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 g.id_publisher = p.id_publisher
WHERE
        g.user_score >= '9.0'
ORDER BY
        g.user_score DESC;
```

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

```
EXPLAIN SELECT
        g.name,
        p.publisher_name
FROM
        GAME g
JOIN
        PUBLISHER p ON g.id_publisher = p.id_publisher
WHERE
        g.user_score >= '9.0'
ORDER BY
        g.user_score DESC;
```

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	11719	33.33	Using where
1	SIMPLE	p	NULL	Eq_ref	PRIMARY	PRIMARY	4	Metacritic.g.id	1	100.00	NULL

								_publis her			

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

```
EXPLAIN ANALYZE SELECT
        g.name,
        p.publisher_name
FROM
        GAME g
JOIN
        PUBLISHER p ON g.id_publisher = p.id_publisher
WHERE
        g.user_score >= ‘9.0’
ORDER BY
        g.user_score DESC;
```

Apresentando e interpretando o resultado do *ANALYZE*:

-> Nested loop inner join (cost=2563 rows=3906) (actual time=2.23..13.8 rows=1892 loops=1)

-> Filter: (g.user_score >= '9.0') (cost=1196 rows=3906) (actual time=0.947..10.8 rows=1892 loops=1)

-> Table scan on g (cost=1196 rows=11719) (actual time=0.933..10.3 rows=12043 loops=1)

-> Single-row index lookup on p using PRIMARY (id_publisher=g.id_publisher) (cost=0.25 rows=1) (actual time=0.00144..0.00147 rows=1 loops=1892)

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

Lendo o plano de execução de baixo para cima, podemos perceber que o maior problema nesta consulta está sendo em relação ao “*table scan on g*” que significa que está sendo preciso varrer toda a tabela GAME para encontrar as linhas que satisfazem a condição, no qual torna a consulta ineficiente. Um pouco mais acima temos o filtro que é utilizado (g.user_score >= ‘9.0’) na estratégia table

scan, e por falta da existência de um índice na coluna "*user_score*" está causando uma lentidão na consulta.

Portanto, é necessário criar um índice para a coluna **user_score** dentro da tabela GAME que é utilizada na cláusula WHERE e no ORDER BY na consulta. Vamos fazer da seguinte forma:

CREATE INDEX game_user_score_IDX ON GAME (user_score);

Após a criação do índice, iremos rodar novamente a consulta utilizando o EXPLAIN para obtermos o novo resultado do plano de execução.

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	range	GAME_PUBLISHER_FK, GAME_USER_SCORE_IDX	GAME_USER_SCORE_IDX	6	NULL	1892	100.00	Using index condition
1	SIMPLE	p	NULL	Eq_ref	PRIMARY	PRIMARY	4	Metacritic.g.id_publisher	1	100.00	NULL

Apresentando o resultado da nova execução do *ANALYZE*:

-> Nested loop inner join (cost=1514 rows=1892) (actual time=2.62..15.6 rows=1892 loops=1)

-> Index range scan on g using GAME_USER_SCORE_IDX over ('9.0' <= user_score), with index condition: (g.user_score >= '9.0') (cost=852 rows=1892) (actual time=2.59..11.9 rows=1892 loops=1)

-> Single-row index lookup on p using PRIMARY (id_publisher=g.id_publisher) (cost=0.25 rows=1) (actual time=0.00158..0.00163 rows=1 loops=1892)

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

De acordo com o plano de execução atual gerado após a criação do índice "**game_user_score_IDX**" é possível ver as melhorias significativas:

A substituição da *table scan* pelo *index range scan* mostrando que agora é utilizado o índice na consulta do qual vai limitar as linhas que irão atender a condição "user_score >= '9.0'". Dessa forma irá reduzir a quantidade de dados que serão lidos e processados.

Outra melhoria notável é o custo de acesso na tabela GAME que caiu de 1196 para 852, que reflete em relação a eficiência do índice na consulta. O custo total da consulta decaiu também de 2563 para 1514.

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%') AND (g.id_genre IS NOT NULL);
```

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%') AND (g.id_genre IS NOT NULL);
```

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_GENRE_FK	NULL	NULL	NULL	11719	1.23	Using Where; Using flesort

1	SIMPLE	ge	NULL	Eq_ref	PRIMARY	PRIMARY	4	Metacritic.g.id_genre	1	100.00	NULL

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%') AND (g.id_genre IS NOT NULL);
```

Apresentando e interpretando o resultado do *ANALYZE*:

-> Nested loop inner join (cost=1247 rows=145) (actual time=6.77..14.8 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=1196 rows=145) (actual time=6.74..13.8 rows=817 loops=1)

-> Table scan on g (cost=1196 rows=11719) (actual time=0.475..9.91 rows=12043 loops=1)

-> Single-row index lookup on ge using PRIMARY (id_genre=g.id_genre) (cost=0.251 rows=1) (actual time=822e-6..871e-6 rows=1 loops=817)

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

De acordo com o plano de execução podemos perceber que o filtro que está sendo aplicado dentro da cláusula WHERE "*g.year_of_release BETWEEN 2001 AND 2012) AND (g.name LIKE 'M%') AND (g.id_genre IS NOT NULL)*" está sendo usado após um *table scan* na tabela **GAME**, ou seja, o filtro está sendo aplicado após ter feito uma varredura em todas as linhas da tabelas que no qual torna a consulta ineficiente. Um outro detalhe nisso é que não há índices criados

e adequados na tabela **GAME** para acelerar a consulta e o filtro baseado nas colunas year_of_release e name.

Em virtude do que vimos acima, a forma que possamos concertar será criando um índice composto entre as colunas year_of_release e name.

CREATE INDEX name_year_of_release_IDX ON GAME (name, year_of_release);

Após isso, temos que também mudar um pouco a nossa query de consulta para que possamos reduzir ainda mais o número de linhas retornadas. Portanto, a consulta vai ficar dessa forma aqui:

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

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	GAME	NULL	range	GAME_GENRE_FK, name_year_of_release_IDX	Name_year_if_release_IDX	134	NULL	1040	11.11	Using Index condition; Using where
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=509 rows=116) (actual time=0.25..1.82 rows=817 loops=1)

-> Filter: (game.id_genre is not null) (cost=468 rows=116) (actual time=0.236..1.42 rows=817 loops=1)

-> Index range scan on GAME using name_year_of_release_IDX over ('M\0' <= name <= 'M???'), with index condition: ((game.year_of_release between 2001 and 2012) and (game.`name` like 'M%')) (cost=468 rows=1040) (actual time=0.234..1.37 rows=817 loops=1)

-> Single-row index lookup on ge using PRIMARY (id_genre=game.id_genre) (cost=0.251 rows=1) (actual time=347e-6..371e-6 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:

Com as melhorias feitas na consulta, o novo plano de execução demonstrou que as otimizações foram bem-sucedida. Um exemplo que demonstra muito bem foi a substituição do “*table scan*” para o *index range* na tabela **GAME**, mostrando que está sendo utilizada o índice composto **name_year_of_release_IDX**. Além disso, o índice também é muito bem usando nos filtros “name LIKE ‘M%’” e “year_of_release BETWEEN 2001 AND 2012”.

Outro detalhe interessante é o uso do índice durante o filtro aplicado, assim minimizando o número de linhas lidas, visto que o custo da consulta original estava como 1196 e caiu para 468, mostrando que menos dados estão sendo processados.

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
    name,
    platform,
    MAX(jp_sales) AS max_sales
FROM
    GAME
WHERE
    (critic_score > 9) AND (critic_count > 50)
GROUP BY
    name,
    platform;
```

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

```
EXPLAIN SELECT
    name,
    platform,
    MAX(jp_sales) AS max_sales
FROM
    GAME
WHERE
    (critic_score > 9) AND (critic_count > 50)
GROUP BY
    name,
    platform;
```

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	GAME	NULL	ALL	NULL	NULL	NULL	NULL	11719	11.11	Using Where; Using temporary

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

EXPLAIN ANALYZE SELECT

Apresentando e interpretando o resultado do *ANALYZE*:

-> Table scan on <temporary> (actual time=14.9..15 rows=829 loops=1)

-> Aggregate using temporary table (actual time=14.9..14.9 rows=829 loops=1)

-> Filter: ((game.critic_score > 9) and (game.critic_count > 50)) (cost=1196 rows=1302) (actual time=0.825..13.2 rows=830 loops=1)

-> Table scan on GAME (cost=1196 rows=11719) (actual time=0.814..12.1 rows=12043 loops=1)

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

De acordo com o plano de execução está ocorrendo o "*table scan*" na tabela **GAME** na qual era ele está varrendo todas as tuplas da tabela, e isso é algo ineficiente em uma consulta. Além disso, ele está varrendo todas as tuplas antes de aplicar o filtro na consulta, e o motivo disso é por conta da falta de índices nos filtros.

Diante disso, para que possamos resolver esses problemas precisamos criar um índice composto para evitar o "*table scan*" que está ocorrendo na tabela **GAME**.

CREATE INDEX critic_score_count_IDX ON GAME (critic_score, critic_count);

Com a criação do índice composto vai evitar o "*table scan*" na tabela **GAME**, entretanto ainda não é o melhor resultado, pois ainda assim está ocorrendo o uso de agregação na tabela temporária que é criada por conta do GROUP BY. Dessa forma, a gente ainda precisa criar mais outro índice para acelerar o filtro e a agregação.

CREATE INDEX grouping_IDX ON GAME (name, platform, jp_sales);

Agora a agregação será mais eficiente e eliminar a necessidade da tabela temporária.

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	GAME	NULL	ALL	index	Critic_score_count_IDX, grouping_IDX	143	NULL	11719	18.02	Using Where

Apresentando o resultado da nova execução do *ANALYZE*:

-> Group aggregate: max(game.jp_sales) (cost=1407 rows=2112) (actual time=1.39..34.7 rows=829 loops=1)

-> Filter: ((game.critic_score > 9) and (game.critic_count > 50)) (cost=1196 rows=2112) (actual time=1.36..33.8 rows=830 loops=1)

-> Index scan on GAME using grouping_IDX (cost=1196 rows=11719) (actual time=1.34..32.1 rows=12043 loops=1)

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

Com as mudanças feitas e a criação dos índices para que a consulta ficasse mais eficiente, é notório o quanto que as mudanças acabaram fornecendo uma melhor eficiência na consulta. A maior mudança foi com a criação dos índices a consulta parou de criar as tabelas temporárias de agregação da MAX(jp_sales) e também o uso do “*table scan*”, que no qual foi substituído por um “index” que irá reduzir o custo e o tempo de execução.