Universidade de Brasília

Faculdade do Gama

Sistemas de Banco de Dados 2

Estudando *Tuning* em Banco de Dados Relacional

(Oportunidade Letiva de Discentes Experientes)

Na disciplina de Sistemas de Banco de Dados 2 (SBD2), oferecida no curso de Engenharia de Software da UnB, alguns estudantes mais experientes nesta disciplina (monitores) compartilharam seus estudos no tema de *Tuning* em Banco de Dados Relacionais nas Consultas SQL, lecionando parte de uma aula regular com a turma de SBD2 atual.

Nesta atividade foram explicados conceitos relevantes ao tema e demonstradas operações simples de serem realizadas pelo Workbench MySQL. Assim, está sendo apresentado abaixo como exemplo e modelo a consulta trabalhada em sala de aula envolvendo uma base de dados utilizada pela turma em exercícios solicitado pelo docente em atividade anterior (base disponível na Área de Compartilhamento - **/aulas/basesDados/projetoBaseDados_Jogos_2020.zip**).

Em seguida, são apresentadas três demandas solicitadas por seu cliente no qual você deverá elaborar uma consulta SQL que resolva cada problema demandado e depois apresentar o resultado da explicação fornecida pelo MySQL (*explain*) sobre sua consulta proposta inicialmente.

Após a explicação de sua solução (*Result Grid*) você deverá solicitar a análise do MySQL (*analyze*) para otimizar a sua proposta inicial, sendo ao final apresentado o resultado e a consulta que atenda a demanda otimizada.

Dessa forma, a apresentação realizada em sala de aula está detalhda a seguir com a indicação de primeira demanda (1) do cliente e você deverá elaborar as outras TRÊS propostas de soluções seguindo EXATAMENTE o mesmo padrão realizado para atender a primeira demanda.

1) Primeira Demanda

Elaborar uma consulta que recupere informações sobre os jogos que possuem vendas na América do Norte superiores a 1 milhão de unidades (note que todos os armazenamentos nesta base estão em unidade de milhão). A consulta deve retornar o nome do jogo, o gênero e as vendas nas regiões Norte-Americana, Europeia e Japonesa, ordenadas de forma decrescentes pela venda na América do Norte.

Proposta de solução inicial:

```
SELECT
    g.name AS game_name,
    ge.description AS genre_description,
    g.na_sales,
    g.eu_sales,
    g.jp_sales
FROM
    GAME g
JOIN
    GENRE ge ON g.id_genre = ge.id_genre
WHERE
    g.na_sales > 1
ORDER BY
    g.na_sales DESC;
```

Na consulta acima foi inicialmente definido os atributos que seriam projetados (apresentados) para o usuário que gerou a demanda como o resultado solicitado para a equipe de **TI** (ou de BD em específico). Em seguida, foi identificada a tabela principal para esta consulta (**GAME**). Uma junção (JOIN) seria necessária entre as tabelas **GAME** e **GENRE** para satisfazer os dados que seriam projetados por esta consulta, sendo implementada a condição para uma junção baseada no atributo **id_genre** presente nas duas tabelas (junção natural nesta consulta porque o projeto desta base de dados implementa esta restrição estabelecendo o relacionamento entre estas duas tabelas).

Na cláusula WHERE desta consulta foi elaborada a condição para a filtragem das vendas acima de um milhão, enquanto a ordenação final solicitada pelo

cliente foi atendida na cláusula ORDER BY desta consulta. Essa ordenação deveria ser decrescente sobre as vendas na América do Norte (**na_sales**).

Solicitando explicação sobre a solução inicial (*explain*):

```
EXPLAIN SELECT
        g.name AS game_name,
        ge.description AS genre_description,
        g.na_sales,
        g.eu_sales,
        g.jp_sales
FROM
        GAME g
JOIN
        GENRE ge ON g.id_genre = ge.id_genre
WHERE
        g.na_sales > 1
ORDER BY
        g.na_sales DESC;
```

Analisando os resultados apresentados pelo MySQL Workbench no *Result Grid* são destacados alguns itens relacionados a consulta proposta inicialmente.

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	ge	NULL	ALL	PRIMARY	NULL	NULL	NULL	12	100.00	Using temporary; Using filesort
1	SIMPLE	g	NULL	ref	GAME_GENRE_FK	GAME_GENRE_FK	5	t2_jogos.ge.id_genre	990	33.33	Using where

Interpretando o *Result Grid* do *EXPLAIN*:

Foi possível observar que na tabela **GENRE** (ge) a coluna ***type*** apresenta a expressão **ALL**, que indica que será realizada uma varredura completa na tabela (*full table scan*). Esse tipo de operação não é desejada por ser menos eficiente, especialmente em tabelas com grande volume de dados. Por outro lado, na

tabela **GAME** (g), o filtro é aplicado utilizando o índice disponível pela restrição implementada pela chave estrangeira GAME_GENRE_FK, o que torna o processo mais eficiente em comparação com a varredura completa.

Além disso, na coluna ***Extra*** do *Result Grid* são disponibilizadas informações adicionais sobre o processamento da consulta em cada tabela. Para a tabela GENRE o sistema está utilizando tabela temporária (*Using temporary*) que corresponde a tabela intermediária gerada durante o processamento da consulta, consumindo espaço nos recursos de memória do computador servidor do banco de dados. Na primeira indicação da ***Extra*** ainda está sendo informado que a realização da ordenação solicitada está sendo feita manualmente (expressão *Using filesort*), o que seria ineficiente em consultas envolvendo grandes volumes de dados. Já na tabela GAME a expressão *Using where* esclarecer que a condição da cláusula WHERE estará sendo aplicada como filtro desejado.

Solicitando a análise da solução inicial (*analyze*):

```
EXPLAIN ANALYZE SELECT
        g.name AS game_name,
        ge.description AS genre_description,
        g.na_sales,
        g.eu_sales,
        g.jp_sales
FROM
        GAME g
JOIN
        GENRE ge ON g.id_genre = ge.id_genre
WHERE
        g.na_sales > 1
ORDER BY
        g.na_sales DESC;
```

Interpretando o resultado do *ANALYZE*:

Após a execução da análise sob a consulta inicial o resultado produzido está sendo apresentado a seguir, a fim de abordar os principais esclarecimentos relacionados a primeira análise realizada em uma consulta SQL.

-> Sort: g.na_sales DESC (actual time=60.2..61.2 rows=757 loops=1)

-> Stream results (**cost=2063** rows=3961) (actual time=0.599..58.7 rows=757 loops=1)

-> Nested loop inner join (**cost=2063** rows=3961) (actual time=0.589..56.3 rows=757 loops=1)

-> Table scan on ge (**cost=1.45** rows=12) (actual time=0.0354..0.0707 rows=12 loops=1)

-> Filter: (g.na_sales > 1.00) (**cost=75.5** rows=330) (actual time=0.289..4.48 rows=63.1 loops=12)

-> Index lookup on g using GAME_GENRE_FK (id_genre=ge.id_genre) (**cost=75.5** rows=990) (actual time=0.188..2.97 rows=1004 loops=12)

É importante averiguar o resultado de baixo para cima para acompanhar a sequência do processamento e os seus resultados de apurações intermediária. Assim, a primeira linha do resultado das análises corresponde as apurações finais.

O resultado apresentado acima corresponde ao plano de execução que segue uma sequência de operações realizadas pelo MySQL para atender a consulta proposta inicialmente. As operações mais abaixo nesta sequência são executadas primeiro, e seus resultados são usados em operações mais altas (ou acima nesta sequência).

É interessante também acompanhar os custos (*cost*) que representam o consumo de recursos previstos pelo otimizador do SGBD (Sistema Gerenciador de Banco de Dados) que são necessários para executar cada operação importante para a execução completa da consulta SQL.

No plano de execução é indicado que o índice proveniente da GAME_GENRE_FK será utilizado apenas para a junção entre **GAME** e **GENRE**, enquanto o filtro da cláusula WHERE (g.**na_sales** > 1) fará que o otimizador precise efetuar um processamento extra para examinar as linhas correspondentes para a junção e aplicar o filtro apenas depois dessa junção. Além disso, a tabela **GENRE** realiza um *full scan* devido à ausência de um índice que permita a busca direta pelos valores necessários durante a junção com a tabela **GAME**. Isso força o otimizador a procurar em todas as tuplas da tabela para atender a condição da junção (**g.id_genre** = **ge.id_genre**).

Diante dessas informações do plano de execução será criado um índice adicional para tabela **GAME** para o atributo **na_sales** utilizado na cláusula WHERE e no ORDER BY da consulta.

CREATE INDEX na_sales_IDX ON GAME (na_sales);

Depois da criação deste novo objeto no SGBD MySQL está sendo executada a solicitação de explicação novamente na mesma consulta (execute novamente o *EXPLAIN* somente, conforme demonstrado anteriormente).

Apresentando o *Result Grid* da nova execução do *EXPLAIN*:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	ge	NULL	eq_ref	PRIMARY	PRIMARY	4	NULL	1	100.00	NULL
1	SIMPLE	g	NULL	range	GAME_GENRE_FK, na_sales_idx	na_sales_idx	3	NULL	757	100.00	Using index condition; Using where; Backward index scan

Apresentando o resultado da nova execução do *ANALYZE*:

-> Nested loop inner join (**cost=606** rows=757) (actual time=0.54..21.5 rows=757 loops=1)

-> Filter: (g.id_genre is not null) (**cost=341** rows=757) (actual time=0.512..7.96 rows=757 loops=1)

-> Index range scan on g using idxNa_sales over (1.00 < na_sales) (reverse), with index condition: (g.na_sales > 1.00) (**cost=341** rows=757) (actual time=0.502..4.95 rows=757 loops=1)

-> Single-row index lookup on ge using PRIMARY (id_genre=g.id_genre) (**cost=0.25** rows=1) (actual time=0.0063..0.00821 rows=1 loops=757)

Após a criação do índice na_sales_idx na tabela GAME, foram obtidas melhores nos custos (cost) do plano de execução da consulta. Esse índice adicional permitiu filtrar diretamente as tuplas da tabela GAME que atendiam a condição da cláusula WHERE, eliminando o custo do table scan na tabela GENRE , além de reduzir o número de tuplas processadas nas etapas subsequentes.

O custo da junção (*Nested loop inner join)* diminui de 2063 para 606 e o tempo real reduziu de 56,3ms para 21,5ms. Isso ocorreu principalmente porque a quantidade de tuplas da GAME que foram recuperadas pela cláusula WHERE eram menores (apenas as que atendiam a condição de na_sales > 1), estabelecendo um número de combinações a serem avaliadas na junção menores que a situação anterior. Assim, o otimizador trabalha com uma quantidade de dados menor, o que reduz o tempo total para se processar a junção.

No novo plano na etapa *Single-row index lookup on ge using* PRIMARY é possível notar que agora a junção realiza uma busca indexada na tabela **GENRE**, para cada tupla da tabela **GAME**. Isso é possível porque a filtragem eficiente em **GAME** reduz o número de buscas necessárias, tornando mais eficiente sua execução.

Dessa forma, a consulta inicialmente proposta foi mantida em sua escrita, mas um novo objeto no SGBD foi criado para tornar o desempenho dessa consulta SQL mais eficiente.

2) Segunda Demanda

Liste o nome dos jogos e o nome de seus respectivos publicadores, para os jogos que possuem a pontuação de usuário (*user_score*) igual a 9 (nove).

Proposta de solução inicial (elabore a consulta que resolva esta demanda):

```
SELECT
   g.name, p.publisher_name
FROM
   GAME g
      JOIN
   PUBLISHER p ON p.id_publisher = g.id_publisher
WHERE
   g.user_score = 9;
```

Solicitando explicação sobre a solução inicial (*explain*):

```
EXPLAIN SELECT
   g.name, p.publisher_name
FROM
   GAME g
      JOIN
   PUBLISHER p ON p.id_publisher = g.id_publisher
WHERE
   g.user_score = 9;
```

Apresentando a tabela e interpretando o *Result Grid* do *EXPLAIN*:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	g	NULL	ALL	GAME_PUBLISHER_FK	NULL	NULL	NULL	11801	10.00	Using where
1	SIMPLE	p	NULL	eq_ref	PRIMARY	PRIMARY	4	metacritic.g.id_publisher	1	100.00	NULL

Na análise do *EXPLAIN*, foi identificado que na tabela GAME (g) a consulta realiza uma varredura completa na tabela, conforme indicado pelo valor **ALL** na coluna

type. Isso ocorre porque não há um índice sendo utilizado na coluna user_score, o que torna o processo menos eficiente, já que todos os registros precisam ser analisados para aplicar o filtro da cláusula WHERE.

Na tabela PUBLISHER (p) o tipo de acesso indicado como **eq_ref**, demonstra que está sendo utilizada a chave primária dessa tabela devido à restrição de integridade referencial estabelecida entre as tabelas GAME e PUBLISHER, permitindo que o índice da chave primária de PUBLISHER seja usado para otimizar a junção.

Além disso, na coluna **Extra** é indicado que na tabela GAME é aplicada à cláusula WHRE para filtrar os dados, o que pode reforçar a necessidade de um índice na coluna user_score para evitar o full table scan.

Solicitando a análise da solução inicial (*analyze*):

```
EXPLAIN ANALYZE SELECT
      g.name, p.publisher_name
FROM
   GAME g
      JOIN
   PUBLISHER p ON p.id_publisher = g.id_publisher
WHERE
   g.user_score = 9;
```

Apresentando e interpretando o resultado do *ANALYZE*:

-> Nested loop inner join (cost=1794 rows=1180) (actual time=0.358..14.6 rows=85 loops=1)

-> Filter: (g.user_score = 9) (cost=1204 rows=1180) (actual time=0.34..14.2 rows=85 loops=1)

-> Table scan on g (cost=1204 rows=11801) (actual time=0.302..11.3 rows=12043 loops=1)

-> Single-row index lookup on p using PRIMARY (id_publisher=g.id_publisher) (cost=0.4 rows=1) (actual time=0.00427..0.00429 rows=1 loops=85)

No nível mais baixo do plano, observa-se que a operação inicial é um **table scan** na tabela g (GAME). Evidenciando o custo elevado (**cost=1204**, **rows=11801**) e pelo número total de linhas examinadas (**actual rows=12043**). Isso ocorre porque a coluna user_score utilizada no filtro da cláusula WHERE não possui um índice.

Ainda na tabela g, o filtro **g.user_score = 9** é aplicado após a leitura dos dados. Isso reduz o número de linhas a 85, porém, esse filtro ocorre de forma ineficiente devido à ausência de um índice.

Na sequência, o plano utiliza um **nested loop inner join** para combinar os resultados filtrados da tabela g com os dados da tabela p (PUBLISHER). Cada linha selecionada da tabela g é utilizada para buscar o correspondente na tabela p. Essa busca é eficiente, pois utiliza um **single-row index lookup** baseado na chave primária da tabela PUBLISHER (**id_publisher=g.id_publisher**).

Por fim a junção é concluída com um total de 85 linhas retornadas, conforme indicado em **actual rows=85**. O custo geral da junção (**cost=1794**) reflete o impacto da varredura completa na tabela g.

Explicar os ajustes realizados para melhoria da consulta inicial da tarefa 2:

CREATE INDEX user_score_IDX ON GAME(user_score);

O índice foi criado para melhorar o desempenho da consulta, especificamente na parte do filtro, onde estamos selecionando os registros em que a pontuação do usuário (user_score) é igual a 9. Quando há um índice na coluna user_score, o banco de dados pode usar esse índice para localizar rapidamente os registros que atendem à condição, ao invés de realizar uma varredura completa.

Apresentando o *Result Grid* da nova execução do *EXPLAIN*:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	g	NULL	ALL	GAME_PUBLISHER_FK,user_score_IDX	NULL	NULL	NULL	12168	10	Using where
1	SIMPLE	p	NULL	eq_ref	PRIMARY	PRIMARY	4	metacritic.g.id_publisher	1	100	NULL

Apresentando o resultado da nova execução do *ANALYZE*:

-> Nested loop inner join (cost=1617 rows=1180) (actual time=0.355..14.7 rows=85 loops=1)

-> Filter: (g.user_score = 9) (cost=1204 rows=1180) (actual time=0.336..14.1 rows=85 loops=1)

-> Table scan on g (cost=1204 rows=11801) (actual time=0.3..11.2 rows=12043 loops=1)

-> Single-row index lookup on p using PRIMARY (id_publisher=g.id_publisher) (cost=0.25 rows=1) (actual time=0.00558..0.00561 rows=1 loops=85)

Apresentação da análise final a partir da consulta inicial proposta e quais são os principais indicativos de que a consulta ficou mais eficiente:

Foi observado no plano de execução, que a consulta ainda realizou uma varredura completa na tabela GAME. Isso sugere que o índice não foi eficaz, provavelmente porque a seletividade de user_score = 9 não era suficientemente alta ou porque o banco de dados optou por realizar a varredura completa.

3) Terceira Demanda

Liste o nome dos jogos, o ano de lançamento e a descrição do gênero, considerando apenas os jogos que possuem um gênero associado, que foram lançados entre os anos de 2001 e 2012 e cujo nome começa com a letra "M".

Proposta de solução inicial (elabore a consulta que resolva esta nova demanda):

```
SELECT
   g.name,
   g.year_of_release,
   ge.description
FROM
   GAME g
JOIN
   GENRE ge
ON
   g.id_genre = ge.id_genre
WHERE
   g.year_of_release BETWEEN 2001 AND 2012
   AND g.name LIKE 'M%';
```

Solicitando explicação sobre a solução inicial (*explain*):

```
EXPLAIN SELECT
   g.name,
   g.year_of_release,
   ge.description
FROM
   GAME g
JOIN
   GENRE ge
ON
   g.id_genre = ge.id_genre
WHERE
   g.year_of_release BETWEEN 2001 AND 2012
   AND g.name LIKE 'M%';
```

Apresentando a tabela e interpretando o *Result Grid* do *EXPLAIN*:

id	select_type	table	partitions	type	possible_keys	key	key_le	ref	rows	filtered	Extra
1	SIMPLE	g	NULL	ALL	GAME_GENRE_FK	NULL	NULL	NULL	12168	1.23	Using where
1	SIMPLE	ge	NULL	eq_ref	PRIMARY	PRIMARY	4	metacritic.g.id_genre	1	100.00	NULL

A consulta processa os dados da tabela GAME (g) utilizando um *table scan* completo, identificado pelo tipo ALL na coluna type. Isso indica que o banco de dados lê todas as linhas da tabela.

Na tabela GENRE (ge), o acesso é mais eficiente, utilizando a chave primária (PRIMARY) como índice, conforme mostrado na coluna key. O tipo de junção eq_ref indica que o banco realiza buscas específicas para cada linha da tabela GAME, utilizando o valor de id_genre para a junção.

A coluna Extra exibe Using where, o que significa que o banco aplica os filtros especificados na cláusula WHERE após realizar o escaneamento completo na tabela GAME.

Solicitando a análise da solução inicial (*analyze*):

```
EXPLAIN ANALYZE SELECT
   g.name,
   g.year_of_release,
   ge.description
FROM
   GAME g
JOIN
   GENRE ge
ON
   g.id_genre = ge.id_genre
WHERE
   g.year_of_release BETWEEN 2001 AND 2012
   AND g.name LIKE 'M%';
```

Apresentando e interpretando o resultado do *ANALYZE*:

-> Nested loop inner join (cost=1294 rows=150) (actual time=6.26..16.9 rows=817 loops=1)

-> Filter: ((g.year_of_release between 2001 and 2012) and (g.`name` like "M%") and (g.id_genre is not null)) (cost=1241 rows=150) (actual time=6.22..15.9 rows=817 loops=1)

-> Table scan on g (cost=1241 rows=12168) (actual time=0.513..12.8 rows=12043 loops=1)

-> Single-row index lookup on ge using PRIMARY (id_genre=g.id_genre) (cost=0.251 rows=1) (actual time=0.00104..0.00107 rows=1 loops=817)

O plano de execução mostra que o banco de dados utilizou um *nested loop join* para realizar a junção entre as tabelas GAME (g) e GENRE (ge). Ele começou com um escaneamento completo na tabela GAME, porque não encontrou um índice eficiente para os filtros aplicados. Após aplicar os critérios, foram selecionadas 817 linhas. Para cada uma dessas linhas, foi feita uma busca na tabela GENRE com base na chave primária (id_genre).

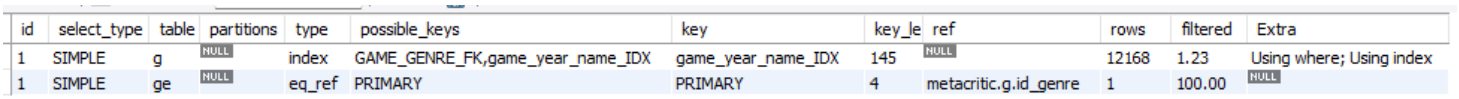
O principal gargalo está no escaneamento completo da tabela GAME.

Explicar os ajustes realizados para melhoria da consulta inicial da tarefa 3:

CREATE INDEX game_year_name_IDX ON GAME (year_of_release, name, id_genre);

O índice game_year_name_IDX foi criado nas colunas year_of_release, name e id_genre para otimizar a execução da consulta, pois essas colunas são essenciais para os filtros e a junção realizados.

Apresentando o *Result Grid* da nova execução do *EXPLAIN*:



id	select_type	table	partitions	type	possible_keys	key	key_le	ref	rows	filtered	Extra
1	SIMPLE	g	NULL	index	GAME_GENRE_FK,game_year_name_IDX	game_year_name_IDX	145	NULL	12168	1.23	Using where; Using index
1	SIMPLE	ge	NULL	eq_ref	PRIMARY	PRIMARY	4	metacritic.g.id_genre	1	100.00	NULL

Apresentando o resultado da nova execução do *ANALYZE*:

-> Nested loop inner join (cost=1294 rows=150) (actual time=1.42..15.9 rows=817 loops=1)

-> Filter: ((g.year_of_release between 2001 and 2012) and (g.`name` like "M%") and (g.id_genre is not null)) (cost=1241 rows=150) (actual time=1.4..12.1 rows=817 loops=1)

-> Covering index scan on g using game_year_name_IDX (cost=1241 rows=12168) (actual time=0.322..8.96 rows=12043 loops=1)

-> Single-row index lookup on ge using PRIMARY (id_genre=g.id_genre) (cost=0.251 rows=1) (actual time=0.0042..0.00423 rows=1 loops=817)

Apresentação da análise final a partir da consulta inicial proposta e quais são os principais indicativos de que a consulta ficou mais eficiente:

A análise da consulta revisada demonstra que a criação do índice foi bem-sucedida em otimizar o desempenho. Antes da implementação do índice, o banco de dados realizava uma busca completa na tabela GAME. Após a criação do índice, houve uma mudança significativa na forma como os dados foram acessados, agora, há o uso de um covering index scan, ao invés de um table scan.

Como podemos observar pelo resultado do da execução do analyze, o tempo foi consideravelmente menor.

4) Quarta Demanda

Listar o máximo do valor de venda de um jogo no Japão por categoria em que a nota dos críticos (*critic_score*) seja maior que 9 e tenha mais de 50 avaliações de críticos (*critic_count*).

Proposta de solução inicial (elabore a consulta que resolva esta nova demanda):

```
SELECT
   g.description,
   MAX(game.jp_sales) AS max_jp_sales
FROM
   game
JOIN
   genre g ON game.id_genre = g.id_genre
WHERE
   game.critic_score > 9
   AND game.critic_count > 50
GROUP BY
   g.description;
```

Solicitando explicação sobre a solução inicial (*explain*):

```
EXPLAIN SELECT
   g.description,
   MAX(game.jp_sales) AS max_jp_sales
FROM
   game
JOIN
   genre g ON game.id_genre = g.id_genre
WHERE
   game.critic_score > 9
   AND game.critic_count > 50
GROUP BY
   g.description;
```

Apresentando a tabela e interpretando o *Result Grid* do *EXPLAIN*:

id	select_type	table	partitions	type	possible_keys	key	key_le	ref	rows	filtered	Extra
1	SIMPLE	game	NULL	ALL	GAME_GENRE_FK	NULL	NULL	NULL	12168	11.11	Using where; Using temporary
1	SIMPLE	g	NULL	eq_ref	PRIMARY	PRIMARY	4	metacritic.game.id_genre	1	100.00	NULL

A consulta processa os dados da tabela GAME (g) utilizando um table scan completo, identificado pelo tipo ALL na coluna type.

Na tabela GENRE (ge), o acesso é mais eficiente, utilizando a chave primária (PRIMARY) como índice, conforme mostrado na coluna key.

O tipo de junção eq_ref indica que o banco realiza buscas específicas para cada linha da tabela GAME, utilizando o valor de id_genre para realizar a junção.

A coluna Extra exibe Using where, o que significa que o banco aplica os filtros especificados na cláusula WHERE após a leitura das linhas na tabela GAME. Além disso, o Using temporary indica que o MySQL cria uma tabela temporária para armazenar os resultados intermediários.

Solicitando a análise da solução inicial (*analyze*):

```
EXPLAIN ANALYZE SELECT
   g.description,
   MAX(game.jp_sales) AS max_jp_sales
FROM
   game
JOIN
   genre g ON game.id_genre = g.id_genre
WHERE
   game.critic_score > 9
   AND game.critic_count > 50
GROUP BY
   g.description;
```

Apresentando e interpretando o resultado do *ANALYZE*:

-> Table scan on <temporary> (actual time=18.2..18.3 rows=12 loops=1)

-> Aggregate using temporary table (actual time=18.2..18.2 rows=12 loops=1)

-> Nested loop inner join (cost=1714 rows=1352) (actual time=0.373..16.5 rows=830 loops=1)

-> Filter: ((game.critic_score > 9) and (game.critic_count > 50) and (game.id_genre is not null)) (cost=1241 rows=1352) (actual time=0.353..14.5 rows=830 loops=1)

-> Table scan on game (cost=1241 rows=12168) (actual time=0.343..13.1 rows=12043 loops=1)

-> Single-row index lookup on g using PRIMARY (id_genre=game.id_genre) (cost=0.25 rows=1) (actual time=0.00206..0.00212 rows=1 loops=830)

O plano de execução mostra que o banco de dados utilizou um nested loop join para realizar a junção entre as tabelas GAME (g) e GENRE (ge). O processo começou com um escaneamento completo da tabela GAME, identificado como um table scan, já que o banco não encontrou um índice eficiente para os filtros aplicados.

Após aplicar o filtro, o banco de dados executou uma junção, onde, para cada uma das 830 linhas selecionadas na tabela GAME, foi realizada uma busca na tabela GENRE usando um índice primário, com a chave id_genre como referência.

O principal gargalo da consulta está no escaneamento completo da tabela GAME.

Explicar os ajustes realizados para melhoria da consulta inicial da tarefa 4:

CREATE INDEX game_critic_count_critic_score_id_genre_IDX ON game (critic_count, critic_score, id_genre);

A criação do índice na tabela game tem como objetivo otimizar a consulta, especificamente as operações de filtragem e junção. Esse índice é composto pelas colunas critic_count, critic_score e id_genre, que são as mais utilizadas nas condições da cláusula WHERE e na junção com a tabela GENRE.

Apresentando o *Result Grid* da nova execução do *EXPLAIN*:

id	select_type	table	partitions	type	possible_keys	key	key_le	ref	rows	filtered	Extra
1	SIMPLE	game	NULL	range	GAME_GENRE_F...	game_critic_c...	5	NULL	830	33.33	Using index condition; Using temporary
1	SIMPLE	g	NULL	eq_ref	PRIMARY	PRIMARY	4	metacritic.game.id_genre	1	100.00	NULL

Apresentando o resultado da nova execução do *ANALYZE*:

-> Table scan on <temporary> (actual time=7.59..7.59 rows=12 loops=1)

-> Aggregate using temporary table (actual time=7.59..7.59 rows=12 loops=1)

-> Nested loop inner join (cost=526 rows=830) (actual time=0.384..5.88 rows=830 loops=1)

-> Index range scan on game using game_critic_count_critic_score_id_genre_IDX over (50 < critic_count), with index condition: (((game.critic_score > 9) and (game.critic_count > 50)) and (game.id_genre is not null)) (cost=374 rows=830) (actual time=0.369..3.47 rows=830 loops=1)

-> Single-row index lookup on g using PRIMARY (id_genre=game.id_genre) (cost=0.25 rows=1) (actual time=0.00253..0.00259 rows=1 loops=830)

Apresentação da análise final a partir da consulta inicial proposta e quais são os principais indicativos de que a consulta ficou mais eficiente:

A análise mostra uma melhoria significativa após a criação do índice game_critic_count_critic_score_id_genre_IDX. No primeiro plano de execução, a consulta realiza um table scan completo na tabela game, o que demora bastante. Já no segundo plano, o banco usa o índice para aplicar os filtros mais rapidamente, reduzindo o tempo de execução.