Query em SQL Padrão

```
CREATE VIEW pecas_conectam_placa_mae AS

SELECT
        p.id_produto,
        tint.fk_interface_id_interface
FROM
        tipo_produto tprod
INNER JOIN produto p
        ON p.fk_tipo_produto_id_tipo = tprod.id_tipo
INNER JOIN tem_interface tint
        ON p.id_produto = tint.fk_produto_id_produto
WHERE
        tprod.nome_tipo NOT IN ('placa mae', 'maquina')
```

Por si só, a visão só tem uma característica de otimização: a garantia da aplicação da regra “tri-join” onde essa visão seria um subquery. Para a otimização da consulta a ordem dos joins é feita a partir das relações com menor número de tuplas esperadas, gerando assim o menor custo de possível para realizar a query sem ajuda de algum outro artifício. É eficaz embutida em outras queries onde originalmente seria usada como subquery, quem dificultam a otimização de consultas nos SGBDS.

Materialização

Felizmente (ou infelizmente na visão do trabalho) o postgresql otimiza ambos os casos, então não há benefícios ganhos com a utilização da visão pura. Há entretanto uma distinta vantagem em usar essa visão de forma materializada. A query é “fria” na escrita, os valores retornados não mudam muito pois as interfaces e o tipo de um produto (salvo erro de cadastro) são constantes, o ritmo de cadastro de produtos novos também é lento.

Queries beneficiadas

1. Selecione o nome das placas-mãe e a quantidade de peças compatíveis com elas, ordenadas de forma decrescente.

A query abaixo foi envia na parte I do trabalho como uma das consultas e teria valor importante para alguém administrando o banco já que a partir dela é possível montar máquinas usando as peças compatíveis através das placas-mãe. Com a utilização da visão materializada, a subquery que ficava em seu lugar foi substituida. Consulta anterior agora utilizando a visão:

```
SELECT
        p.nome_produto placa_mae,
        COUNT(p.nome_produto) quantidade_pecas_compativeis
FROM
        produto p
INNER JOIN tipo_produto tprod
        ON p.fk_tipo_produto_id_tipo = tprod.id_tipo
INNER JOIN tem_interface tint
        ON p.id_produto = tint.fk_produto_id_produto
INNER JOIN materialized_pecas_conectam_placa_mae pecas
        ON tint.fk_interface_id_interface = pecas.fk_interface_id_interface
WHERE
        tprod.nome_tipo = 'placa mae'
GROUP BY
        p.nome_produto
ORDER BY COUNT(p.nome_produto) DESC;
```

Descrição textual da query:

Selecione o nome do produto e chamando a coluna de placa mae e conte a quantidade de linhas com o nome_produto, chamando a coluna de quantidade_pecas_compativeis da tabela produto, cruzando com a tabela tipo_produto para filtrar por placa mães, além de cruzar também com a tabela tem_interface para saber puxar as interfaces que a placa mae possui e cruzando também com uma view materializada materialized_pecas_conectam_placa_mae que da a informação das peças que conectam com a placa mãe, usando as suas ids para cruzar. Faça essa seleção onde o nome do tipo de produto seja igual à placa mãe, agrupando pelo nome do produto e ordernar de forma descrescente pela quantidade de produtos com o mesmo nome.

Resultado do Explain da Query

	QUERY PLAN text
2	Sort Key: (count(p.nome_produto)) DESC
3	Sort Method: quicksort Memory: 25kB
4	-> HashAggregate (actual rows=3 loops=1)
5	Group Key: p.nome_produto
6	-> Nested Loop (actual rows=24 loops=1)
7	-> Nested Loop (actual rows=48 loops=1)
8	-> Nested Loop (actual rows=48 loops=1)
9	-> Nested Loop (actual rows=12 loops=1)
10	-> Hash Join (actual rows=3 loops=1)
11	Hash Cond: (p.fk_tipo_produto_id_tipo = tprod.id_tipo)
12	-> Seq Scan on produto p (actual rows=18 loops=1)
13	-> Hash (actual rows=1 loops=1)
14	Buckets: 1024 Batches: 1 Memory Usage: 9kB
15	-> Index Scan using tipo_produto_nome_tipo_key on tipo_produto tprod ...
16	Index Cond: ((nome_tipo)::text = 'placa mae'::text)
17	-> Index Only Scan using tem_interface_pkey on tem_interface tint (actual row...
18	Index Cond: (fk_produto_id_produto = p.id_produto)
19	Heap Fetches: 12
20	-> Index Only Scan using tem_interface_pkey on tem_interface tint_1 (actual rows...
21	Index Cond: (fk_interface_id_interface = tint.fk_interface_id_interface)
22	Heap Fetches: 48
23	-> Index Scan using produto_pkey on produto p_1 (actual rows=1 loops=48)
24	Index Cond: (id_produto = tint_1.fk_produto_id_produto)
25	-> Index Scan using tipo_produto_pkey on tipo_produto tprod_1 (actual rows=0 loops=48)
26	Index Cond: (id_tipo = p_1.fk_tipo_produto_id_tipo)
27	Filter: ((nome_tipo)::text <> ALL ('{"placa mae",maquina}'::text[]))
28	Rows Removed by Filter: 0
29	Planning Time: 0.716 ms
30	Execution Time: 0.361 ms

Figura 1: Query sem visão

Resultado Da Query com Visão Materializada

	QUERY PLAN text
1	Sort (actual rows=3 loops=1)
2	Sort Key: (count(p.nome_produto)) DESC
3	Sort Method: quicksort Memory: 25kB
4	-> HashAggregate (actual rows=3 loops=1)
5	Group Key: p.nome_produto
6	-> Hash Join (actual rows=24 loops=1)
7	Hash Cond: (pecas.fk_interface_id_interface = tint.fk_interface_id_interface)
8	-> Seq Scan on materialized_pecas_conectam_placa_mae pecas (actual rows=16 loops...
9	-> Hash (actual rows=12 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Nested Loop (actual rows=12 loops=1)
12	-> Hash Join (actual rows=3 loops=1)
13	Hash Cond: (p.fk_tipo_produto_id_tipo = tprod.id_tipo)
14	-> Seq Scan on produto p (actual rows=18 loops=1)
15	-> Hash (actual rows=1 loops=1)
16	Buckets: 1024 Batches: 1 Memory Usage: 9kB
17	-> Index Scan using tipo_produto_nome_tipo_key on tipo_produto tprod (ac...
18	Index Cond: ((nome_tipo)::text = 'placa mae'::text)
19	-> Index Only Scan using tem_interface_pkey on tem_interface tint (actual rows=4...
20	Index Cond: (fk_produto_id_produto = p.id_produto)
21	Heap Fetches: 12
22	Planning Time: 0.456 ms
23	Execution Time: 0.185 ms

Figura 2: Query com visão materializada

Como pode ser visto, ao utilizar a materialização, o processamento da query, é bem mais breve, pois alguns passos,são poupados ao materializar uma view. Ao criar essa forma de view, percebe-se que o sgbd realiza menos operações, como por exemplo, a necessidade de um heap fetch, economizando no tempo de execução.

2. Selecione as chaves primárias e as frequências de uso das interfaces pelos produtos.

Essa query é nova e poderia ser usada em uma análise pelos fabricantes de peças ou de máquinas, principalmente de placas-mãe. O benefício gerado seria da mesma forma que fora para a query 1.

```
SELECT
        fk_interface_id_interface,
        COUNT(fk_interface_id_interface)
FROM
        materialized_pecas_conectam_placa_mae pecas
GROUP BY
        fk_interface_id_interface
ORDER BY fk_interface_id_interface DESC
```

Descrição Textual da Query

Selecione os ids da interface e conte os mesmos da tabela materialized_pecas_conectam_placa_mae, agrupando pela id da interface e ordenando pela mesma de forma decrescente

Alterações via visão

Como foi mencionado previamente, a visão é “fria” no quesito escrita e também não há muita utilidade em atualizar as três tabelas envolvidas a partir da visão, já que essa não traz o conjunto completo de colunas das tabelas, algo que seria necessário nesse caso particular.

Teorema “Tri-join”

Dado três relações A, B, C com cardinalidades |A|, |B| e |C| tal que |A| < |B| < |C|. A coluna A faz join com B e B faz join C. As sequências de joins de menor custo são:

$$(A \bowtie B) \bowtie C$$
$$(B \bowtie A) \bowtie C$$

Com as outras opções, mais custosas, sendo:

$$(C \bowtie B) \bowtie A$$
$$(B \bowtie C) \bowtie A$$

Note que para quaisquer duas tabelas X e Y, a complexidade no pior caso é o produto da cardinalidade de cada relação, cada tupla em X deve ser comparada em cada tupla em Y. A complexidade então fica como**:**

$$\mathbf{O(|X|)O(|Y|)}$$

Sendo assim, o número de tuplas retornado em cada join interno é de **|A||C|** e **|B||C|** respectivamente. Para o segundo join, o pior caso depende dessas quantidades retornadas no primeiro join e da quantidade na relação que sobrou. Logo as complexidades desse último join ficam iguais:

$$\mathbf{O(|A|)O(|B|)O(|C|) = O(|C|)(|B|)O(|A|)}$$

As complexidades finais são a soma das complexidades do primeiro e segundo join:

$$\mathbf{O(|A|)O(|B|) + O(|A|)O(|B|)O(|C|)}$$

$$\mathbf{O(|C|)O(|B|) + O(|A|)O(|B|)O(|C|).}$$

Como **|A| < |C|** a complexidade de pior caso das sequências de joins cujo primeiro join envolve a menor relação A é a menor dentre as possibilidades.