Regularidade da utilização das operações

No ato da realização de uma compra, chamamos a função com o nome "compra" que trata dos três casos possíveis: o caso com desconto na operação 1, o caso com desconto na operação 2, e o caso de uma compra sem descontos. Os *select* que verifica o cumprimento de ser melhor cliente é efetuado sempre que a compra é chamada. Visto isto, torna-se necessário a sua otimização. Os restantes *selects* como por exemplo a obtenção dos produtos abrangidos pelo desconto são executados apenas quando o cliente cumpre os requisitos e assim são executados com menor frequência.

Analisando o estado atual da base de dados, aproximadamente 55% dos clientes são clientes novos acreditamos que a operação 2 será a mais executada das três e a mais carente de otimização.

Por outro lado a 3ª operação é chamada quando o gestor da loja entender que é necessário repor stock. Assim prevemos que a 3ª operação não será tão utilizada como as outras duas visto que estas são utilizadas sempre que há uma compra e então haverão mais compras que pedidos de stock contudo será otimizada também.

Na criação dos índices tivemos em consideração o *overhead* relativo à memória necessária para o seu cálculo, por isso, tentámos utilizar índices extra apenas nas tabelas estritamente necessárias para garantir uma otimização clara das operações.

Operação 1 – Problemas de Otimização

Para esta operação iremos recorrer a índices que otimizem os *selects*. Começamos com uma das pesquisas mais custosas que temos e que é feito sempre que uma compra é feita, o *select* que vai procurar os 1000 melhores clientes, isto é, aqueles que tenham feito compras no top de productos do mês anterior.

	QUERY PLAN text
26	-> Limit (cost=1102.99..1103.49 rows=200 width=36) (actual time=28.631..28.728 rows=1000 loops=1)
27	-> Sort (cost=1102.99..1103.49 rows=200 width=36) (actual time=28.629..28.673 rows=1000 loops=1)
28	Sort Key: (sum(o.totalamount)) DESC, o.customerid
29	Sort Method: top-N heapsort Memory: 127kB
30	-> HashAggregate (cost=1092.85..1095.35 rows=200 width=36) (actual time=23.051..24.832 rows=4879 loops...
31	Group Key: o.customerid
32	-> Hash Join (cost=821.34..1072.85 rows=4000 width=20) (actual time=13.295..18.800 rows=6519 loops=1)
33	Hash Cond: (o.customerid = c.customerid)
34	-> Seq Scan on orders o (cost=0.00..220.00 rows=12000 width=20) (actual time=0.027..1.513 rows=120...
35	-> Hash (cost=738.00..738.00 rows=6667 width=4) (actual time=13.189..13.189 rows=10923 loops=1)
36	Buckets: 16384 (originally 8192) Batches: 1 (originally 1) Memory Usage: 513kB
37	-> Seq Scan on customers c (cost=0.00..738.00 rows=6667 width=4) (actual time=0.030..10.128 row...
38	Filter: (age > 50)
39	Rows Removed by Filter: 9077
40	Planning time: 0.970 ms
41	Execution time: 59.409 ms

Fazendo o *explain analyse* do *select* em questão, observamos que demorou cerca de 60 ms a executar e que carece de óbvia otimização. Sendo que este *select* é feito com um join iremos então otimizar a relação interna. Visto que a relação interna contém seleções tendo em conta a data do mês anterior decidimos criar um índice que tem em conta a data, utilizando o algoritmo B-Tree pois como estamos a fazer o índice na coluna da data, a estrutura não terá à partida de se reordenar tanto o que permite que os acessos sejam mais rápidos.

	QUERY PLAN text
20	Index Cond: ((orderdate >= date_trunc('month'::text, (CURRENT_DATE - '1 mon'::interval month))) ...
21	-> Hash (cost=1105.49..1105.49 rows=200 width=4) (never executed)
22	-> Subquery Scan on tc (cost=1102.99..1105.49 rows=200 width=4) (never executed)
23	-> Limit (cost=1102.99..1103.49 rows=200 width=36) (never executed)
24	-> Sort (cost=1102.99..1103.49 rows=200 width=36) (never executed)
25	Sort Key: (sum(o.totalamount)) DESC, o.customerid
26	-> HashAggregate (cost=1092.85..1095.35 rows=200 width=36) (never executed)
27	Group Key: o.customerid
28	-> Hash Join (cost=821.34..1072.85 rows=4000 width=20) (never executed)
29	Hash Cond: (o.customerid = c.customerid)
30	-> Seq Scan on orders o (cost=0.00..220.00 rows=12000 width=20) (never executed)
31	-> Hash (cost=738.00..738.00 rows=6667 width=4) (never executed)
32	-> Seq Scan on customers c (cost=0.00..738.00 rows=6667 width=4) (never executed)
33	Filter: (age > 50)
34	Planning time: 1.118 ms
35	Execution time: 0.304 ms

Após a implementação do índice, conseguimos que o tempo de execução diminuísse consideravelmente. No exemplo em cima obtivemos um tempo bastante baixo pois não tínhamos efetuado compras do mês anterior (não necessitando de verificar o resto das condições).

Resultados na execução da compra

Efetuamos um teste depois fazer uma compra, onde nessa mesma compra é aplicado o desconto, com índice e sem índice para compararmos o tempo de execução da função *compra*. A compra que efectuámos foi com o cliente com id = 16280 a comprar uma unidade do produto com id = 8.

	QUERY PLAN text
1	Result (cost=0.00..0.26 rows=1 width=4) (actual time=90.950..90.951 rows=1 loops=1)
2	Planning time: 0.055 ms
3	Execution time: 90.976 ms

Exemplo da compra sem índice: aproximadamente 91 ms

	QUERY PLAN text
1	Result (cost=0.00..0.26 rows=1 width=4) (actual time=61.684..61.685 rows=1 loops=1)
2	Planning time: 0.027 ms
3	Execution time: 61.699 ms

Exemplo da compra com índice: aproximadamente 62 ms

Operação 2 – Problemas de Otimização

A operação 2 tem a função de aplicar um desconto, numa seleção de produtos, na primeira compra dos clientes.

Otimizações no código Plpgsql

1-

Para vermos se era a primeira compra de um cliente efetuamos o seguinte *query*:

```
SELECT COUNT(DISTINCT CH.customerid) INTO n_vezes
FROM cust_hist CH
WHERE CH.customerid = cust
GROUP BY CH.orderid;
```

No caso de um cliente ter várias compras este *query* tem como resposta exatamente o número de compras que efetuou. Mas para verificarmos se é a primeira compra do cliente não precisamos de um *query* tão elaborado, para tal fizemos as seguintes modificações: retiramos o DISTINCT e o GROUP BY, assim o resultado da *query* deixa de ser o número exato de compras mas conseguimos verificar se é a primeira compra de um cliente ou não. Com esta simplificação mantemos o objetivo e diminuímos o tempo de execução.

2-

Nas entregas anteriores não focamos a atenção na otimização e então tínhamos o seguinte SELECT dentro do DECLARE, ou seja, era efetuado em todas as compras e não apenas quando sabíamos que era a primeira compra do cliente, passamos então esta operação para a zona do código que que apenas é executada quando sabemos que é a primeira compra do cliente.

```
SELECT array(SELECT p.prod_id
    FROM inventory i, products p
    WHERE (i.prod_id = p.prod_id)
    AND p.price < 15
    ORDER BY i.quan_in_stock DESC, p.prod_id
    LIMIT 50) into lista_prods_desc;
```

Otimizações utilizando índices

Nesta operação iremos otimizar o *select* que procura os 50 produtos com preço inferior a 15€ com maior quantidade de stock, pois é o maior bottleneck de toda a operação.

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Limit (cost=559.64..559.77 rows=50 width=8) (actual time=7.465..7.471 rows=50 loops=1)
2	-> Sort (cost=559.64..567.98 rows=3333 width=8) (actual time=7.464..7.466 rows=50 loops=1)
3	Sort Key: i.quan_in_stock DESC, p.prod_id
4	Sort Method: top-N heapsort Memory: 28kB
5	-> Hash Join (cost=267.66..448.92 rows=3333 width=8) (actual time=3.905..6.995 rows=2843 loops=1)
6	Hash Cond: (i.prod_id = p.prod_id)
7	-> Seq Scan on inventory i (cost=0.00..155.00 rows=10000 width=8) (actual time=0.020..0.965 rows=10000 loops=1)
8	-> Hash (cost=226.00..226.00 rows=3333 width=4) (actual time=3.831..3.831 rows=2843 loops=1)
9	Buckets: 4096 Batches: 1 Memory Usage: 132kB
10	-> Seq Scan on products p (cost=0.00..226.00 rows=3333 width=4) (actual time=0.018..3.189 rows=2843 loops=1)
11	Filter: (price < '15'::numeric)
12	Rows Removed by Filter: 7157
13	Planning time: 0.403 ms
14	Execution time: 7.582 ms

No *screenshot* em cima é possível observar a execução do *select* sem índices. Um dos bottlenecks deste select é o facto de perder tempo a fazer um scan completo à tabela dos

produtos. Este scan completo à tabela pode ser evitado através da criação de um índice na coluna dos preços.

	QUERY PLAN text
1	Limit (cost=542.42..542.55 rows=50 width=8) (actual time=3.735..3.741 rows=50 loops=1)
2	-> Sort (cost=542.42..550.75 rows=3333 width=8) (actual time=3.734..3.735 rows=50 loops=1)
3	Sort Key: i.quan_in_stock DESC, p.prod_id
4	Sort Method: top-N heapsort Memory: 28kB
5	-> Hash Join (cost=250.44..431.70 rows=3333 width=8) (actual time=1.411..3.416 rows=2843 loops=1)
6	Hash Cond: (i.prod_id = p.prod_id)
7	-> Seq Scan on inventory i (cost=0.00..155.00 rows=10000 width=8) (actual time=0.026..0.649 rows=10000 loops=1)
8	-> Hash (cost=208.78..208.78 rows=3333 width=4) (actual time=1.357..1.357 rows=2843 loops=1)
9	Buckets: 4096 Batches: 1 Memory Usage: 132kB
10	-> Bitmap Heap Scan on products p (cost=66.12..208.78 rows=3333 width=4) (actual time=0.446..0.986 rows=2843 loops=1)
11	Recheck Cond: (price < '15'::numeric)
12	Heap Blocks: exact=101
13	-> Bitmap Index Scan on idx (cost=0.00..65.28 rows=3333 width=0) (actual time=0.422..0.422 rows=2843 loops=1)
14	Index Cond: (price < '15'::numeric)
15	Planning time: 0.265 ms
16	Execution time: 3.851 ms

Resultados na execução da compra

Utilizando o caso de uso que lançámos na fase anterior, fazemos uma compra com o cliente com o id = 4 (cliente sem compras anteriores) que compra 50 unidades do produto com o id = 10000. Sem qualquer otimização obtemos o seguinte resultado:

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Result (cost=0.00..0.26 rows=1 width=4) (actual time=73.028..73.028 rows=1 loops=1)
2	Planning time: 0.034 ms
3	Execution time: 73.388 ms

Após a aplicação do índice no *select* da lista que contém os produtos com desconto, obtemos os seguintes resultados:

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Result (cost=0.00..0.26 rows=1 width=4) (actual time=58.440..58.440 rows=1 loops=1)
2	Planning time: 0.018 ms
3	Execution time: 58.449 ms

Utilizando o índice anterior em conjunto com índice criado para otimizar o select da operação 1 obtemos um resultado com um valor muito inferior. Isto deve-se ao facto do select da operação 1 necessitar de ser realizado em todas as compras.

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Result (cost=0.00..0.26 rows=1 width=4) (actual time=7.023..7.023 rows=1 loops=1)
2	Planning time: 0.020 ms
3	Execution time: 7.033 ms

Para este caso de uso houve uma redução do tempo de execução significativa. É um resultado bastante positivo uma vez que será a operação mais utilizada das três que desenvolvemos.

Operação 3 - Problemas de Otimização

A operação 3 tem como objetivo dividir os produtos na base de dados em duas categorias, os mais vendidos e os medianamente vendidos e pedir 150% de stock para os mais vendidos e 120% para os medianamente vendidos.

Em seguida encontra-se o screenshot com o tempo de execução da operação 3 para o caso de uso usado na etapa passada. Este valor irá servir de referência para comparações de desempenho.

Data Output | Explain | Messages | Notifications | Query History

	QUERY PLAN text
1	Result (cost=0.00..0.26 rows=1 width=4) (actual time=29.071..29.071 rows=1 loops=1)
2	Planning time: 0.050 ms
3	Execution time: 29.445 ms

Otimizações no código Plpgsql

Como primeira alteração fomos ao array dos produtos e na fase anterior utilizámos 2 selects, um para criar a lista e outro para obter o tamanho da mesma. Ora isto é desnecessário uma vez que podemos utilizar a biblioteca do PostgresSQL para calcular o tamanho do array.

	QUERY PLAN text
1	Result (cost=0.00..0.26 rows=1 width=4) (actual time=22.379..22.380 rows=1 loops=1)
2	Planning time: 0.017 ms
3	Execution time: 22.741 ms

Como podemos observar neste screenshot para a mesma operação utilizando o array length em vez de um select conseguimos melhorar tanto o *Planning Time* como o *Execution Time*.

Otimizações utilizando índices

Utilizando um índice B-tree na tabela *inventory,* conseguimos ordenar as linhas da tabela pela da percentagem de stock*.* Visto isto, quando fazemos uma procura pelos itens com menos de 20% de stock, deixa de ser necessário fazer um scan completo à tabela.

	QUERY PLAN text
1	Result (cost=0.00..0.26 rows=1 width=4) (...
2	Planning time: 0.017 ms
3	Execution time: 14.088 ms

Execução da operação 3 após a introdução do índice.