 6.925 | 2.537 | 3.706 |
| RecentGamesPerTag (ms) | 173.833 | 190.172 | 193.027 | 191.529 |
| SearchGames (ms) | 1062.820 | 1214.188 | 1001.943 | 976.505 |
| Throughput (txn/s) | 102.844 | 123.794 | 225.172 | 221.539 |
| Response Time (ms) | 155.390 | 128.754 | 71.106 | 72.084 |
| Abort Rate (%) | 0.459 | 0.372 | 0.385 | 0.391 |

A análise das últimas três métricas (**Throughput**, **Response Time** e **Abort Rate**) revela que o uso de índices transacionais (com ou sem os analíticos) proporciona o melhor desempenho global do sistema. O **Throughput** atinge o máximo com apenas índices transacionais (225.172 txn/s), evidenciando maior capacidade de processamento. O **Response Time** também é significativamente menor nesse cenário (71.106 ms), indicando maior eficiência na resposta às transações. Já a **Abort Rate** mantém-se relativamente baixa em todos os cenários, mas é ligeiramente superior sem índices (0.459%) e menor com apenas índices analíticos (0.372%). Conclui-se que os índices transacionais são cruciais para

desempenho, enquanto os analíticos têm um impacto mais modesto relativamente aos resultados das *queries* transacionais (*benchmark*).

## 3.2. Otimização através de parâmetros

### 3.2.1. shared_buffers

O parâmetro `shared_buffers` determina a quantidade de memória reservada pelo PostgreSQL para armazenar dados em cache, permitindo acessos mais rápidos às informações frequentemente utilizadas.

A recomendação genérica do PostgreSQL para configurar o `shared_buffers` é de aproximadamente 25% da memória total do sistema.

Tabela 11: Análise do uso de memória por parte da VM

| Category | Total | Used | Free | Shared | buff_cache | Available |
|---|---|---|---|---|---|---|
| Mem | 15988 | 857 | 3833 | 140 | 11776 | 15130 |

Assim, o valor teórico recomendado seria de aproximadamente 4GB para o `shared_buffers` (25% de 15GB).

Para identificar o valor ótimo, realizamos testes de benchmark com seis configurações diferentes: 128MB, 256MB, 512MB, 1GB, 2GB e 4GB.

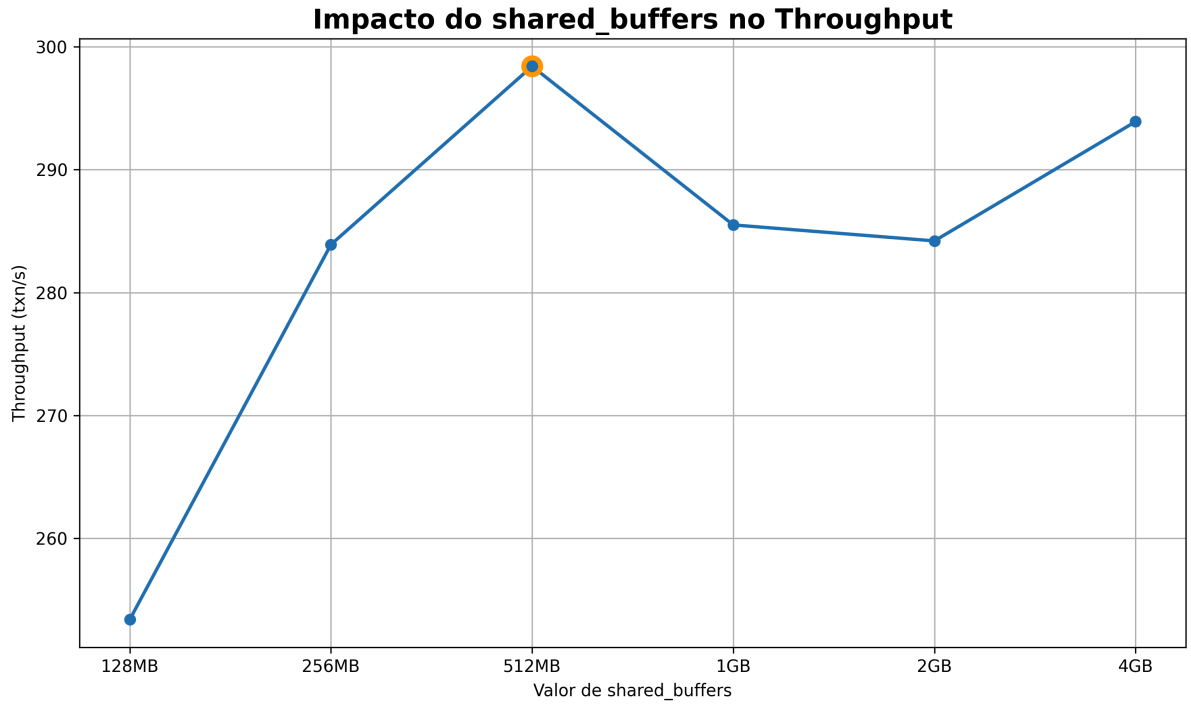


Figura 24: Gráfico demonstrativo do impacto do shared_buffers no throughput

Com um `shared_buffers` de 128MB, o sistema processou cerca de 253,4 transações por segundo, com um tempo médio de resposta de aproximadamente 63 ms. Ao aumentar para 256MB, verificou-se uma melhoria de aproximadamente 12%, atingindo 283,9 transações por segundo e reduzindo o tempo de resposta para 56,4 ms. O ponto ótimo foi alcançado com a configuração de 512MB, que proporcionou um *throughput* de 298,4 transações por segundo e um tempo de resposta de 53,6 ms. Esta configuração representa um aumento de 17,7% em relação ao valor base de 128MB.

No nosso caso, a utilização de 512MB para o `shared_buffers` representa aproximadamente 3,4% de uma memória total de 15GB, significativamente abaixo da recomendação de 25%. Isto sugere que o nosso sistema beneficia mais da utilização da memória para outros fins, possivelmente para a cache do sistema operativo ou para outros processos concorrentes.

### 3.2.2. work_mem

Após a otimização bem-sucedida do parâmetro `shared_buffers`, direcionamos a nossa atenção para o parâmetro `work_mem`, que constitui outro elemento crucial na configuração do PostgreSQL para maximizar o desempenho. O `work_mem` determina a quantidade de memória que cada operação de ordenação e *hash* pode utilizar antes de recorrer a ficheiros temporários em disco. Este parâmetro influencia diretamente a eficiência de *queries* que requerem operações como `ORDER BY`, `GROUP BY`, `DISTINCT`, ou junções que necessitem de tabelas *hash*, como é o caso de, por exemplo, *queries* como `getGameRecentReviews`, `getUserTopGames`, entre outras.

Para identificar o valor ótimo de `work_mem`, mantivemos o `shared_buffers` no valor ótimo previamente determinado (512MB) e realizamos testes com cinco configurações diferentes: 4MB, 8MB, 16MB, 32MB e 64MB.

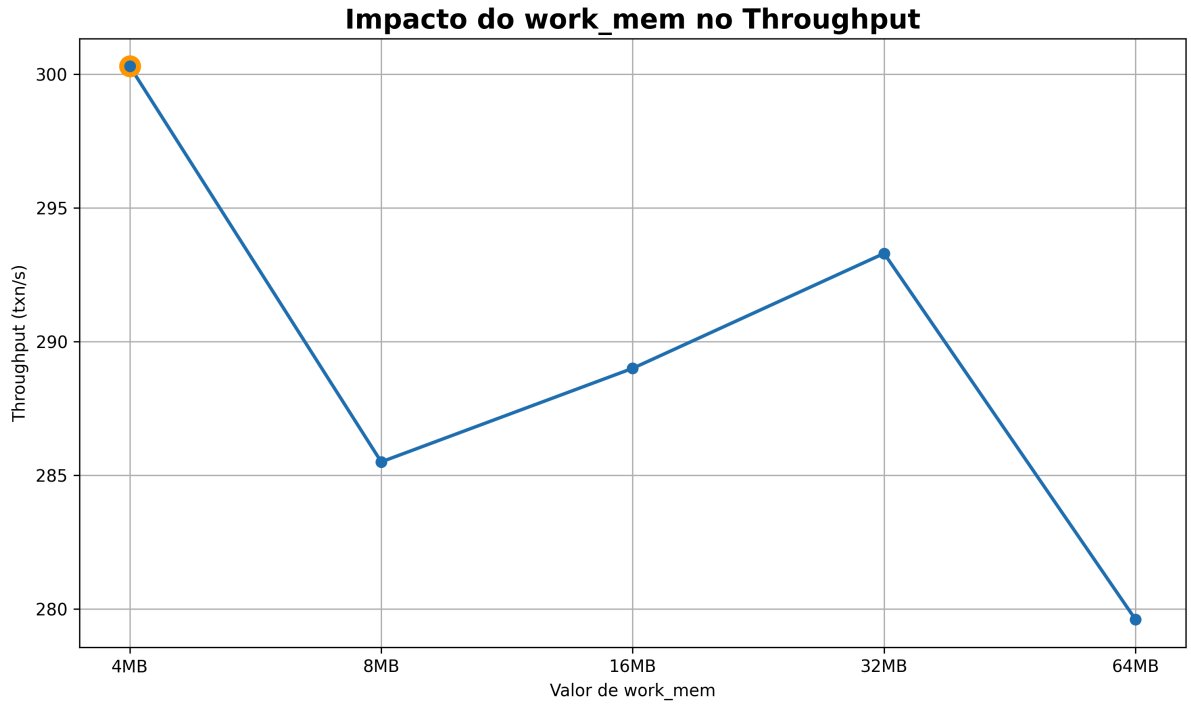


Figura 25: Gráfico demonstrativo do impacto do work_mem no throughput

Os resultados dos testes revelam que o valor mais reduzido de `work_mem` testado, 4MB, proporcionou o melhor desempenho global, com um *throughput* de 300,3 transações por segundo e um tempo médio de resposta de 53,4 milissegundos. Este resultado contraria a intuição inicial de que valores mais elevados deste parâmetro beneficiariam o desempenho, especialmente considerando a presença de *queries* que saem beneficiadas com este parâmetro.

Esta tendência decrescente no desempenho com o aumento do `work_mem` indica que, para o nosso *workload* específico, a alocação excessiva de memória para operações individuais pode estar a prejudicar o desempenho global do sistema, possivelmente devido à concorrência por recursos de memória entre múltiplas sessões ativas.

### 3.2.3. Outros

Para além dos parâmetros `shared_buffers` e `work_mem`, que demonstraram melhorias significativas de desempenho conforme documentado anteriormente neste relatório, a nossa equipa conduziu testes adicionais com outros parâmetros de configuração do *PostgreSQL*.

Realizamos uma análise exaustiva dos parâmetros `effective_cache_size`, `maintenance_work_mem` e `random_page_cost`, aplicando a mesma metodologia de *benchmarking* utilizada nos testes anteriores. Testámos diversos valores para cada um destes parâmetros, monitorizando cuidadosamente métricas de desempenho como *throughput*, tempo de resposta e *abort rate*.

Ao contrário do que observámos com `shared_buffers` e `work_mem`, as variações nestes parâmetros adicionais não resultaram em melhorias significativas de desempenho no nosso ambiente específico. Em alguns casos, observamos até ligeiras degradações no desempenho quando os valores foram alterados relativamente às configurações predefinidas.

Considerando que o objetivo principal deste relatório é documentar as otimizações que efetivamente contribuíram para melhorias de desempenho, optamos por excluir a análise detalhada destes parâmetros.

### 3.2.4. Paralelismo

Após a otimização do `shared_buffers`, direcionamos a análise para o paralelismo no *PostgreSQL*, implementando configurações para melhorar o processamento de *queries* complexas. Primeiramente, configuramos os parâmetros de paralelismo:

```
ALTER SYSTEM SET max_worker_processes = 16;
ALTER SYSTEM SET max_parallel_workers = 16;
SET max_parallel_workers_per_gather = 4;
SET parallel_tuple_cost = 0.1;
SET parallel_setup_cost = 100;
SET min_parallel_table_scan_size = '1MB';
SET min_parallel_index_scan_size = '512kB';
```

Estas configurações aumentaram significativamente o *throughput* para 328,289 transações por segundo, com tempo médio de resposta de 48,847 ms - uma melhoria notável em relação aos testes anteriores. Contrariando os resultados iniciais do `work_mem`, onde valores menores (4MB) apresentaram melhor desempenho, decidimos retestar este parâmetro no contexto do paralelismo. Aumentando o `work_mem` para 64MB, obtivemos uma melhoria adicional no *throughput* para 333,622 transações por segundo e redução do tempo médio de resposta para 47,932 ms. Esta otimização combinada demonstra que, com o paralelismo ativo, valores mais elevados de `work_mem` beneficiam o sistema, revelando uma interação positiva entre paralelismo e memória de trabalho. O aumento de 4MB para 64MB no `work_mem`, quando combinado com as otimizações de paralelismo, resultou num ganho de desempenho de aproximadamente 11,1% no *throughput* em relação aos melhores resultados anteriores (300,3 txn/s).

### 3.2.5. fsync

Após alcançar melhorias através da configuração de paralelismo e aumento do `work_mem` para 64MB, prosseguimos com a desativação do `fsync`, parâmetro responsável por garantir a sincronização dos dados escritos em disco para durabilidade.

A modificação para `SET fsync = off` resultou em nova melhoria de desempenho, com *throughput* de 336,733 transações por segundo e tempo médio de resposta de 47,397 ms. Notavelmente, todas as operações de escrita apresentaram redução significativa nos tempos de resposta, com destaque para

`NewUser` (redução de 1,854ms para 1,008ms, uma melhoria de 45,6%) e `NewGame` (redução de 2,483ms para 1,448ms, melhoria de 41,7%).

A desativação do `fsync` proporcionou um ganho adicional de desempenho de aproximadamente 0,9% no *throughput* global comparado à configuração anterior. Vale realçar que esta configuração, embora benéfica para performance, compromete a durabilidade dos dados em caso de falha do sistema.

#### 3.2.6. wal_compression

Após desativar o `fsync`, prosseguimos com a desativação do `wal_compression`, parâmetro que comprime dados no `Write-Ahead Log`. Esta modificação resultou em um ligeiro decréscimo no desempenho, com o *throughput* reduzindo para 332,844 transações por segundo e o tempo médio de resposta aumentando para 48,152 ms. A maioria das operações manteve tempos de resposta similares aos observados com `fsync` desativado, com pequenas variações. No entanto, para alguns tipos de transações, como `GameReviews`, notou-se uma melhoria significativa, com redução de 11,849ms para 10,079ms.

#### 3.2.7. Resultados Obtidos

Tabela 12: Análise do *Throughput* após a alteração de parâmetros

| Métrica | Baseline (Apenas índices transacionais) | Alterar shared_buffers | Paralelismo | Desligar fsync |
|---|---|---|---|---|
| Throughput (txn/s) | 225.2ms | 300.3ms | 333.2ms | 336.7ms |

Partindo de um *throughput* inicial de 225,2ms com apenas índices transacionais, observamos um ganho acumulado significativo de aproximadamente 50% depois da introdução das otimizações dos parâmetros, culminando em 336,7ms após todas as otimizações.

O padrão de retornos decrescentes evidenciado (33,3% na primeira otimização, 11% na segunda e apenas 1,1% na terceira) revela claramente a hierarquia dos constrangimentos que afetavam o sistema. A memória constituía o principal gargalo, seguida do processamento e, por último, das operações de escrita em disco.

# 4. Conclusão e Trabalho Futuro

Este trabalho permitiu investigar de forma prática o impacto de estratégias de indexação e parametrização no desempenho de um sistema de gestão de bases de dados considerando tanto cargas transacionais como analíticas. Ao longo da análise, foram aplicados conhecimentos adquiridos na unidade curricular de Administração de Bases de Dados, com foco em como decisões técnicas podem influenciar diretamente a eficiência do sistema. Os testes com diferentes tipos de índices revelaram que os índices transacionais têm um papel determinante na melhoria do desempenho, destacando-se pelo aumento do *throughput* e pela redução significativa do tempo médio de resposta. Já os índices analíticos mostraram um contributo mais discreto neste contexto, sendo menos eficazes para as operações transacionais testadas. Complementarmente, procedeu-se à afinação de parâmetros do PostgreSQL, onde se verificou um aumento expressivo no desempenho após ajustes como o aumento da memória alocada (*shared_buffers*), ativação de paralelismo e desativação do fsync. Estes resultados demonstram não só a importância de uma infraestrutura bem configurada, como também a existência de limites naturais ao ganho incremental das otimizações. De forma geral, este trabalho proporcionou uma compreensão mais aprofundada sobre a gestão eficiente de bases de dados, demonstrando como intervenções técnicas, tanto ao nível lógico como físico, podem transformar o comportamento do sistema em cenários de elevada carga transacional e analítica.

## A Script de otimização

```
import os
import argparse
import subprocess
import logging
from dotenv import load_dotenv

# ========== LOGGING SETUP ==========
logging.basicConfig(
    level=logging.INFO,
    format='%(levelname)s - %(message)s'
)

# ========== LOAD ENV VARIABLES ==========
load_dotenv()

PG_USER = os.getenv("PG_USER")
PG_PASSWORD = os.getenv("PG_PASSWORD")
PG_DB = os.getenv("PG_DB")
PG_HOST = os.getenv("PG_HOST", "localhost")
PG_PORT = os.getenv("PG_PORT", "5432")

# ========== PATHS ==========
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
SCRIPTS_DIR = os.path.join(BASE_DIR, "scripts")


# ========== FUNCTIONS ==========
def run_sql_file(file_path):
    """
    Run a SQL script without saving the output to a file, just print results.
    """
    logging.info(f"Running SQL script: {file_path}")

    env = os.environ.copy()
    env["PGPASSWORD"] = PG_PASSWORD

    result = subprocess.run([
        "psql",
        "-U", PG_USER,
        "-d", PG_DB,
        "-h", PG_HOST,
        "-p", PG_PORT,
        "-f", file_path
    ], env=env, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

    if result.returncode != 0:
        logging.error(f"Error executing: {file_path}. Return code:
{result.returncode}")
    else:
        logging.info(f"Successfully executed: {file_path}")



def run_transactionals_optimizations(query=None):
    """
```

```
    Run optimization scripts for transactionals queries.
    If a query is specified, only run optimizations for that query.
    """
    transactionals_dir = os.path.join(SCRIPTS_DIR, "transactional")
    for subdir in sorted(os.listdir(transactionals_dir)):
        subdir_path = os.path.join(transactionals_dir, subdir)
        if os.path.isdir(subdir_path):
            optimization_script = os.path.join(subdir_path, f"{subdir}
_optimizations.sql")

            if os.path.exists(optimization_script):
                if query is None or query == subdir:
                    logging.info(f"Running optimizations for {subdir}. This action
may take a while...")
                    run_sql_file(optimization_script)
            else:
                logging.error(f"Optimization script not found:
{optimization_script}")



def run_analytics_optimizations(query=None):
    """
    Run optimization scripts for analytics queries.
    If a query is specified, only run optimizations for that query.
    """
    analytics_dir = os.path.join(SCRIPTS_DIR, "analytical")
    for q_folder in sorted(os.listdir(analytics_dir)):
        folder_path = os.path.join(analytics_dir, q_folder)
        if os.path.isdir(folder_path):
            if query is not None and query != q_folder:
                continue  # Skip if the query doesn't match the target query
            opt_file = os.path.join(folder_path, f"{q_folder}_optimizations.sql")
            if os.path.exists(opt_file):
                logging.info(f"Running optimizations for {q_folder}. This action may
take a while...")
                run_sql_file(opt_file)
            else:
                logging.error(f"Optimization script not found: {opt_file}")


# ========== MAIN ==========
def main():
    parser = argparse.ArgumentParser(description="Run SQL optimization scripts on
PostgreSQL.")

    # Arguments for running optimizations
    parser.add_argument("--optimize_analytics", action="store_true", help="Run
optimization scripts for all analytics queries.")
    parser.add_argument("--optimize_transactionals", action="store_true", help="Run
optimization scripts for all transactional queries.")

    # For individual query optimizations
    parser.add_argument("--optimize_Q1", action="store_true", help="Run optimization
for Q1 analytics query.")
    parser.add_argument("--optimize_Q2", action="store_true", help="Run optimization
```

```
for Q2 analytics query.")
    parser.add_argument("--optimize_Q3", action="store_true", help="Run optimization
for Q3 analytics query.")
    parser.add_argument("--optimize_Q4", action="store_true", help="Run optimization
for Q4 analytics query.")
    parser.add_argument("--optimize_Q5", action="store_true", help="Run optimization
for Q5 analytics query.")

    # Similar options for transactional queries
    parser.add_argument("--optimize_gameReviews", action="store_true", help="Run
optimization for gameReviews transactional query.")
    parser.add_argument("--optimize_recentGamesPerTag", action="store_true",
help="Run optimization for recentGamesPerTag transactional query.")
    parser.add_argument("--optimize_searchGames", action="store_true", help="Run
optimization for searchGames transactional query.")
    parser.add_argument("--optimize_userInfo", action="store_true", help="Run
optimization for userInfo transactional query.")

    # To run all optimizations (both analytics and transactionals)
    parser.add_argument("--all_optimizations", action="store_true", help="Run all
optimization scripts (analytics + transactionals).")

    args = parser.parse_args()

    no_flags = not any([
        args.optimize_analytics,
        args.optimize_transactionals,
        args.optimize_Q1,
        args.optimize_Q2,
        args.optimize_Q3,
        args.optimize_Q4,
        args.optimize_Q5,
        args.optimize_gameReviews,
        args.optimize_recentGamesPerTag,
        args.optimize_searchGames,
        args.optimize_userInfo,
        args.all_optimizations
    ])

    # If no flags are provided or --all_optimizations is used
    if args.all_optimizations or no_flags:
        logging.info("Running all optimizations (analytics + transactionals)...")
        run_analytics_optimizations()
        run_transactionals_optimizations()
    else:
        if args.optimize_analytics:
            logging.info("Running optimizations for all analytics queries...")
            run_analytics_optimizations()
        if args.optimize_transactionals:
            logging.info("Running optimizations for all transactional queries...")
            run_transactionals_optimizations()

        # Running individual query optimizations for analytics
        if args.optimize_Q1:
            run_analytics_optimizations(query="Q1")
        if args.optimize_Q2:
```

```
            run_analytics_optimizations(query="Q2")
        if args.optimize_Q3:
            run_analytics_optimizations(query="Q3")
        if args.optimize_Q4:
            run_analytics_optimizations(query="Q4")
        if args.optimize_Q5:
            run_analytics_optimizations(query="Q5")

        # Running individual query optimizations for transactionals
        if args.optimize_gameReviews:
            run_transactionals_optimizations(query="gameReviews")
        if args.optimize_recentGamesPerTag:
            run_transactionals_optimizations(query="recentGamesPerTag")
        if args.optimize_searchGames:
            run_transactionals_optimizations(query="searchGames")
        if args.optimize_userInfo:
            run_transactionals_optimizations(query="userInfo")


if __name__ == "__main__":
    main()
```

# B Script para *Tunning* de parâmetros

```
#!/usr/bin/env python3
import subprocess
import re
import psycopg2
import time
import os
import logging
from typing import Dict, List, Tuple, Any
from dotenv import load_dotenv

# Carrega as variáveis de ambiente do arquivo .env
load_dotenv(dotenv_path='../.env')

# Configuração de logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("postgres_tuning.log"),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)

# Configuração da conexão PostgreSQL usando variáveis de ambiente
DB_CONFIG = {
    "host": os.getenv("DB_HOST", "localhost"),
    "port": int(os.getenv("DB_PORT", 5432)),
    "database": os.getenv("DB_NAME", "adb"),
    "user": os.getenv("DB_USER", "postgres"),
    "password": os.getenv("DB_PASSWORD", "1234")
}
```

```
# Comando para executar benchmark
BENCHMARK_CMD = [
    "java", "-jar", os.getenv("BENCHMARK_CMD_JAR_PATH", "target/transactional-1.0-
SNAPSHOT.jar"),
    "-d", f"jdbc:postgresql://{DB_CONFIG['host']}:{DB_CONFIG['port']}/
{DB_CONFIG['database']}",
    "-U", DB_CONFIG["user"],
    "-P", DB_CONFIG["password"],
    "-W", "15",     # Período de warmup em segundos
    "-R", "180",    # Duração do teste em segundos
    "-c", "16"      # Número de clientes concorrentes
]

# Parâmetros a serem testados, em ordem
PARAMETERS_TO_TEST = [
    {
        "name": "shared_buffers",
        "values": ["128MB", "256MB", "512MB", "1GB", "2GB", "4GB"],
        "requires_restart": True
    },
    {
        "name": "work_mem",
        "values": ["4MB", "8MB", "16MB", "32MB", "64MB"],
        "requires_restart": False
    },
    {
        "name": "effective_cache_size",
        "values": ["1GB", "2GB", "4GB", "8GB"],
        "requires_restart": False
    },
    {
        "name": "maintenance_work_mem",
        "values": ["64MB", "128MB", "256MB", "512MB"],
        "requires_restart": False
    },
    {
        "name": "random_page_cost",
        "values": ["4.0", "3.0", "2.0", "1.5", "1.0"],
        "requires_restart": False
    },
    # Parâmetros de paralelismo
    {
        "name": "max_worker_processes",
        "values": ["4", "8", "16", "32"],
        "requires_restart": True
    },
    {
        "name": "max_parallel_workers",
        "values": ["4", "8", "16", "32"],
        "requires_restart": True
    },
    {
        "name": "max_parallel_workers_per_gather",
        "values": ["0", "2", "4", "8"],
        "requires_restart": False
    },
```

```
    {
        "name": "parallel_tuple_cost",
        "values": ["0.1", "0.2", "0.5", "1.0"],
        "requires_restart": False
    },
    {
        "name": "parallel_setup_cost",
        "values": ["1000", "1500", "2000"],
        "requires_restart": False
    },
    {
        "name": "min_parallel_table_scan_size",
        "values": ["4MB", "8MB", "16MB"],
        "requires_restart": False
    },
    {
        "name": "min_parallel_index_scan_size",
        "values": ["256kB", "512kB", "1MB", "2MB"],
        "requires_restart": False
    }
]

def execute_command(cmd: List[str]) -> str:
    """Executa um comando e retorna a saída."""
    try:
        logger.info(f"Executando: {' '.join(cmd)}")
        result = subprocess.run(cmd, capture_output=True, text=True, check=True)
        return result.stdout
    except subprocess.CalledProcessError as e:
        logger.error(f"Erro ao executar comando: {e}")
        logger.error(f"Saída de erro: {e.stderr}")
        raise

def parse_benchmark_output(output: str) -> float:
    """Extrai o throughput do resultado do benchmark."""
    match = re.search(r'throughput \(txn/s\) = ([\d.]+)', output)
    if match:
        return float(match.group(1))
    else:
        logger.error("Não foi possível encontrar o throughput no resultado")
        logger.debug(f"Saída completa: {output}")
        return 0.0

def get_postgresql_config_file() -> str:
    """Obtém o caminho do arquivo postgresql.conf."""
    try:
        with psycopg2.connect(**DB_CONFIG) as conn:
            with conn.cursor() as cur:
                cur.execute("SHOW config_file;")
                result = cur.fetchone()
                return result[0] if result else None
    except Exception as e:
        logger.error(f"Erro ao obter arquivo de configuração: {e}")
        raise

def update_postgresql_parameter(param_name, param_value):
```

```
    try:
        # Conectar ao banco de dados PostgreSQL
        conn = psycopg2.connect("dbname=adb user=postgres password=1234
host=localhost port=5432")
        cur = conn.cursor()

        # Desabilitar transação automática para garantir que o comando ALTER SYSTEM
não esteja em uma transação
        conn.set_session(autocommit=True)

        # Alterar o parâmetro
        cur.execute(f"ALTER SYSTEM SET {param_name} = '{param_value}';")
        print(f"Parâmetro {param_name} atualizado para {param_value}.")

        # Fechar o cursor e a conexão
        cur.close()
        conn.close()

    except Exception as e:
        print(f"Erro ao atualizar parâmetro {param_name}: {e}")

def restart_postgresql() -> None:
    """Reinicia o serviço PostgreSQL."""
    try:
        # Tenta primeiro com systemctl (Linux com systemd)
        logger.info("Reiniciando PostgreSQL")
        os.system("sudo systemctl restart postgresql")
        # Aguarda um tempo para o PostgreSQL voltar online
        time.sleep(10)

        # Verifica se está online
        test_connection()

    except Exception as e:
        logger.error(f"Erro ao reiniciar PostgreSQL: {e}")
        logger.info("Por favor, reinicie manualmente o PostgreSQL e pressione Enter
para continuar...")
        input()

def test_connection() -> bool:
    """Testa a conexão com o PostgreSQL."""
    max_attempts = 5
    attempt = 0

    while attempt < max_attempts:
        try:
            with psycopg2.connect(**DB_CONFIG) as conn:
                with conn.cursor() as cur:
                    cur.execute("SELECT 1")
                    return True
        except Exception as e:
            attempt += 1
            logger.warning(f"Tentativa {attempt} falhou: {e}")
            time.sleep(5)

    raise Exception("Não foi possível conectar ao PostgreSQL após várias tentativas")
```

```
def run_benchmark() -> Tuple[float, Dict[str, Any]]:
    """Executa o benchmark e retorna o throughput e métricas detalhadas."""
    # Registra configuração atual dos parâmetros mais relevantes
    current_config = {}
    try:
        with psycopg2.connect(**DB_CONFIG) as conn:
            with conn.cursor() as cur:
                for param in [p["name"] for p in PARAMETERS_TO_TEST]:
                    cur.execute(f"SHOW {param};")
                    current_config[param] = cur.fetchone()[0]
    except Exception as e:
        logger.warning(f"Não foi possível obter configuração atual: {e}")

    # Executa o benchmark
    start_time = time.time()
    output = execute_command(BENCHMARK_CMD)
    execution_time = time.time() - start_time
    throughput = parse_benchmark_output(output)

    # Extrai outras métricas potencialmente úteis
    response_time_match = re.search(r'response time \(ms\) = ([\d.]+)', output)
    abort_rate_match = re.search(r'abort rate \(%\) = ([\d.]+)', output)

    # Extrai métricas de funções individuais
    function_metrics = {}
    function_section = re.search(r'Response time per function \(ms\)(.*?)Overall
metrics', output, re.DOTALL)
    if function_section:
        functions_text = function_section.group(1)
        for line in functions_text.strip().split('\n'):
            if '=' in line:
                func_name, time_ms = line.split('=')
                function_metrics[func_name.strip()] = float(time_ms.strip())

    metrics = {
        "timestamp": time.strftime("%Y-%m-%d %H:%M:%S"),
        "throughput": throughput,
        "response_time": float(response_time_match.group(1)) if response_time_match
else None,
        "abort_rate": float(abort_rate_match.group(1)) if abort_rate_match else None,
        "execution_time": execution_time,
        "current_config": current_config,
        "function_metrics": function_metrics,
        "raw_output": output
    }

    logger.info(f"Benchmark concluído - Throughput: {throughput} txn/s")

    # Salva esta execução em um arquivo de histórico
    save_benchmark_run(metrics)

    return throughput, metrics

def save_benchmark_run(metrics: Dict[str, Any]) -> None:
    """Salva os resultados de uma execução do benchmark em um arquivo de
```

```
histórico."""
    os.makedirs("resultados_testes/historico", exist_ok=True)

    # Gera um nome de arquivo único com timestamp
    timestamp = time.strftime("%Y%m%d_%H%M%S")

    # Adiciona informação sobre configuração ao nome do arquivo
    config_summary = ""
    if metrics["current_config"]:
        # Pega o primeiro parâmetro configurado para identificar o teste
        for param in PARAMETERS_TO_TEST:
            if param["name"] in metrics["current_config"]:
                config_summary = f"{param['name']}_{metrics['current_config']
[param['name']]}"
                break

    filename = f"resultados_testes/historico/benchmark_{timestamp}
_{config_summary}.txt"

    with open(filename, "w") as f:
        f.write(f"Execução de Benchmark - {metrics['timestamp']}\n")
        f.write("="*50 + "\n\n")

        f.write("Métricas gerais:\n")
        f.write(f"- Throughput: {metrics['throughput']} txn/s\n")
        f.write(f"- Tempo de resposta: {metrics['response_time']} ms\n")
        f.write(f"- Taxa de aborto: {metrics['abort_rate']} %\n")
        f.write(f"- Tempo de execução: {metrics['execution_time']:.2f} segundos\n\n")

        f.write("Configuração atual:\n")
        for param, value in metrics["current_config"].items():
            f.write(f"- {param}: {value}\n")

        f.write("\nTempo de resposta por função (ms):\n")
        for func, time_ms in metrics["function_metrics"].items():
            f.write(f"- {func}: {time_ms} ms\n")

        f.write("\nSaída completa:\n")
        f.write("-"*50 + "\n")
        f.write(metrics["raw_output"])

def optimize_parameters() -> Dict[str, str]:
    """Função principal que testa e otimiza cada parâmetro."""
    optimized_params = {}

    logger.info("Iniciando processo de otimização de parâmetros PostgreSQL")

    # Testa cada parâmetro
    for param in PARAMETERS_TO_TEST:
        param_name = param["name"]
        values = param["values"]
        requires_restart = param["requires_restart"]

        logger.info(f"\n{'='*50}")
        logger.info(f"Testando parâmetro: {param_name}")
        logger.info(f"{'='*50}")
```

```
        best_value = None
        best_throughput = 0.0
        results = {}

        for value in values:
            logger.info(f"\nTestando {param_name} = {value}")
            update_postgresql_parameter(param_name, value)

            if requires_restart:
                restart_postgresql()

            # Executamos o benchmark
            throughput, metrics = run_benchmark()
            results[value] = metrics

            if throughput > best_throughput:
                best_throughput = throughput
                best_value = value

        # Definimos o melhor valor encontrado
        logger.info(f"\nMelhor valor para {param_name}: {best_value} (Throughput:
{best_throughput} txn/s)")
        update_postgresql_parameter(param_name, best_value)

        if requires_restart:
            restart_postgresql()

        optimized_params[param_name] = best_value

        # Salvamos os resultados detalhados deste parâmetro
        save_parameter_results(param_name, results, best_value)

    return optimized_params

def save_parameter_results(param_name: str, results: Dict[str, Dict], best_value:
str) -> None:
    """Salva os resultados de um parâmetro em um arquivo."""
    # Cria diretório para resultados se não existir
    os.makedirs("resultados_testes", exist_ok=True)

    with open(f"resultados_testes/tuning_results_{param_name}.txt", "w") as f:
        f.write(f"Resultados para {param_name}\n")
        f.write("="*50 + "\n\n")

        # Tabela de resultados
        f.write(f"{'Valor':<10} | {'Throughput (txn/s)':<20} | {'Tempo de resposta
(ms)':<25} | {'Taxa de aborto (%)':<20}\n")
        f.write("-"*80 + "\n")

        for value, metrics in results.items():
            mark = " *" if value == best_value else ""
            f.write(f"{value + mark:<10} | {metrics['throughput']:<20.3f} |
{metrics['response_time']:<25.3f} | {metrics['abort_rate']:<20.3f}\n")

        f.write("\n\n")
```

```
        f.write(f"Melhor valor: {best_value} (Throughput: {results[best_value]
['throughput']} txn/s)\n")

        # Saída completa para cada valor testado
        f.write("\nDetalhes de cada teste:\n")

        for value, metrics in results.items():
            f.write("\n" + "-"*50 + "\n")
            f.write(f"Teste com {param_name} = {value}\n")
            f.write("-"*50 + "\n")
            f.write(metrics['raw_output'])

    # Também salvamos um arquivo com apenas os resultados do melhor valor
    with open(f"resultados_testes/best_{param_name}_{best_value}.txt", "w") as f:
        f.write(f"Melhor valor para {param_name} = {best_value}\n")
        f.write("="*50 + "\n\n")
        f.write(f"Throughput: {results[best_value]['throughput']} txn/s\n")
        f.write(f"Tempo de resposta: {results[best_value]['response_time']} ms\n")
        f.write(f"Taxa de aborto: {results[best_value]['abort_rate']} %\n\n")
        f.write("Saída completa:\n")
        f.write("-"*50 + "\n")
        f.write(results[best_value]['raw_output'])

def save_final_results(optimized_params: Dict[str, str]) -> None:
    """Salva os resultados finais da otimização."""
    # Cria diretório para resultados se não existir
    os.makedirs("resultados_testes", exist_ok=True)

    # Salva em formato legível
    with open("resultados_testes/postgres_optimized_config.txt", "w") as f:
        f.write("Parâmetros otimizados do PostgreSQL\n")
        f.write("="*50 + "\n\n")

        for param, value in optimized_params.items():
            f.write(f"{param} = {value}\n")

    # Também salva em formato adequado para postgresql.conf
    with open("resultados_testes/postgresql_optimized.conf", "w") as f:
        f.write("# Configuração otimizada do PostgreSQL\n")
        f.write("# Gerado automaticamente pelo script de otimização\n")
        f.write(f"# Data: {time.strftime('%Y-%m-%d %H:%M:%S')}\n\n")

        for param, value in optimized_params.items():
            f.write(f"{param} = {value}\n")

def get_current_parameters() -> Dict[str, str]:
    """Obtém os valores atuais dos parâmetros do PostgreSQL."""
    current_params = {}
    try:
        with psycopg2.connect(**DB_CONFIG) as conn:
            with conn.cursor() as cur:
                for param in [p["name"] for p in PARAMETERS_TO_TEST]:
                    cur.execute(f"SHOW {param};")
                    current_params[param] = cur.fetchone()[0]
    except Exception as e:
        logger.error(f"Erro ao obter parâmetros atuais: {e}")
```

```
    return current_params

def create_report(optimized_params: Dict[str, str], initial_params: Dict[str, str],
                  initial_metrics: Dict[str, Any], final_metrics: Dict[str, Any]) ->
None:
    """Cria um relatório final com todos os resultados."""
    os.makedirs("resultados_testes", exist_ok=True)

    with open("resultados_testes/relatorio_final.md", "w") as f:
        f.write("# Relatório de Otimização do PostgreSQL\n\n")
        f.write(f"Data: {time.strftime('%Y-%m-%d %H:%M:%S')}\n\n")

        f.write("## Resumo\n\n")

        # Tabela comparativa de throughput
        f.write("### Comparação de Performance\n\n")
        f.write("| Métrica | Antes da Otimização | Após a Otimização | Melhoria |\n")
        f.write("|---------|---------------------|-------------------|----------|\n")

        initial_throughput = initial_metrics.get("throughput", 0)
        final_throughput = final_metrics.get("throughput", 0)

        if initial_throughput > 0:
            improvement = ((final_throughput - initial_throughput) /
initial_throughput) * 100
            f.write(f"| Throughput (txn/s) | {initial_throughput:.2f} |
{final_throughput:.2f} | {improvement:.2f}% |\n")
        else:
            f.write(f"| Throughput (txn/s) | N/A | {final_throughput:.2f} | N/A |\n")

        f.write("\n")

        # Parâmetros otimizados
        f.write("## Parâmetros Otimizados\n\n")
        f.write("| Parâmetro | Valor Original | Valor Otimizado |\n")
        f.write("|-----------|----------------|----------------|\n")

        for param, value in optimized_params.items():
            original = initial_params.get(param, "N/A")
            f.write(f"| {param} | {original} | {value} |\n")

        f.write("\n")

        # Mais detalhes sobre a performance final
        f.write("## Detalhes de Performance Final\n\n")
        f.write(f"- **Throughput**: {final_metrics.get('throughput', 'N/A')} txn/
s\n")
        f.write(f"- **Tempo de Resposta**: {final_metrics.get('response_time', 'N/
A')} ms\n")
        f.write(f"- **Taxa de Aborto**: {final_metrics.get('abort_rate', 'N/A')}%
\n\n")

        # Tempos de resposta por função
        if 'function_metrics' in final_metrics and final_metrics['function_metrics']:
            f.write("### Tempo de Resposta por Função (ms)\n\n")
```

```
            f.write("| Função | Tempo (ms) |\n")
            f.write("|--------|------------|\n")

            # Ordena por tempo de resposta (do maior para o menor)
            sorted_funcs = sorted(final_metrics['function_metrics'].items(),
                                  key=lambda x: x[1], reverse=True)

            for func, time_ms in sorted_funcs:
                f.write(f"| {func} | {time_ms} |\n")

        f.write("\n")

        # Instruções para aplicar configuração
        f.write("## Como Aplicar esta Configuração\n\n")
        f.write("Para aplicar esta configuração otimizada, você pode:\n\n")
        f.write("1. Copiar os parâmetros do arquivo `postgresql_optimized.conf` para
o seu arquivo `postgresql.conf`\n")
        f.write("2. Reiniciar o PostgreSQL para aplicar as alterações\n\n")

        f.write("Comando para reiniciar (Linux com systemd):\n")
        f.write("
```

bashnsudo systemctl restart postgresqln

```
\n\n")

def main():
    """Função principal."""
    try:
        logger.info("Iniciando script de otimização de parâmetros PostgreSQL")

        # Cria diretórios para resultados
        os.makedirs("resultados_testes", exist_ok=True)
        os.makedirs("resultados_testes/historico", exist_ok=True)

        # Verifica conexão com o banco
        test_connection()

        # Obtém parâmetros atuais
        initial_params = get_current_parameters()

        # Executa um benchmark com a configuração atual
        logger.info("Executando benchmark com a configuração atual...")
        initial_throughput, initial_metrics = run_benchmark()

        logger.info(f"Configuração inicial - Throughput: {initial_throughput} txn/s")
        logger.info(f"Salvando detalhes na pasta 'resultados_testes/historico'")

        # Executa a otimização
        optimized_params = optimize_parameters()

        # Salva resultados finais
        save_final_results(optimized_params)

        logger.info("\nOtimização concluída!")
        logger.info("Os parâmetros otimizados foram salvos em 'resultados_testes/
postgres_optimized_config.txt'")
```

```
        # Executa um benchmark final com todos os parâmetros otimizados
        logger.info("\nExecutando benchmark final com todos os parâmetros
otimizados...")
        final_throughput, final_metrics = run_benchmark()

        # Melhoria percentual
        if initial_throughput > 0:
            improvement = ((final_throughput - initial_throughput) /
initial_throughput) * 100
            logger.info(f"\nMelhoria de performance: {improvement:.2f}%")

        logger.info(f"\nResultado final - Throughput: {final_throughput} txn/s")
        logger.info(f"Tempo de resposta: {final_metrics['response_time']} ms")
        logger.info(f"Taxa de aborto: {final_metrics['abort_rate']} %")

        # Cria relatório final
        create_report(optimized_params, initial_params, initial_metrics,
final_metrics)
        logger.info("\nRelatório final gerado em 'resultados_testes/
relatorio_final.md'")

    except Exception as e:
        logger.error(f"Erro durante a execução: {e}")
        import traceback
        logger.error(traceback.format_exc())

if __name__ == "__main__":
    main()
```