UNIVERSIDADE FEDERAL DE SÃO CARLOS – CAMPUS SOROCABA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

CURSO: SISTEMAS DE BANCOS DE DADOS

PROF.ª SAHUDY GONZÁLES

PROJETO PRÁTICO: E-Commerce “Old Fashion”

FASE: Final

TEMA: 4. HISTÓRICO DE LOJAS + PRODUTOS

GRUPO 05

CAIO FERNANDO PERES 769298

ISABELLE TOMAZELA BARIZON 759507

RAFAEL TOFOLI SEREICIKAS 760934

Data: 10/01

SOROCABA
2020

SUMÁRIO

CONSIDERAÇÕES FINAIS 34

1. MINIMUNDO

Em nosso *e-commerce*, cada loja possuirá sua própria página, tendo suas informações armazenadas relativas apenas ao seu nome, código de identificação, telefone e endereço. Este último não é obrigatório pois uma loja pode ser totalmente virtual. Cada produto está atrelado a uma loja, significando que, caso existam dois produtos iguais vendidos em lojas diferentes, eles serão registrados de maneira individual no banco de dados. Estes terão suas informações armazenadas relativas ao nome, preço, estoque disponível na loja e caso seja informado, seu fabricante e país de origem. Em relação a categoria do produto, a loja terá que escolher entre opções pré-definidas pelo próprio *e-commerce* (isso foi feito apenas como uma maneira de organizar melhor os dados) e é obrigatório que o produto tenha apenas uma categoria. Será mantido um histórico das transações de todos os produtos, com o nome do produto, CPF do comprador, data da compra, quantidade, status e CEP de entrega. Cada CPF pertence a um comprador que também estará registrado no sistema, junto de seu respectivo nome, telefone, CEP e número da residência. O CEP de entrega de um produto pode não ser o mesmo cadastrado pelo comprador. Um comprador só pode comprar o mesmo produto em uma única transação por dia. Caso deseje comprar novamente, deverá esperar o dia seguinte, mesmo que o pedido seja cancelado.

2. DIAGRAMA DO ESQUEMA DO BANCO DE DADOS

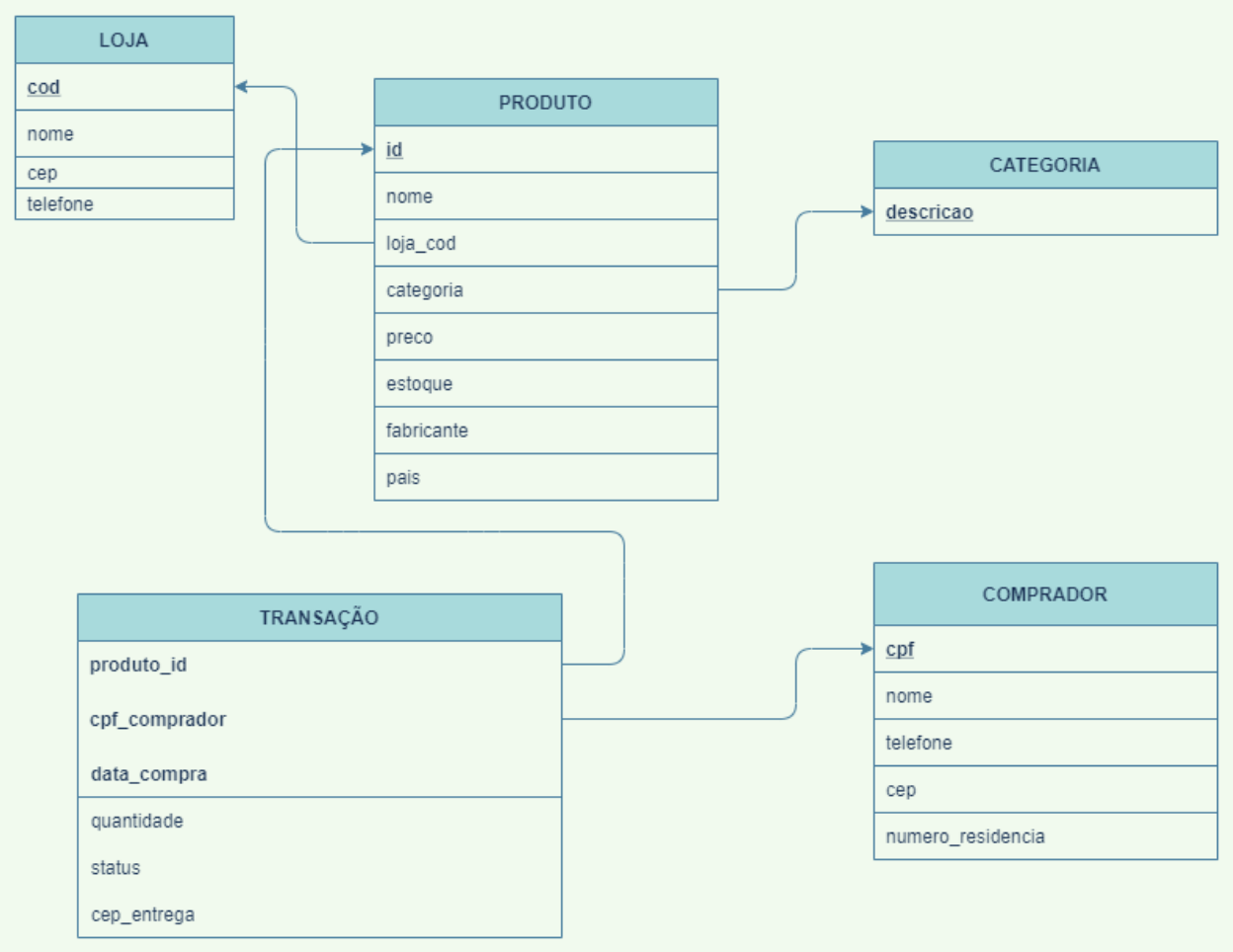

Imagem 1. Diagrama do esquema de banco de dados.

3. ENUNCIADO DAS CONSULTAS

3.1 BUSCA ABSOLUTA

Informações sobre os produtos, de uma certa loja, cuja quantidade total de unidades vendidas está dentro de um dado intervalo

```
SELECT produto.nome as nome_produto, categoria, preco, ROUND(AVG(quantidade), 2) AS media_por_lote, SUM(quantidade) AS quantidade_total FROM transacao, produto, loja
    WHERE produto_id = id AND cod = loja_cod AND loja.nome = <nome_loja>
    GROUP BY id
    HAVING SUM(quantidade) > <quantidade_minima> AND SUM(quantidade) <= <quantidade_maxima>
```

Campos de visualização do resultado: `nome_loja, nome_produto, categoria, preco, media_por_lote, quantidade_total`
Campos de busca (ou das condições): `sum(transacao.quantidade) (absoluta), sum(transacao.quantidade) (absoluta)`
Operadores das condições: `sum(transacao.quantidade) (>= , <=)`

3.2 BUSCA RELATIVA

Quantidade de produtos entregues de uma certa loja (usando `LIKE`) em um determinado período agrupados por sua categoria ordenado descendente pela quantidade.

```
SELECT loja.nome, loja.cod, categoria, sum(vendas.total) FROM
    (SELECT produto_id  as id, sum(quantidade) as total FROM
transacao WHERE
      transacao.data_compra    BETWEEN    <DATA_INICIAL>    AND
<DATA_FINAL> AND status = 'Entregue' GROUP BY produto_id)
as vendas, loja, produto
WHERE  vendas.id  =  produto.id  AND  loja.cod  =  loja_cod  AND
loja.nome LIKE <NOME_LOJA>
GROUP BY loja.cod, produto.categoria
ORDER BY loja.cod, sum(vendas.total) DESC
```

Campos de visualização do resultado: `cod, categoria, sum`
Campos de busca (ou das condições): `transacao.data_compra (absoluta), transacao.status (absoluta), loja.nome (relativa)`
Operadores das condições: `transacao.data_compra (>=,<=), transacao.status (=), loja.nome (ILIKE)`

4. POPULANDO O BD

Para a geração de dados, foi utilizado um *script* próprio escrito em *Python*, que se encontra junto à entrega do relatório no AVA2 e ao novo backup do banco de dados contendo os índices criados nesta fase.

Todos os números e nomes encontrados nas tabelas foram gerados aleatoriamente através de 3 bibliotecas: *Faker*, *Random* e *Barnum*. No entanto muitos dados precisavam de alguma formatação especial para serem inseridos no banco de

dados tomemos como um exemplo o CEP, no qual precisa estar no formato (XXXXX-XXX), para resolver isto nós primeiro geramos um número de 5 dígitos seguido de um de 3 dígitos e os concatenamos com um ‘ - ’ entre eles.

4.1 TABELAS CATEGORIA E LOJA

Nosso grupo decidiu que ambas as tabelas tenham apenas 300 registros cada, essa decisão foi tomada pois em ambas as nossas consultas os resultados seriam muito limitados caso houvessem muitas lojas ou muitas categorias. A tabela Categoria em específico possui apenas o atributo `descricao`, por isso o seu tamanho em bytes muito inferior ao da tabela Loja.

Nome da tabela	Categoria
Número de registros	300
Tamanho (bytes)	16.384

Tabela 1

Nome da tabela	Loja
Número de registros	300
Tamanho (bytes)	24.576

Tabela 2

4.2 TABELAS COMPRADOR E PRODUTO

De acordo com as especificações do trabalho essas duas tabelas deveriam ter 500.000 registros cada, porém decidimos deixar com ambos um valor um pouco menor, de 400.000 registros. A razão disso poderá ser melhor explicada logo em seguida, quando formos falar sobre a tabela `transacao`.

Nome da tabela	Comprador
Número de registros	400.000
Tamanho (bytes)	34.832.384

Tabela 3

Nome da tabela	Produto
Número de registros	400.000
Tamanho (bytes)	44.564.480

Tabela 4

4.3 TABELA TRANSAÇÃO

A tabela transação é a que possui mais registros no banco de dados com um total de 2.000.000 de registros, fizemos isso pois esta tabela utiliza como uma chave primária composta do CPF do comprador com o ID de produto, a razão de diminuir a quantidade de registros nestas duas tabelas foi justamente aumentar a quantidade de intercalações que ocorrem nessa tabela.

Graças a essa diferença na quantidade de registro, conseguimos fazer com que cada comprador tenha comprado em média 5 produtos, e cada produto tenha sido comprado por 5 pessoas diferentes.

Nome da tabela	Transação
Número de registros	2.000.000
Tamanho (bytes)	153.198.592

Tabela 5

5. TÉCNICAS DE ACESSO EFICIENTE AO BANCO DE DADOS

5.1 BUSCA ABSOLUTA

Versão do PostgreSQL para esta consulta: PostgreSQL 12.4, compiled by Visual C++ build 1914, 64-bit

Relembrando a consulta: Informações sobre os produtos, de uma certa loja, cuja quantidade total de unidades vendidas está dentro de um dado intervalo.

```
SELECT produto.nome as nome_produto, categoria, preco, ROUND(AVG(quantidade), 2) AS media_por_lote, SUM(quantidade) AS quantidade_total FROM transacao, produto, loja
    WHERE produto_id = id AND cod = loja_cod AND loja.nome = <nome_loja>
    GROUP BY id
HAVING SUM(quantidade) > <quantidade_minima> AND SUM(quantidade) <= <quantidade_maxima>
```

Para a otimização da consulta, a simples criação de alguns índices foi mais que efetivo. Foram eles:

```
CREATE INDEX ON loja(nome varchar_pattern_ops)
CREATE INDEX ON produto(loja_cod)
CREATE INDEX ON transacao(produto_id)
CREATE INDEX ON transacao(quantidade)
```

Os que surtiram mais efeito no desempenho foram os criados em `loja_cod` e `produto_id`, que, no plano de execução que será visto a seguir, pode ser observado que a varredura passou de sequencial para binária durante as operações de junção.

Não mudamos a estrutura da consulta em si, tentamos mover a condição do `nome_loja` para uma própria subconsulta dentro da cláusula `FROM`, mas a tabela `loja` é tão pequena que acabou diminuindo o desempenho, então mantivemos como estava anteriormente.

5.1.1 Plano da Consulta Inicial

Data Output

	QUERY PLAN text
1	Finalize GroupAggregate (cost=38893.87..39776.93 rows=33 width=77) (actual time=2805.166..2839.977 rows=701 loops=1)
2	Group Key: produto.id
3	Filter: ((sum(transacao.quantidade) > 20) AND (sum(transacao.quantidade) <= 50))
4	Rows Removed by Filter: 650
5	-> Gather Merge (cost=38893.87..39590.73 rows=5556 width=77) (actual time=2805.148..2824.444 rows=3291 loops=1)
6	Workers Planned: 2
7	Workers Launched: 2
8	-> Partial GroupAggregate (cost=37893.84..37949.40 rows=2778 width=77) (actual time=2513.316..2517.273 rows=1097 loops=3)
9	Group Key: produto.id
10	-> Sort (cost=37893.84..37900.79 rows=2778 width=41) (actual time=2513.296..2513.666 rows=2242 loops=3)
11	Sort Key: produto.id
12	Sort Method: quicksort Memory: 309kB
13	Worker 0: Sort Method: quicksort Memory: 297kB
14	Worker 1: Sort Method: quicksort Memory: 294kB
15	-> Parallel Hash Join (cost=7564.05..37734.95 rows=2778 width=41) (actual time=597.623..2508.466 rows=2242 loops=3)
16	Hash Cond: (transacao.produto_id = produto.id)
17	-> Parallel Seq Scan on transacao (cost=0.00..27034.33 rows=833333 width=8) (actual time=15.961..1706.663 rows=666667 loops=3)
18	-> Parallel Hash (cost=7557.12..7557.12 rows=555 width=37) (actual time=574.502..574.504 rows=455 loops=3)
19	Buckets: 2048 Batches: 1 Memory Usage: 176kB
20	-> Hash Join (cost=6.76..7557.12 rows=555 width=37) (actual time=14.255..573.472 rows=455 loops=3)
21	Hash Cond: (produto.loja_cod = loja.cod)
22	-> Parallel Seq Scan on produto (cost=0.00..7106.67 rows=166667 width=41) (actual time=4.388..533.021 rows=133333 loops=3)
23	-> Hash (cost=6.75..6.75 rows=1 width=4) (actual time=0.317..0.318 rows=1 loops=3)
24	Buckets: 1024 Batches: 1 Memory Usage: 9kB
25	-> Seq Scan on loja (cost=0.00..6.75 rows=1 width=4) (actual time=0.218..0.303 rows=1 loops=3)
26	Filter: ((nome)::text = 'Harper Group'::text)
27	Rows Removed by Filter: 299
28	Planning Time: 1.067 ms
29	Execution Time: 2840.438 ms

Imagem 2. Plano da consulta inicial da busca absoluta

Primeiramente, na linha 25, foi feita uma busca sequencial na tabela `loja` filtrando pelo seu nome. Depois, foi colocada, na estrutura de mapeamento *Hash*, na linha 23, ao mesmo tempo que foi executada, na linha 22, uma busca sequencial (que é feita paralelamente) na tabela `produto` com os resultados da busca sequencial anterior. E então tivemos uma junção por *Hash* na linha 20. Após isso, foi colocada numa estrutura de mapeamento *Hash*, na linha 18, ao mesmo tempo que houve uma busca sequencial (feita paralelamente) na tabela `transacao`, na linha 17. Posteriormente, tivemos uma junção *Hash* em paralelo na linha 15. Seu resultado foi ordenado pelo método *QuickSort* na linha 10. Então, houve uma agregação parcial, na linha 8, pelo id da tabela `produto`. Em seguida, foi feito um *Gather Merge* na linha 5. Por último, na linha 1, ocorre a finalização da agregação pelo `id` da tabela `produto` e a filtragem em relação às quantidades pedidas na consulta.

É possível notar um tempo elevado para o retorno da busca. Isso se deve pelo fato de que a consulta utiliza buscas sequenciais, filtros e ordenação, nas linhas 26, 25, 22, 17, 12 e 3, que são operações bem custosas. Utilizam uma forma não otimizada de buscar pelos dados, tendo que verificar todos os dados em busca pelo critério.

5.1.2 Plano da Consulta Otimizada

Data Output

	QUERY PLAN text
1	HashAggregate (cost=4353.37..4470.12 rows=33 width=77) (actual time=20.947..22.117 rows=701 loops=1)
2	Group Key: produto.id
3	Filter: ((sum(transacao.quantidade) > 20) AND (sum(transacao.quantidade) <= 50))
4	Rows Removed by Filter: 650
5	-> Nested Loop (cost=27.22..4270.03 rows=6667 width=41) (actual time=0.684..17.354 rows=6727 loops=1)
6	-> Nested Loop (cost=26.79..3157.72 rows=1333 width=37) (actual time=0.665..2.670 rows=1364 loops=1)
7	-> Seq Scan on loja (cost=0.00..6.75 rows=1 width=4) (actual time=0.038..0.076 rows=1 loops=1)
8	Filter: ((nome)::text = 'Harper Group'::text)
9	Rows Removed by Filter: 299
10	-> Bitmap Heap Scan on produto (cost=26.79..3137.59 rows=1338 width=41) (actual time=0.601..2.154 rows=1364 loops=1)
11	Recheck Cond: (loja_cod = loja.cod)
12	Heap Blocks: exact=1209
13	-> Bitmap Index Scan on produto_loja_cod_idx (cost=0.00..26.46 rows=1338 width=0) (actual time=0.402..0.402 rows=1364 loops=1)
14	Index Cond: (loja_cod = loja.cod)
15	-> Index Scan using transacao_produto_id_idx on transacao (cost=0.43..0.77 rows=6 width=8) (actual time=0.005..0.009 rows=5 loops=1364)
16	Index Cond: (produto_id = produto.id)
17	Planning Time: 2.304 ms
18	Execution Time: 22.974 ms

Imagem 3. Plano da consulta otimizada da busca absoluta

Primeiramente o Postgresql abre o procedimento de agregação pelo `id` do `produto`, na linha 1, e então abre o procedimento de filtragem sob a condição da soma da quantidade estar dentro de um determinado intervalo.

Após isso, houve uma abertura de procedimento de busca, na linha 5, e outro novamente dentro deste, na linha 6. Então acontece um *scan* sequencial na loja buscando pelo nome da `loja`, na linha 7. Depois, acontece um *Bitmap heap scan*, na linha 10, e um *bitmap index scan*, na linha 13, no código da loja. Isso aconteceu no loop mais interno. No loop externo, na linha 15, foi utilizado o índice no *`produto_id`* para uma busca, e então os valores são retornados.

Esses *scans* são mais rápidos do que os *scans* sequenciais, pois utilizam formas inteligentes de buscarem os dados, como ao usar os índices previamente criados, nas linhas 15, 13 e 10, diminuindo consideravelmente o tempo da consulta.

5.1.3 Tabelas de Comparação do Tempo de Execução

- Nome da loja: “Harper Group”. Quantidade mínima: 20. Quantidade máxima: 50

Tempo de execução	**Consulta Inicial**	**Consulta Otimizada**	**Diferença (%)**
1ª execução	2 secs 591 msec.	125 msec.	95.17%
2ª execução	2 secs 710 msec.	133 msec.	95.09%
3ª execução	3 secs 745 msec	114 msec.	96.95%
4ª execução	3 secs 322 msec.	117 msec.	96.47%
5ª execução	3 secs 562 msec.	142 msec.	96.01%
Média	3 secs 186 msec	126 msec.	96.04%

Tabela 6. Apresentação de resultados de tempo de execução. Fórmula para cálculo do % = ((Vot - Vini)/Vini × 100)

-

➢ Nome da loja: “Harper Group”. Quantidade mínima: 5. Quantidade máxima: 200

Tempo de execução	Consulta Inicial	Consulta Otimizada	Diferença (%)
1ª execução	3 secs 59 msec.	112 msec.	96.33%
2ª execução	3 secs 664 msec.	118 msec.	96.77%
3ª execução	3 secs 970 msec.	133 msec.	96.64%
4ª execução	3 secs 358 msec.	131 msec.	96.09%
5ª execução	2 secs 827 msec.	128 msec.	95.47%
Média	3 secs 375 msec	124 msec.	96.32%

Tabela 7. Apresentação de resultados de tempo de execução. Fórmula para cálculo do % = ((Vot - Vini)/Vini × 100)

5.2 BUSCA RELATIVA

Versão do PostgreSQL para esta consulta: PostgreSQL 12.5 (Ubuntu 12.5-0ubuntu0.20.04.1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0, 64-bit.

Relembrando a consulta: Quantidade de produtos entregues de uma certa loja (usando `LIKE`) em um determinado período agrupados por sua categoria ordenada descendente pela quantidade.

```
    SELECT loja.nome, loja.cod, categoria, sum(vendas.total)
FROM
   (SELECT produto_id  as id, sum(quantidade) as total FROM
transacao WHERE
     transacao.data_compra   BETWEEN   <DATA_INICIAL>   AND
<DATA_FINAL> AND status = 'Entregue' GROUP BY produto_id)
as vendas, loja, produto
WHERE vendas.id = produto.id AND loja.cod = loja_cod AND
loja.nome LIKE <NOME_LOJA>
GROUP BY loja.cod, produto.categoria
ORDER BY loja.cod, sum(vendas.total) DESC
```

5.2.1 Considerações iniciais

Tivemos muita dificuldade para conseguir otimizar esta consulta, pois em muitos dos casos, o desempenho acabava ficando pior com os índices quando comparados com a consulta sem eles.

Mostraremos a seguir todas nossas tentativas de otimização e nossas estratégias usadas durante o desenvolvimento do projeto. Assim como o melhor resultado que conseguimos obter e as condições em que ele é possível.

5.2.1.1 Mudanças na estrutura da consulta

Tentamos mudar a estrutura da consulta para tentar obter um desempenho melhor, porém nenhuma das alternativas chegaram a serem melhores que a nossa original.

Algumas das alternativas foram: a criação de uma subconsulta na cláusula `FROM` que continha as lojas com o nome que foi especificado; a passagem da tabela `loja` para a subconsulta da tabela `transacao`; e até a remoção de todas as subconsultas. No final concordamos em procurar outras maneiras de otimizar a consulta.

5.2.1.2 Visão materializada

Uma proposta para a otimização foi a criação de uma visão materializada, no qual o resultado de uma subconsulta já estaria salvo no próprio armazenamento do sistema, tirando a necessidade desta ser executada e aumentando o desempenho. Porém ela não surtiu efeito.

```
CREATE MATERIALIZED VIEW total_transacoes
AS (SELECT data_compra , sum(quantidade) as total, produto_id as id FROM transacao WHERE status = 'Enviado' GROUP BY data_compra, produto_id)
```

O problema foi que, para que funcionasse no nosso contexto, seria necessário um agrupamento de três atributos que juntos não se repetiam. No final a visão ficou com quase o mesmo tamanho da tabela original e não houve diferença no desempenho.

5.2.1.3 Índices criados

A utilização de índices acabou sendo nossa única opção de otimização. Os primeiros índices criados foram os dos atributos de chave estrangeira, eles são usados em todos os planos de execução que usem junção, independentemente das variáveis na consulta. Aumentaram o desempenho da primeira consulta consideravelmente, porém nesta, nem tanto:

```
CREATE INDEX ON produto(loja_cod)
CREATE INDEX ON transacao(produto_id)
```

Os índices nos atributos da tabela `transacao` são usados dependendo das variáveis inseridas pelo usuário. O índice em `data_compra` não é usado em períodos grandes de tempo, somente nos períodos que sejam aproximadamente menores que um mês:

```
CREATE INDEX ON transacao(data_compra, status)
CREATE INDEX ON transacao(data_compra)
CREATE INDEX ON transacao(status)
```

Tentamos criar mais um índice desta vez no atributo `nome` da tabela `loja`, porém não chegou a ser usado, a tabela `loja` no final possuía muitas poucas tuplas e o plano de execução resolveu não usar:

```
CREATE INDEX ON loja(nome varchar_pattern_ops)
```

Vale observar que também tentamos a utilização destes índices nas outras estruturas de consultas que criamos, porém elas continuaram com o desempenho pior que a nossa original.

5.2.2 Plano da Consulta Inicial

	QUERY PLAN text
1	Sort (cost=46358.50..46359.03 rows=210 width=48) (actual time=10625.166..10625.854 rows=199 loops=1)
2	Sort Key: loja.cod, (sum(vendas.total)) DESC
3	Sort Method: quicksort Memory: 40kB
4	-> GroupAggregate (cost=46345.68..46350.40 rows=210 width=48) (actual time=10595.256..10596.741 rows=199 loops=1)
5	Group Key: loja.cod, produto.categoria
6	-> Sort (cost=46345.68..46346.20 rows=210 width=24) (actual time=10595.203..10595.914 rows=225 loops=1)
7	Sort Key: loja.cod, produto.categoria
8	Sort Method: quicksort Memory: 42kB
9	-> Hash Join (cost=37988.72..46337.58 rows=210 width=24) (actual time=10212.052..10593.500 rows=225 loops=1)
10	Hash Cond: (produto.id = vendas.id)
11	-> Gather (cost=1006.79..9345.14 rows=4000 width=20) (actual time=10.359..388.924 rows=3961 loops=1)
12	Workers Planned: 2
13	Workers Launched: 2
14	-> Hash Join (cost=6.79..7945.14 rows=1667 width=20) (actual time=3.403..4851.058 rows=1320 loops=3)
15	Hash Cond: (produto.loja_cod = loja.cod)
16	-> Parallel Seq Scan on produto (cost=0.00..7494.67 rows=166667 width=20) (actual time=2.823..4811.834 rows=133333 loops=3)
17	-> Hash (cost=6.75..6.75 rows=3 width=4) (actual time=0.109..0.111 rows=3 loops=3)
18	Buckets: 1024 Batches: 1 Memory Usage: 9kB
19	-> Seq Scan on loja (cost=0.00..6.75 rows=3 width=4) (actual time=0.041..0.098 rows=3 loops=3)
20	Filter: ((nome)::text ~~ 'An%'::text)
21	Rows Removed by Filter: 297
22	-> Hash (cost=36719.35..36719.35 rows=21007 width=12) (actual time=10201.483..10201.705 rows=22770 loops=1)
23	Buckets: 32768 Batches: 1 Memory Usage: 1235kB
24	-> Subquery Scan on vendas (cost=36299.21..36719.35 rows=21007 width=12) (actual time=10183.536..10193.793 rows=22770 loops=1)
25	-> Finalize HashAggregate (cost=36299.21..36509.28 rows=21007 width=12) (actual time=10183.534..10190.941 rows=22770 loops=1)
26	Group Key: transacao.produto_id
27	-> Gather (cost=34320.31..36209.26 rows=17990 width=12) (actual time=10163.723..10170.954 rows=23187 loops=1)
28	Workers Planned: 2
29	Workers Launched: 2
30	-> Partial HashAggregate (cost=33320.31..33410.26 rows=8995 width=12) (actual time=10164.499..10167.186 rows=7729 loops=3)
31	Group Key: transacao.produto_id
32	-> Parallel Seq Scan on transacao (cost=0.00..33275.33 rows=8995 width=8) (actual time=13.455..10146.473 rows=7802 loops=3)
33	Filter: ((data_compra >= '2019-04-15'::date) AND (data_compra <= '2019-04-30'::date) AND ((status)::text = 'Entregue'::text))
34	Rows Removed by Filter: 658864
35	Planning Time: 407.715 ms
36	Execution Time: 10629.636 ms

Imagem 4. Plano da consulta inicial da busca relativa

Primeiramente é aberto o procedimento de organização dos dados a serem apresentados através do *Quicksort* na ordem decrescente pela soma do total de vendas e código da loja, na linha 1.

Após isso, é aberto o procedimento de agregação pelo código da loja e categoria do produto, na linha 4. Então é aberto outro *Quicksort* pelo código da loja e categoria do produto, na linha 6, e uma junção pelo `id` das tabelas `vendas` e `produto`, na linha 9.

Depois, é aberto um *gather* com 2 *workers*, na linha 11, e novamente é aberto mais uma junção pelo código da tabela `produto` e `loja`, na linha 14, e então começa um *scan* sequencial no produto, na linha 16. Após isso é aberto um procedimento de *Hash*, na linha 17, e então um procedimento, na linha 19, de *scan* sequencial na `loja` filtrando pelo nome da loja representando uma *substring*.

Então, é aberto um novo *Hash*, na linha 22, *scan* da *subquery* na tabela `vendas`, na linha 24, uma agregação pelo `id` do produto da `transacao`, na linha 25, um *gather* com 2 *workers*, na linha 27, mais uma agregação parcial, na linha 30, e um *scan* sequencial, na linha 32, na `transacao` sob a condição da data de compra estar sob um intervalo e o *status* estar definido como entregue.

Finalmente, quando acaba esse último *scan*, os procedimentos são retornados para finalizar a agregação parcial, o *gather*, a agregação de fato, o scan da *subquery*, o *Hash*, o *scan* na *loja*, o outro *Hash*, o *scan* no `produto`, a junção do código da loja e produto, o gather, a junção do `id` das `vendas` e do `produto`, o *Quicksort* no código da loja e categoria do produto, a agregação pelo código da loja e categoria do produto e o *Quicksort* pelo código da loja e pela soma do total de vendas, decrescentemente.

Podemos perceber que a consulta é custosa, pois envolve o uso de ordenação, buscas sequenciais e filtros, nas linhas 33, 32, 20, 19, 16, 8 e 3. São operações de natureza custosa e complexidade linear.

5.2.3 Plano da Consulta Otimizado - Caso 1

```
1  Sort  (cost=29931.30..29931.83 rows=210 width=48) (actual time=7617.053..7617.075 rows=199 loops=1)
2    Sort Key: loja.cod, (sum(vendas.total)) DESC
3    Sort Method: quicksort  Memory: 40kB
4    -> GroupAggregate  (cost=29918.48..29923.20 rows=210 width=48) (actual time=7616.522..7616.756 rows=199 loops=1)
5        Group Key: loja.cod, produto.categoria
6        -> Sort  (cost=29918.48..29919.00 rows=210 width=24) (actual time=7616.506..7616.534 rows=225 loops=1)
7            Sort Key: loja.cod, produto.categoria
8            Sort Method: quicksort  Memory: 42kB
9            -> Hash Join  (cost=21502.94..29910.38 rows=210 width=24) (actual time=2804.032..7614.152 rows=225 loops=1)
10               Hash Cond: (produto.id = vendas.id)
11               -> Nested Loop  (cost=26.79..8423.73 rows=4000 width=20) (actual time=42.809..4912.614 rows=3961 loops=1)
12                   -> Seq Scan on loja  (cost=0.00..6.75 rows=3 width=4) (actual time=0.024..0.081 rows=3 loops=1)
13                       Filter: ((nome)::text ~~ 'An%'::text)
14                       Rows Removed by Filter: 297
15                   -> Bitmap Heap Scan on produto  (cost=26.79..2792.28 rows=1338 width=20) (actual time=30.935..1636.013 rows=1320 loops=3)
16                       Recheck Cond: (loja_cod = loja.cod)
17                       Heap Blocks: exact=3545
18                       -> Bitmap Index Scan on produto_loja_cod_idx  (cost=0.00..26.46 rows=1338 width=0) (actual time=25.891..25.891 rows=1320 loops=3)
19                           Index Cond: (loja_cod = loja.cod)
20               -> Hash  (cost=21213.56..21213.56 rows=21007 width=12) (actual time=2692.807..2692.809 rows=22770 loops=1)
21                   Buckets: 32768  Batches: 1  Memory Usage: 1235kB
22                   -> Subquery Scan on vendas  (cost=20793.42..21213.56 rows=21007 width=12) (actual time=2676.779..2687.105 rows=22770 loops=1)
23                       -> HashAggregate  (cost=20793.42..21003.49 rows=21007 width=12) (actual time=2676.777..2682.885 rows=22770 loops=1)
24                           Group Key: transacao.produto_id
25                           -> Bitmap Heap Scan on transacao  (cost=575.37..20685.49 rows=21587 width=8) (actual time=72.872..2636.698 rows=23407 loops=1)
26                               Recheck Cond: ((data_compra >= '2019-04-15'::date) AND (data_compra <= '2019-04-30'::date))
27                               Filter: ((status)::text = 'Entregue'::text)
28                               Rows Removed by Filter: 5846
29                               Heap Blocks: exact=14877
30                               -> Bitmap Index Scan on transacao_data_compra_idx  (cost=0.00..569.98 rows=26955 width=0) (actual time=43.327..43.327 rows=29253 loops=1)
31                                   Index Cond: ((data_compra >= '2019-04-15'::date) AND (data_compra <= '2019-04-30'::date))
32 Planning Time: 453.944 ms
33 Execution Time: 7617.929 ms
```

Imagem 5. Plano da consulta otimizada - Caso 1

Primeiramente é aberto o procedimento de organização dos dados a serem apresentados através do *Quicksort* na ordem decrescente pela soma do total de vendas e código da loja, na linha 1.

Após isso, é aberto o procedimento de agregação pelo código da loja e categoria do produto, na linha 4. Então é aberto outro *Quicksort* pelo código da loja e categoria do produto, na linha 6, e uma junção pelo `id` das tabelas `vendas` e `produto`, na linha 9.

É aberto o procedimento de iteração aninhada, na linha 11, e começa o *scan* sequencial na loja sob a condição do texto ser igual a uma *substring*, na linha 12. Então é iniciado um *bitmap heap scan* na tabela `produto`, na linha 15, e é re-checado a condição de junção de tabelas.

Após isso, é aberta uma busca utilizando o índice no código da loja, na linha 18. É aberto então um *Hash*, na linha 20, e iniciado um *scan* da *subquery* na tabela `vendas`, na linha 22. É feito uma agregação no `id` do produto da tabela `transacao`, na linha 23, e iniciado um *bitmap heap scan* na tabela `transacao`, na linha 25. Foi re-checada a condição (data de compra dentro de um intervalo) e filtrado pelo status como entregue.

Por fim, é iniciado um *scan* usando o índice na data da compra, sob a condição do intervalo, na linha 30. Após finalizado esse *scan*, os procedimentos são retornados para finalizar a filtragem, o *heap scan*, a agregação, a *subquery*, o *Hash*, o *scan* com indice, o *bitmap heap scan* no `produto`, a filtragem pela *substring*, o *scan* sequencial, a iteração aninhada, a junção *Hash*, o *Quicksort*, a agregação e o outro *Quicksort*.

Podemos ver uma melhora no desempenho em relação a consulta anterior, por parte dos *Bitmap Index Scan* e dos *Bitmap Heap scan,* nas linhas 30, 25, 18 e 15, por utilizarem métodos inteligentes de busca.

5.2.4 Plano da Consulta Otimizado - Caso 2

➢ Nome da loja: “An%”. Data inicial: 2017-12-30. Data final: 2019-10-15.

	QUERY PLAN text
1	Sort (cost=162968.30..162970.55 rows=900 width=65) (actual time=3910.538..3910.580 rows=874 loops=1)
2	Sort Key: loja.cod, (sum((sum(transacao.quantidade)))) DESC
3	Sort Method: quicksort Memory: 129kB
4	-> GroupAggregate (cost=162880.50..162924.14 rows=900 width=65) (actual time=3906.787..3909.446 rows=874 loops=1)
5	Group Key: loja.cod, produto.categoria
6	-> Sort (cost=162880.50..162888.60 rows=3239 width=41) (actual time=3906.651..3907.664 rows=3586 loops=1)
7	Sort Key: loja.cod, produto.categoria
8	Sort Method: quicksort Memory: 526kB
9	-> Hash Join (cost=8474.16..162691.64 rows=3239 width=41) (actual time=50.802..3889.374 rows=3586 loops=1)
10	Hash Cond: (transacao.produto_id = produto.id)
11	-> GroupAggregate (cost=0.43..149732.48 rows=323858 width=12) (actual time=25.206..3719.244 rows=363482 loops=1)
12	Group Key: transacao.produto_id
13	-> Index Scan using transacao_produto_id_idx on transacao (cost=0.43..141738.61 rows=951057 width=8) (actual time=25.172..3467.583 rows=957704 loops=1)
14	Filter: ((data_compra >= '2017-12-30'::date) AND (data_compra <= '2019-10-15'::date) AND ((status)::text = 'Entregue'::text))
15	Rows Removed by Filter: 1042296
16	-> Hash (cost=8423.73..8423.73 rows=4000 width=37) (actual time=23.165..23.167 rows=3961 loops=1)
17	Buckets: 4096 Batches: 1 Memory Usage: 348kB
18	-> Nested Loop (cost=26.79..8423.73 rows=4000 width=37) (actual time=0.433..21.496 rows=3961 loops=1)
19	-> Seq Scan on loja (cost=0.00..6.75 rows=3 width=21) (actual time=0.019..0.068 rows=3 loops=1)
20	Filter: ((nome)::text ~~ 'An%'::text)
21	Rows Removed by Filter: 297
22	-> Bitmap Heap Scan on produto (cost=26.79..2792.28 rows=1338 width=20) (actual time=0.317..6.886 rows=1320 loops=3)
23	Recheck Cond: (loja_cod = loja.cod)
24	Heap Blocks: exact=3545
25	-> Bitmap Index Scan on produto_loja_cod_idx (cost=0.00..26.46 rows=1338 width=0) (actual time=0.173..0.173 rows=1320 loops=3)
26	Index Cond: (loja_cod = loja.cod)
27	Planning Time: 0.903 ms
28	JIT:
29	Functions: 27
30	Options: Inlining false, Optimization false, Expressions true, Deforming true
31	Timing: Generation 2.458 ms, Inlining 0.000 ms, Optimization 1.917 ms, Emission 22.780 ms, Total 27.154 ms
32	Execution Time: 3913.219 ms

Imagem 6. Plano da consulta otimizada - Caso 2

Primeiramente, foi feito um escaneamento *Bitmap Index* usando o índice da `loja_cod` da tabela `produto`, na linha 25, seguido de um escaneamento *Bitmap Heap* na tabela `produto`, na linha 22, ao mesmo tempo que uma busca sequencial na tabela `loja`, na linha 19, feita após a remoção de tuplas pelo filtro de nome. Posteriormente, foi feita uma iteração aninhada, na linha 18, que então foi colocada numa estrutura de mapeamento *Hash*, na linha 16. Ao mesmo tempo que o mapeamento *Hash* foi feita, tivemos um agrupamento pelo `produto_id` da tabela `transacao`, na linha 11, que, por sua vez, foi feita a partir dos resultados de um escaneamento por índice, na linha 13, usando o índice do `produto_id`, após a remoção de tuplas pelo filtro das datas pedidas e do *status* da transação.

Isso tudo é seguido de uma junção *Hash* em cima da condição do *produto_id* da tabela `transacao` ser igual ao `id` da tabela `produto`, na linha 9. Após isso, foi feita uma ordenação pelo código da loja e a categoria do produto pelo método *Quicksort*, na linha 6. Posteriormente, foi feito um agrupamento pelo código da loja e a categoria do produto, na linha 4. Para finalizar, foi feito uma ordenação pelo código da loja e a soma do total de vendas na ordem decrescente pelo método *Quicksort*, na linha 1.

Nessa consulta, foi obtido uma melhora no tempo de busca ao modificar a estrutura da consulta, trocando as posições das verificações dos filtros, dessa forma conseguimos um melhor aproveitamento dos recursos e dos índices.

5.2.5 Tabelas de Comparação do Tempo de Execução

➢ Nome da loja: “An%”. Data inicial: 2017-12-30. Data final: 2019-10-15.

Tempo de execução	Consulta Inicial	Consulta Otimizada	Diferença (%)
1ª execução	9secs 472msecs	1min 56sec	...[1]
2ª execução	8secs 969msecs	1min 45secs	...
3ª execução	9secs 150msecs	2min 3secss	...
4ª execução	11secs 593msecs	1min 59secs	...
5ª execução	9secs 600msecs	1min 56secs	...
Média	9secs 756msecs	1 min 55 secs 800 msecs	...

Tabela 8. Apresentação de resultados de tempo de execução.

➢ Nome da loja: “An%”. Data inicial: 2019-04-01. Data final: 2019-04-30.

Tempo de execução	Consulta Inicial	Consulta Otimizada	Diferença (%)
1ª execução	7sec 118msec	5sec 548msec	22.05%
2ª execução	19 secs 255msec	5sec 865msec	69.54%
3ª execução	8sec 321msec	6sec 282msec	24.50%
4ª execução	15sec 571msec	6sec 076msec	60.97%
5ª execução	12sec 200msec	6secs 200msec	49.18%
Média	12sec 493 msecs	5secs 994msec	52.02%

Tabela 9. Apresentação de resultados de tempo de execução. Fórmula para cálculo do % = ((Vot - Vini)/Vini × 100)

[1] “...” representa que o dado é desconhecido

➢ Nome da loja: “An%”. Data inicial: 2019-04-15. Data final: 2019-10-30.

Tempo de execução	Consulta Inicial	Consulta Otimizada	Diferença (%)
1ª execução	8sec 065msec	6sec 324msec	21.58%
2ª execução	17secs 983msecs	5sec 761msec	67.96%
3ª execução	7secs 553msecs	5sec 824msec	22.89%
4ª execução	9secs 060msecs	6sec 043msec	33.30%
5ª execução	11secs 800msecs	6sec 624msec	43.86%
Média	10 sec 892msec	6sec 115msec	43.85%

Tabela 10. Apresentação de resultados de tempo de execução. Fórmula para cálculo do % = ((Vot - Vini)/Vini × 100)

5.2.6 Considerações finais da consulta

Quando o período inserido pelo usuário é muito grande os índices criados aumentam o tempo de execução exponencialmente, a única solução é não usar os índices na consulta nestas situações, isso poderá ser feito posteriormente no próprio sistema com a criação de uma condição que verifique o tamanho do período de tempo informado e decida se os índices serão usados ou não.

6. MUDANÇAS DA FASE INTERMEDIÁRIA

Levando em consideração as correções e idéias recebidas durante a correção da última fase, fizemos as seguintes alterações:

- Adicionamos descrições e comentários sobre as tabelas do tópico 4.
- Colocamos palavras em inglês em itálico e mudamos a fonte das palavras que se referem a objetos do banco de dados para melhorar a compreensão do texto.
- Adicionamos a qual linha se referia a descrição de cada plano de consulta
- Adicionamos a explicação de cada plano de consulta
- Alteramos a disposição dos itens das tabelas presentes no relatório
- Colocamos legendas nas imagens

7. PROGRAMAÇÃO COM O BANCO DE DADOS

Nessa fase nós paramentamos as variáveis de nossas duas consultas utilizando o comando CREATE FUNCTION do postgresql, os resultados finais foram os seguintes:

7.1 *STORED PROCEDURE* DA CONSULTA ABSOLUTA:

```
CREATE FUNCTION produtos_em_um_periodo(varchar, int, int)
RETURNS TABLE (nome_produto varchar, nomecategoria varchar, preco double precision, quantidade_media numeric, quantidade_total bigint) AS $$
DECLARE
    nome_da_loja ALIAS FOR $1;
    qtd_inicial ALIAS FOR $2;
    qtd_final ALIAS FOR $3;
BEGIN
    RETURN QUERY SELECT produto.nome as nome_produto, categoria, preco, ROUND(AVG(quantidade), 2) AS media_por_lote, SUM(quantidade) AS quantidade_total FROM transacao, produto, loja
WHERE produto_id = id AND cod = loja_cod AND loja.nome = nome_da_loja
GROUP BY id
HAVING SUM(quantidade) > qtd_inicial AND SUM(quantidade) <= qtd_final;
END;
$$ LANGUAGE plpgsql;
```

Chamada da função:

```
SELECT * from produtos_em_um_periodo(<NOME_LOJA>, <QUANTIDADE_INICIAL>, <QUANTIDADE_FINAL>);
```

Devido ao tipo *varchar* do parâmetro, a entrada referente ao nome deve estar entre aspas simples, como tanto a quantidade final como a inicial são inteiros, isto não é necessário.

7.2 *STORED PROCEDURE* DA CONSULTA RELATIVA:

```
CREATE FUNCTION vendas_por_categoria(character varying(50), date, date)
RETURNS TABLE (cod_loja int, nome_loja character varying(50), nome_categoria character varying(50), sum numeric) AS $$
DECLARE
    nome_da_loja ALIAS FOR $1;
    data_inicial ALIAS FOR $2;
    data_final ALIAS FOR $3;
BEGIN
    RETURN QUERY SELECT loja.cod, loja.nome, categoria, sum(vendas.total) FROM
    (SELECT produto_id as id, sum(quantidade) as total FROM transacao WHERE
     transacao.data_compra BETWEEN data_inicial AND data_final AND status = 'Entregue' GROUP BY produto_id)
    as vendas, loja, produto
    WHERE vendas.id = produto.id AND loja.cod = loja_cod AND loja.nome LIKE nome_da_loja
    GROUP BY loja.cod, produto.categoria
    ORDER BY loja.cod, sum(vendas.total) DESC;
END;
$$ LANGUAGE plpgsql;
```

Chamada da função:

```
SELECT * FROM vendas_por_categoria(<NOME_LOJA>, <DATA_INICIAL>, <DATA_FINAL>)
```

Assim como na última consulta o parâmetro referente ao nome da loja também deve ser inserido entre aspas simples, em relação às datas iniciais e finais, ambas devem ser inseridas no formato *‘ano-mês-dia’*.

8. CONTROLE DE ACESSO DE USUÁRIOS

8.1 GRAFO DE CONCESSÕES DE USUÁRIOS E TABELA DE PERFIS DE USUÁRIOS, SEU PAPEL E PRIVILÉGIOS

Na imagem 7 e na tabela 11 abaixo temos, respectivamente, o grafo de concessões de usuários e a tabela dos perfis de usuários, seu papel e privilégios.

Criamos um novo usuário que colocamos como dono do banco de dados do projeto chamado `db_owner`. O `db_owner` por sua vez criou os usuários `sem_cadastro`, `dono_loja`, `moderador` e `comprador`, e também concedeu os privilégios específicos que cada um deve ter, que pode ser encontrado na tabela 11.

O usuário `dono_loja` representa o usuário que cria uma conta comercial no nosso sistema, sendo que este é responsável pela criação e concessão dos privilégios do usuário `funcionario_loja`. Este último, por sua vez, representa o usuário utilizado pelo funcionário de uma loja do *e-commerce,* que também tem os mesmos privilégios que o `dono_loja` com exceção dos privilégios de inserir e deletar na tabela `loja`.

O usuário `sem_cadastro` representa o usuário do sistema que o acessa mas ainda não possui nenhum tipo de cadastro. Enquanto, o usuário comprador representa o usuário que possui um cadastro como conta pessoal, e faz compras no *e-commerce*.

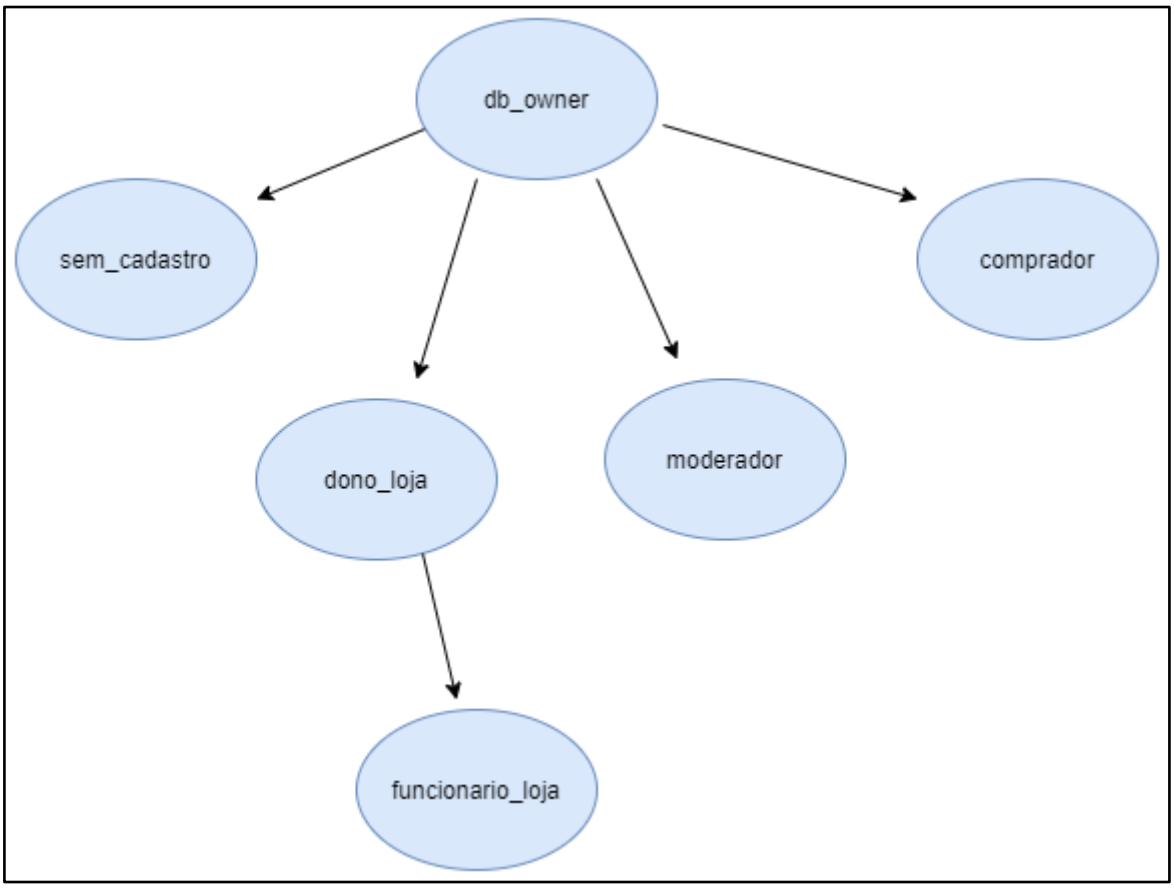

Imagem 7. Grafo de concessões

	moderador	dono_loja	funcionario_loja	comprador	sem _cadastro
comprador	S, D	S	S	S, U, D	I
transacao	S, U, D	S, U, D	S, U, D	S, I	Sem acesso
loja	S, D	S, U, I, D	S, U	S	S, I
categoria	S, U, I, D	S	S	S	S
produto	S, D	S, U, I, D	S, U, I, D	S	S

Tabela 11. Perfis de usuários, seu papel e privilégios (SELECT (S), INSERT (I), DELETE (D) e UPDATE (U))

8.2 PERMISSÕES DE CADA USUÁRIO EM UM PROJETO APLICADO

No primeiro contato com a nossa aplicação, o usuário não estará cadastrado e se encontrará na página inicial do site, onde ele já poderá acessar informações sobre as categorias e os produtos que estão disponíveis em nosso banco de dados. Porém ainda não poderá realizar nenhuma ação mais significativa, nisso ele terá que se para a tela de cadastro.

Na tela de cadastro, o usuário terá a opção de cadastrar sua conta como conta pessoal, para, principalmente, realizar a compra dos produtos presentes em nosso *e-commerce*, entre outras ações, ou como conta comercial, no qual será feito o cadastro da sua loja. Após cadastrar sua loja, o usuário poderá adicionar novos produtos, onde irá selecionar a qual categoria estes pertencem, sendo que a categoria já é pré-definida em nossa aplicação.

A conta de uma loja não precisa ser gerenciada por uma pessoa só, caso o dono já tenha uma loja física, por exemplo, ele poderá adicionar outras contas como funcionários da loja, isso os dá permissão para gerenciar os produtos e as transações da loja, mas não os dá o direito de modificar informações da loja em si, ao contrário do dono.

O moderador é um usuário especial com o propósito de cuidar do ambiente de nosso *e-commerce*, ele possui permissões especiais que permitem que ele altere dados como as categorias do site e remova contas tanto de usuários quanto de lojas que quebrem os termos de uso.

Em relação às nossas consultas, ambas retornam dados relevantes às transações de determinadas lojas, portanto poderão ser acessadas somente pelos usuários `dono_loja` e `funcionario_loja`. Haverá uma tela especial chamada de relatório de vendas, nela o usuário poderá escolher quais informações ele deseja visualizar, e será por esta tela que estarão disponíveis nossas duas consultas.

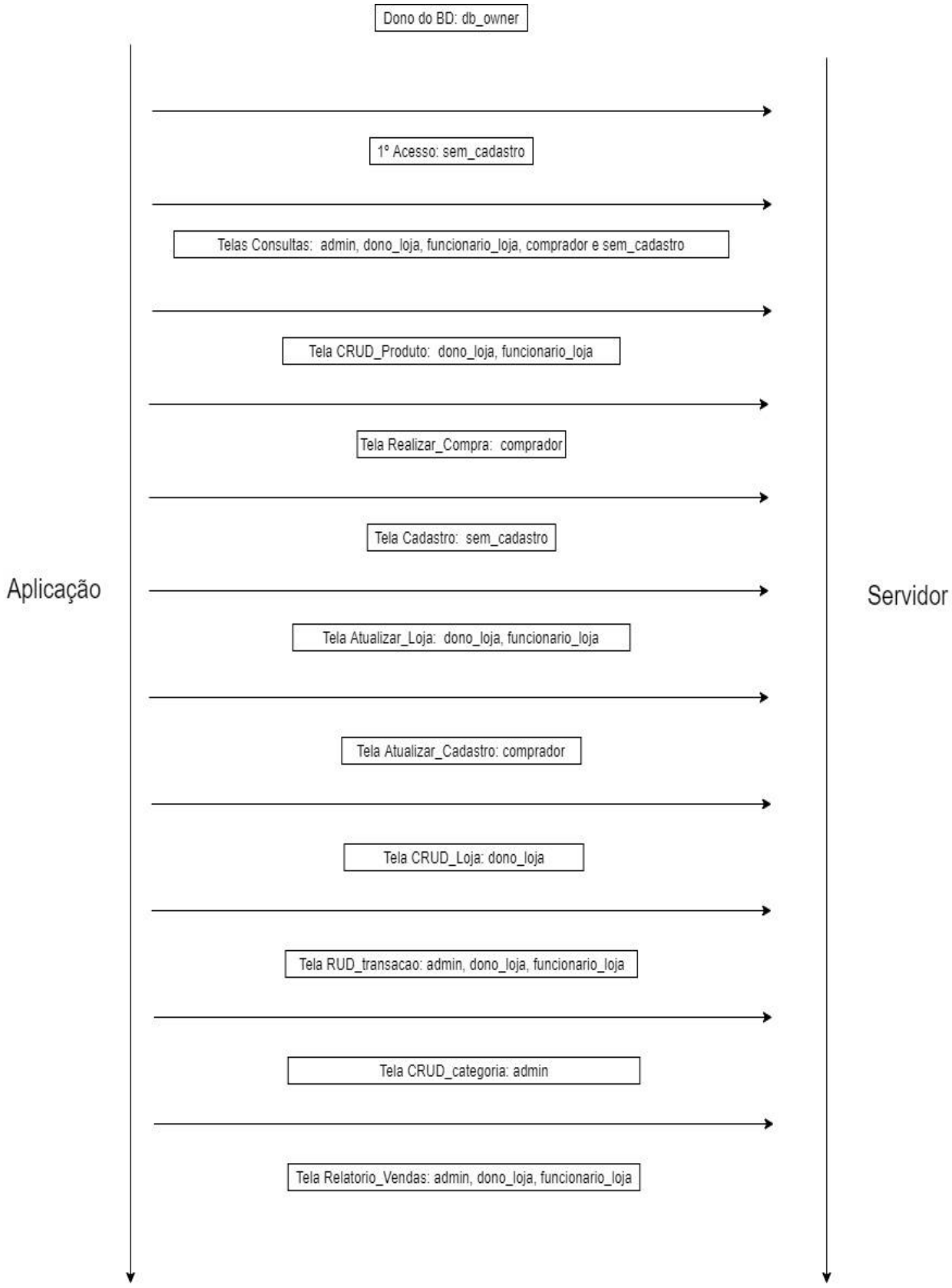

Imagem 8. Conexões usando perfis de usuários

8.3 SCRIPT DAS CONCESSÕES

-- Criamos o usuário chamado `db_owner` com o privilégio de criar outros usuários.

```
CREATE ROLE db_owner WITH CREATEROLE;
```

-- Concedemos todos os privilégios de cada tabela do nosso projeto para o `db_owner` (a partir do usuário `postgres`)

```
GRANT ALL PRIVILEGES ON comprador TO db_owner WITH GRANT
OPTION;
GRANT ALL PRIVILEGES ON transacao TO db_owner WITH GRANT
OPTION;
GRANT ALL PRIVILEGES ON loja TO db_owner WITH GRANT OPTION;
GRANT ALL PRIVILEGES ON categoria TO db_owner WITH GRANT
OPTION;
GRANT ALL PRIVILEGES ON produto TO db_owner WITH GRANT
OPTION;
```

-- Então alteramos o dono do banco de dados para `db_owner`

```
ALTER DATABASE <nome_bd> OWNER TO db_owner;
```

-- Colocamos o `db_owner` para atuar como usuário atual

```
SET ROLE db_owner;
```

-- MODERADOR:

-- Criamos o usuário `moderador`

```
CREATE ROLE moderador WITH NOINHERIT;
```

-- Concedemos os privilégios que o `moderador` possui de cada tabela

```
GRANT SELECT, DELETE ON TABLE comprador TO moderador
GRANT SELECT, UPDATE, DELETE ON TABLE transacao TO
moderador;
GRANT SELECT, DELETE ON TABLE loja TO moderador;
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE categoria
TO moderador;
GRANT SELECT, DELETE ON TABLE produto TO moderador;
```

-- COMPRADOR:

-- Criamos o usuário `comprador`

```
CREATE ROLE comprador WITH NOINHERIT LOGIN PASSWORD
'senha1';
```

-- Concedemos os privilégios que o `comprador` possui de cada tabela

```
GRANT SELECT, UPDATE, DELETE ON TABLE comprador TO
comprador;
GRANT SELECT, INSERT ON TABLE transacao TO comprador;
GRANT SELECT ON TABLE loja TO comprador;
GRANT SELECT ON TABLE categoria TO comprador;
GRANT SELECT ON TABLE produto TO comprador;
```

-- SEM_CADASTRO:

-- Criamos o usuário `sem_cadastro`

```
CREATE ROLE sem_cadastro WITH NOINHERIT;
```

-- Concedemos os privilégios que o usuário `sem_cadastro` possui de cada tabela

```
GRANT INSERT ON comprador TO sem_cadastro;
GRANT SELECT, INSERT ON loja TO sem_cadastro;
GRANT SELECT ON categoria TO sem_cadastro;
GRANT SELECT ON produto TO sem_cadastro;
```

-- DONO_LOJA

-- Criamos o usuário `dono_loja` com o privilégio de criar outros usuários.

```
CREATE ROLE dono_loja WITH CREATEROLE
```

-- Concedemos os privilégios que o usuário `dono_loja` possui de cada tabela

```
GRANT SELECT ON comprador TO dono_loja WITH GRANT OPTION;
GRANT SELECT, DELETE, UPDATE ON transacao TO dono_loja
WITH GRANT OPTION;
GRANT SELECT, UPDATE, INSERT, DELETE ON loja TO dono_loja
WITH GRANT OPTION;
GRANT SELECT ON categoria TO dono_loja WITH GRANT OPTION;
GRANT SELECT, UPDATE, INSERT, DELETE ON produto TO
dono_loja WITH GRANT OPTION;
```

-- Colocamos o `dono_loja` para atuar como usuário atual

```
SET ROLE dono_loja;
```

-- FUNCIONARIO_LOJA

-- Criamos o usuário `funcionario_loja`

```
CREATE ROLE funcionario_loja WITH NOINHERIT;
```

-- Concedemos os privilégios que o usuário `funcionario_loja` possui de cada tabela

```
GRANT SELECT ON comprador TO funcionario_loja;
GRANT SELECT, DELETE, UPDATE ON transacao TO
funcionario_loja;
GRANT SELECT, UPDATE ON loja TO funcionario_loja;
GRANT SELECT ON categoria TO funcionario_loja;
GRANT SELECT, UPDATE, INSERT, DELETE ON produto TO
funcionario_loja;
```

9. CONSIDERAÇÕES FINAIS

O projeto foi diferente de outras experiências que tivemos com banco de dados, pois foi abordado conceitos importantes como acesso de usuário, onde teve que ser bem planejado para não violar nenhuma restrição de acesso.

Também exploramos a parametrização das nossas consultas, transformando-as em *stored procedures* para facilitar o acesso. Além disso, trabalhamos com os planos das consultas para estudarmos formas de melhorar seus desempenhos.

Com esse trabalho, conseguimos colocar em prática muitos dos ensinamentos vistos em aula assim tornando mais palpável e melhorando a compreensão da matéria.

Em geral não tivemos muita dificuldade durante o desenvolvimento, embora tenhamos demorado um bom tempo discutindo como seria o esquema final do banco de dados, foi um projeto sem muitos tropeços.

Talvez a nossa principal barreira tenha sido durante a fase de otimização de consultas, especificamente em nossa consulta relativa, mesmo conseguindo resultados em algumas situações eles em geral não atingiram nossa meta desejada.

Nosso projeto teve algumas alterações durante seu desenvolvimento para promover melhorias em seu conteúdo. Em relação ao diagrama do banco de dados, mudamos a tabela `categoria` que antes possuía duas colunas sendo elas `cod` (que era a chave primária) representando o código de cada tupla da tabela, e `descricao`, onde havia o nome de cada categoria, e agora, ela possui apenas uma coluna chamada `descricao`, e, consequentemente, a coluna na tabela `produto`, que é chave estrangeira que se refere a tabela `categoria`, `categoria_cod` mudou para `categoria`. Além disso, também mudamos nossas consultas. Antes, nossa consulta absoluta era “Quantidade de produtos vendidos em um determinado período agrupados por sua categoria.”, e agora é “Informações sobre os produtos, de uma certa loja, cuja quantidade total de unidades vendidas está dentro de um dado intervalo”. E a nossa consulta relativa era “Informações sobre os produtos, de uma loja (com o nome fornecido como parâmetro), cuja quantidade total de unidades vendidas está dentro de um dado intervalo., e mudamos para “Quantidade de produtos entregues de uma certa loja (usando `LIKE`) em um determinado período agrupados por sua categoria ordenado descendente pela quantidade.”. Os enunciados das consultas eram relativamente simples, então a mudança se deu ao fato de visarmos enunciados mais elaborados, assim sendo mais adequado para o desenvolvimento do projeto da disciplina.