UNIVERSIDADE FEDERAL DE SÃO CARLOS - CAMPUS SOROCABA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Sistemas de Bancos de Dados

Profa. Dra. Sahudy Montenegro González

Tema 3: Avaliações de produtos por consumidores

GRUPO 1
GABRIEL VIANA TEIXEIRA - 759465
GUILHERME PEREIRA FANTINI - 759468
TALES BALTAR LOPES DA SILVA - 759535

10 de Janeiro de 2021
Entrega Final

SUMÁRIO

1. DEFINIÇÃO DO PROBLEMA

Nos últimos anos, a tecnologia impactou o cotidiano da sociedade ao trazer soluções a diversos problemas. O *e-commerce* é uma atividade cada vez mais presente na nossa rotina que facilita a compra e venda de produtos ou serviços.

Mediante o panorama, este trabalho pretende desenvolver um componente do sistema de banco de dados de um *e-commerce* com o objetivo de realizar as funcionalidades relacionadas a avaliações de produtos por consumidores.

1.1. COMPORTAMENTO DO MUNDO REAL

Em um *e-commerce*, um consumidor pode realizar diversas compras de diversos produtos. Para cada compra, o consumidor pode avaliar os produtos adquiridos.

O consumidor é identificado pelo CPF e dispõe de nome, data de nascimento, e-mail, senha e cidade. O produto é identificado por id próprio e dispõe de nome, preço, descrição e categoria. A compra é identificada pelo consumidor, produto e a data de compra e dispõe de quantidade, identificador se a compra foi entregue, número de estrelas avaliada pelo consumidor e comentário.

A Figura 1 apresenta o modelo relacional elaborado para o panorama exposto.

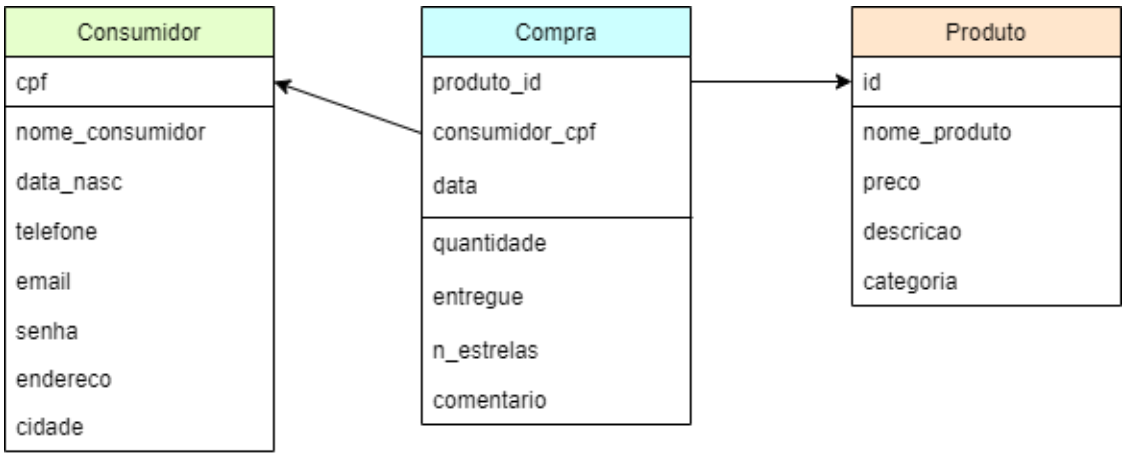

Figura 1 - Diagrama do modelo relacional para o minimundo.

2. DESCRIÇÃO DAS CONSULTAS

Para cada consulta, é utilizado o esquema apresentado na Figura 1. Dessa forma, temos as seguintes tabelas:

- **`consumidor(cpf, nome_consumidor, data_nasc, email, senha, cidade);`**
- **`produto(id, nome_produto, preco, descricao, categoria);`**
- **`compra(produto_id, consumidor_cpf, data, quantidade, entregue, n_estrelas, comentario).`**

1 - Mostrar a média de notas de produtos de uma determinada categoria dentro de uma faixa de preços.

Campos de visualização do resultado: **`nome_produto, preco, descricao, categoria, media.`**

Campos de busca: **`categoria (relativa), preco (absoluta)`**.

Operadores das condições: **`categoria (ILIKE), preco (<=, >=)`**.

2 - Recuperar todos os consumidores que possuem um histórico de compras similar a um dado consumidor e que morem na mesma cidade, ordenado por maior nível de similaridade.

Campos de visualização: **`nome_consumidor, cpf, nome_consumidor', cpf', cidade, qtd_produtos_similares)`**.

Campos de busca: **`cidade (relativa), cpf_consumidor(absoluta)`**.

Operadores de condições: **`cidade (ILIKE), cpf_consumidor(==)`**.

Para esclarecer o funcionamento da consulta 2, foi elaborado um exemplo. Considerando os estados das tabelas Consumidor, Produto e Compra estão representados pelas Tabelas 1, 2 e 3, respectivamente.

Ao realizar a consulta 2, com o objetivo de recuperar os consumidores com histórico de compras similares ao consumidor de CPF 12345678911, o resultado da consulta será semelhante a Tabela 4.

3. ESPECIFICAÇÃO DAS CONSULTAS EM SQL

1 - Mostrar a média de notas de produtos de uma determinada categoria dentro de uma faixa de preços.

```
SELECT nome_produto, preco, descricao, categoria, avg(n_estrelas) AS
media
FROM produto
JOIN compra
    ON produto_id = id
WHERE categoria ILIKE <nome_categoria>
    AND preco BETWEEN <preco_min> AND <preco_max>
    AND entregue = true
GROUP BY id;
```

Código 1 - Primeira consulta.

2 - Recuperar todos os consumidores que possuem um histórico de compras similar a um dado consumidor e que morem na mesma cidade, ordenado por maior nível de similaridade.

```
SELECT nome_consumidor,
         cpf,
         (SELECT nome_consumidor FROM consumidor WHERE cpf = <cpf>),
         (SELECT cpf FROM consumidor WHERE cpf = <cpf>),
         cidade,
         COUNT(cpf) AS qtd_produtos_similares
FROM
      (SELECT DISTINCT nome_consumidor, cpf, produto_id, cidade
      FROM consumidor JOIN compra
      ON cpf = consumidor_cpf
      WHERE cpf <> <cpf> AND cidade ILIKE (SELECT cidade FROM consumidor
WHERE cpf=<cpf>)
      AND (produto_id) IN
            (SELECT produto_id
             FROM consumidor
             JOIN compra
             ON cpf=consumidor_cpf
             WHERE cpf = <cpf>)) AS resultado
GROUP BY nome_consumidor, cpf, cidade
ORDER BY qtd_produtos_similares DESC;
```

Código 2 - Segunda consulta.

4. POPULANDO O BD

Para consolidar as consultas planejadas, o banco de dados foi populado com *script* próprio utilizando a linguagem Python (versão 3.8) com as bibliotecas `psycopg2` e `pandas`. A tabela seguinte mostra o número de registros e o tamanho de cada registro.

Nome da tabela	Quantidade de registros	Tamanho de cada registro
Consumidor	500000	113B
Produto	500000	73B
Compra	500000	47B

Tabela 1 - Informações sobre a população do BD.

5. ESPECIFICAÇÃO DAS CONSULTAS OTIMIZADAS

Após a definição dos códigos em SQL das consultas definidas, foi realizada a otimização das duas consultas. Inicialmente, a abordagem preferencial era a criação de índices, contudo para a primeira consulta, foi necessária a mudança do esquema relacional.

5.1. PRIMEIRA CONSULTA

Plano de execução

No plano de execução, inicialmente ocorre uma busca sequencial paralela na tabela `Produto`, aplicando um filtro no nome da categoria e no intervalo de preço definido. Como a quantidade de tuplas a conferir possui muitos dados, a tarefa de buscar sequencialmente é dividida em *hash buckets* com execução em paralelo. Já na tabela `Compra`, também é realizada uma filtragem utilizando busca sequencial paralela, para satisfazer a condição de *`entregue=true`*.

Após a aplicação dos filtros em cada tabela separadamente, é necessário realizar um *join* entre as duas tabelas, as opções disponíveis são: *nested loop join* (um *scan* sequencial na relação à direita para cada entrada da esquerda e que poderia ser otimizado, caso houvesse índices na relação à direita), um *merge join* (um *merge* das tabelas ordenadas, se as tuplas

fossem ordenadas ficaria bastante eficiente) e, por último, *hash join* (junção em que a tabela à direita é lida e guardada numa tabela *hash* e, para cada entrada na esquerda, é feita uma busca na tabela *hash*).

A opção escolhida foi o *hash join* paralelo, principalmente devido as tabelas(`Produto` e `Compra`) não cumprirem os requisitos necessários para que os outros *joins* sejam realizados de forma otimizada. Por último, como um produto pode ser comprado várias vezes, é necessário fazer um agrupamento pelo id dos produtos, portanto primeiro é feito uma ordenação das tuplas no `produto_id`, dividido em 2 *workers*, pois a quantidade de tuplas para a operação é suficientemente grande e não cabe na memória, então após ser feito em cada *worker* separadamente, o agrupamento parcial é realizado e, por fim, há um *merge* que junta os 2 *workers* ordenados, finalizando o agrupamento e a consulta.

```
1  Finalize GroupAggregate  (cost=18906.16..22231.70 rows=27680 width=73) (actual time=1758.839..1840.414 rows=22018 loops=1)
2  Group Key: produto.id
3  -> Gather Merge  (cost=18906.16..21770.37 rows=23066 width=97) (actual time=1758.827..1807.690 rows=25730 loops=1)
4        Workers Planned: 2
5        Workers Launched: 2
6        -> Partial GroupAggregate  (cost=17906.13..18107.96 rows=11533 width=97) (actual time=1544.566..1561.398 rows=8577 loops=3)
7              Group Key: produto.id
8              -> Sort  (cost=17906.13..17934.97 rows=11533 width=73) (actual time=1544.547..1546.814 rows=9291 loops=3)
9                    Sort Key: produto.id
10                   Sort Method: quicksort  Memory: 1675kB
11                   Worker 0:  Sort Method: quicksort  Memory: 1663kB
12                   Worker 1:  Sort Method: quicksort  Memory: 1790kB
13                   -> Parallel Hash Join  (cost=10409.58..17128.03 rows=11533 width=73) (actual time=1204.952..1528.092 rows=9291 loops=3)
14                         Hash Cond: (compra.produto_id = produto.id)
15                         -> Parallel Seq Scan on compra  (cost=0.00..6446.36 rows=103654 width=12) (actual time=43.436..304.649 rows=83212 loops=3)
16                               Filter: entregue
17                               Rows Removed by Filter: 83456
18                         -> Parallel Hash  (cost=10119.83..10119.83 rows=23180 width=65) (actual time=1160.988..1160.988 rows=18567 loops=3)
19                               Buckets: 65536  Batches: 1  Memory Usage: 6560kB
20                               -> Parallel Seq Scan on produto  (cost=0.00..10119.83 rows=23180 width=65) (actual time=0.096..1131.683 rows=18567 loops=3)
21                                     Filter: (((categoria)::text ~~* 'Vestuário'::text) AND (preco >= '50'::double precision) AND (preco <= '300'::double precision))
22                                     Rows Removed by Filter: 148100
```

Figura 2 - Plano de Execução Inicial da Consulta 1

Otimização

Para essa primeira consulta, foi necessária a mudança do esquema relacional, para isso, duas novas colunas foram adicionadas à tabela `Produto`. Isso foi feito para que não seja necessário realizar o cálculo da média a cada consulta.

```
ALTER TABLE produto ADD media real;
```

```
ALTER TABLE produto ADD vendidos int;
```

Código 3 - Alteração da tabela `Produto`.

O código abaixo, serviu para atualizar a coluna `media`, que apresenta a média da nota (`n_estrelas`) que aquele determinado produto recebeu. Essa coluna foi criada na tabela `Produto`.

```
UPDATE produto
SET media = compra.media_produto
FROM (select produto_id, avg(n_estrelas) as media_produto FROM compra GROUP
BY produto_id) as compra
WHERE compra.produto_id = id;
```

Código 4 - Atualização da coluna `media`.

O código abaixo, serviu para atualizar a coluna `vendidos`, atributo que guarda a quantidade de vezes que o determinado produto foi comprado. Essa coluna foi criada na tabela `Produto`.

```
UPDATE produto
SET vendidos = compra.num_vendidos
FROM (SELECT produto_id, count(*) as num_vendidos FROM compra GROUP BY
produto_id) as compra
WHERE compra.produto_id=produto.id
```

Código 5 - Atualização da coluna `num_vendidos`.

Além disso, foi criado um índice sob a coluna `preco`, pois no plano de execução foi observada uma varredura sequencial envolvendo o preço, ou seja, é custoso à medida que a quantidade de tuplas aumenta. Dependendo dos argumentos da consulta, a faixa de preço pode englobar poucas tuplas, então é preferível que essa varredura utilize o índice proposto em vez de passar por todas as linhas da tabela `produto`.

```
CREATE INDEX preco_index ON produto(preco)
```

Código 6 - Criação do índice `preco_index`.

Consulta Resultante

```
SELECT nome_produto, preco, descricao, categoria, media
FROM produto
WHERE
categoria ILIKE <categoria>
AND preco BETWEEN <preco_min> AND <preco_max>
AND media IS NOT NULL
```

Código 7 - Consulta 1 otimizada.

Plano de execução otimizado

Como foi mencionado anteriormente, o número de tuplas retornadas depende do intervalo de preços passado como argumento. Assim, o otimizador pode utilizar o índice em caso de poucas tuplas ou fazer um *scan* sequencial para a situação com muitas tuplas.

No caso médio, foi utilizado um *Bitmap Index Scan*, que consiste em construir um *bitmap* com as localizações das tuplas e resgatar apenas as páginas que contém as tuplas necessárias. Como são muitas tuplas para caber em memória, o primeiro *Bitmap Index Scan* salva apenas as páginas que contém as tuplas e não as tuplas em si. Dessa forma, ao realizar o *Bitmap Heap Scan* é necessário, além de filtrar a `categoria` e a `media`, também é necessário realizar o *recheck* das condições da coluna `preco`.

	QUERY PLAN text
1	Bitmap Heap Scan on produto (cost=1550.49..13469.53 rows=12738 width=66) (actual time=182.358..994.414 rows=13158 loops=1)
2	Recheck Cond: ((preco >= '50'::double precision) AND (preco <= '200'::double precision))
3	Filter: ((media IS NOT NULL) AND ((categoria)::text ~~* 'Vestuário'::text))
4	Rows Removed by Filter: 61764
5	Heap Blocks: exact=8103
6	-> Bitmap Index Scan on preco_index (cost=0.00..1547.30 rows=73488 width=0) (actual time=166.828..166.829 rows=74922 loops=1)
7	Index Cond: ((preco >= '50'::double precision) AND (preco <= '200'::double precision))

Figura 3 - Plano de Execução da consulta 1 otimizada.

5.2. SEGUNDA CONSULTA

Plano de execução

No plano de execução, primeiro é feito um *index scan* na tabela `Consumidor` procurando o consumidor que corresponde ao cpf passado como argumento para o consumidor base e, em

seguida, é realizado um *scan* paralelo sequencial na tabela `Compra` para encontrar todas as compras que o dado consumidor realizou, uma ordenação também é feita baseada no `produto_id` para facilitar a busca para a próxima sub consulta.

Para encontrar os demais consumidores que moram na mesma cidade, é utilizado um *index scan* no `consumidor_cpf`. As tuplas resultantes são ordenadas por `nome_consumidor, cpf, produto_id e cidade`, as tuplas duplicadas (*distinct*) são desconsideradas e há um agrupamento por `nome_consumidor, cpf, cidade`. No final, há novamente duas buscas na tabela `Consumidor` utilizando o *index scan* para recuperar o nome do consumidor e o cpf, assim uma última ordenação no resultado final baseado na quantidade de produtos similares (`qtd_produto_similares`) em ordem decrescente.

	QUERY PLAN text
1	Sort (cost=10538.45..10538.46 rows=1 width=495) (actual time=4766.680..4766.745 rows=7 loops=1)
2	Sort Key: (count(resultado.cpf)) DESC
3	Sort Method: quicksort Memory: 25kB
4	InitPlan 1 (returns $0)
5	-> Index Scan using consumidor_pkey on consumidor consumidor_2 (cost=0.42..8.44 rows=1 width=7) (actual time=20.167..20.169 rows=1 loops=1)
6	Index Cond: ((cpf)::text = '90093928519'::text)
7	InitPlan 2 (returns $1)
8	-> Index Only Scan using consumidor_pkey on consumidor consumidor_3 (cost=0.42..8.44 rows=1 width=12) (actual time=0.025..0.026 rows=1 loops=1)
9	Index Cond: (cpf = '90093928519'::text)
10	Heap Fetches: 1
11	-> GroupAggregate (cost=10521.54..10521.56 rows=1 width=495) (actual time=4766.650..4766.725 rows=7 loops=1)
12	Group Key: resultado.nome_consumidor, resultado.cpf, resultado.cidade
13	-> Sort (cost=10521.54..10521.54 rows=1 width=29) (actual time=4746.435..4746.501 rows=12 loops=1)
14	Sort Key: resultado.nome_consumidor, resultado.cpf, resultado.cidade
15	Sort Method: quicksort Memory: 25kB
16	-> Subquery Scan on resultado (cost=10521.50..10521.53 rows=1 width=29) (actual time=4722.267..4722.338 rows=12 loops=1)
17	-> Unique (cost=10521.50..10521.52 rows=1 width=33) (actual time=4722.264..4722.333 rows=12 loops=1)
18	InitPlan 3 (returns $2)
19	-> Index Scan using consumidor_pkey on consumidor consumidor_4 (cost=0.42..8.44 rows=1 width=10) (actual time=0.048..0.049 rows=1 loops=1)
20	Index Cond: ((cpf)::text = '90093928519'::text)
21	-> Sort (cost=10513.06..10513.07 rows=1 width=33) (actual time=4722.263..4722.327 rows=13 loops=1)
22	Sort Key: consumidor.nome_consumidor, consumidor.cpf, compra.produto_id, consumidor.cidade
23	Sort Method: quicksort Memory: 26kB
24	-> Nested Loop (cost=10502.70..10513.05 rows=1 width=33) (actual time=4302.258..4722.258 rows=13 loops=1)
25	-> Nested Loop (cost=10502.28..10510.82 rows=4 width=16) (actual time=4212.062..4479.175 rows=23 loops=1)
26	-> Unique (cost=10501.86..10501.87 rows=2 width=4) (actual time=4110.692..4110.778 rows=9 loops=1)
27	-> Sort (cost=10501.86..10501.86 rows=2 width=4) (actual time=4110.691..4110.763 rows=9 loops=1)
28	Sort Key: compra_1.produto_id
29	Sort Method: quicksort Memory: 25kB
30	-> Gather (cost=1000.42..10501.85 rows=2 width=4) (actual time=520.372..4110.721 rows=9 loops=1)
31	Workers Planned: 2
32	Workers Launched: 2

Figura 4.1 - Primeira parte do plano de execução inicial da Consulta 2.

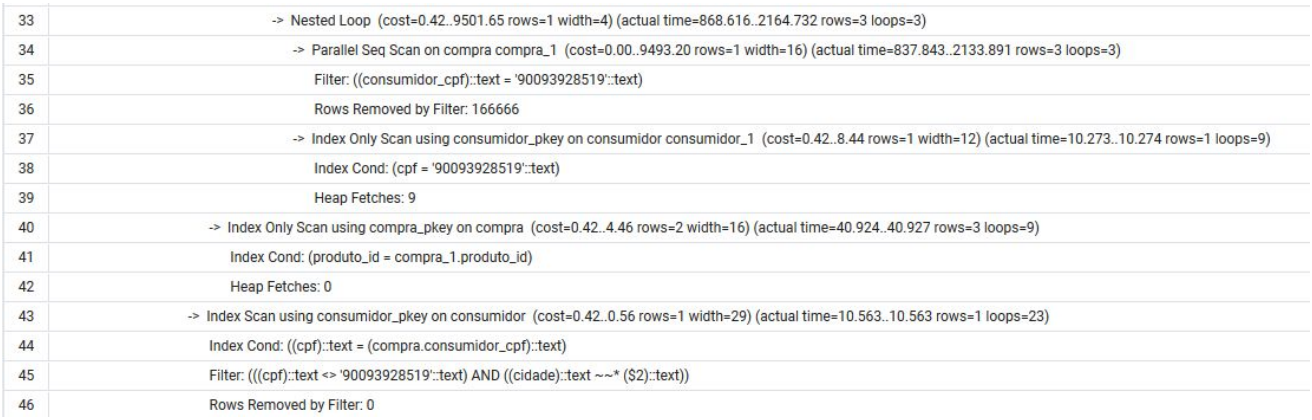

Figura 4.2 - Segunda parte do plano de execução inicial da Consulta 2.

Como a consulta necessita encontrar todos os produtos comprados por cada consumidor e, assim, comparar com a lista de produtos compradas pelo cpf de entrada, então a proposta de otimização é ter um índice composto que envolve o cpf e os produtos comprados, facilitando a contagem, que é a operação mais custosa dessa consulta.

```
CREATE INDEX ON compra(consumidor_cpf, produto_id)
```

Código 8 - Criação do índice na tabela `Compra`.

Além disso, na consulta colocamos todas as condições antes de fazer a junção

```
SELECT.... JOIN compra ON cpf=consumidor_cpf AND cpf=<cpf>
```

Código 9 - Condições das junções.

Anteriormente era utilizado uma cláusula `WHERE` após a junção.

Contudo o principal ganho de performance foi na utilização do índice e não na alteração da consulta.

Consulta resultante

```
SELECT nome_consumidor,
      cpf,
      (SELECT nome_consumidor FROM consumidor WHERE cpf=<cpf>),
      (SELECT cpf FROM consumidor WHERE cpf=<cpf>),
```

```
      cidade,
      COUNT(cpf) AS qtd_produtos_similares
FROM
      (SELECT DISTINCT nome_consumidor, cpf, produto_id, cidade
      FROM consumidor
      JOIN compra
      ON cpf=consumidor_cpf
      AND cpf <> <cpf>
      AND cidade ILIKE (SELECT cidade FROM consumidor WHERE cpf=<cpf>)
      AND (produto_id) IN
      (SELECT produto_id
      FROM consumidor
      JOIN compra
      ON cpf=consumidor_cpf
      AND cpf=<cpf>)) AS resultado
GROUP BY nome_consumidor, cpf, cidade
ORDER BY qtd_produtos_similares DESC;
```

Código 10 - Consulta 2 otimizada.

Plano de execução

O plano de execução da consulta otimizada foi muito similar ao da consulta original, a diferença da otimizada é justamente a utilização do índice (`consumidor_cpf, produto_id`) na hora de encontrar os produtos que cada consumidor comprou, anteriormente eram feitas 2 consultas separadas, uma para o `consumidor_cpf` e outra para o `produto_id`.

Além disso, houve a utilização de um *Materialize*, salvando na memória o resultado intermediário para utilizar em outras operações.

	QUERY PLAN text
1	Sort (cost=49.53..49.53 rows=1 width=495) (actual time=18.788..18.799 rows=7 loops=1)
2	Sort Key: (count(resultado.cpf)) DESC
3	Sort Method: quicksort Memory: 25kB
4	InitPlan 1 (returns $0)
5	-> Index Scan using consumidor_pkey on consumidor consumidor_2 (cost=0.42..8.44 rows=1 width=7) (actual time=0.018..0.018 rows=1 loops=1)
6	Index Cond: ((cpf)::text = '90093928519'::text)
7	InitPlan 2 (returns $1)
8	-> Index Only Scan using consumidor_pkey on consumidor consumidor_3 (cost=0.42..8.44 rows=1 width=12) (actual time=0.017..0.017 rows=1 loops=1)
9	Index Cond: (cpf = '90093928519'::text)
10	Heap Fetches: 1
11	-> GroupAggregate (cost=32.61..32.64 rows=1 width=495) (actual time=18.758..18.776 rows=7 loops=1)
12	Group Key: resultado.nome_consumidor, resultado.cpf, resultado.cidade
13	-> Sort (cost=32.61..32.62 rows=1 width=29) (actual time=18.708..18.719 rows=12 loops=1)
14	Sort Key: resultado.nome_consumidor, resultado.cpf, resultado.cidade
15	Sort Method: quicksort Memory: 25kB
16	-> Subquery Scan on resultado (cost=32.58..32.60 rows=1 width=29) (actual time=18.692..18.707 rows=12 loops=1)
17	-> Unique (cost=32.58..32.59 rows=1 width=33) (actual time=18.690..18.704 rows=12 loops=1)
18	InitPlan 3 (returns $2)
19	-> Index Scan using consumidor_pkey on consumidor consumidor_4 (cost=0.42..8.44 rows=1 width=10) (actual time=0.012..0.012 rows=1 loops=1)
20	Index Cond: ((cpf)::text = '90093928519'::text)
21	-> Sort (cost=24.14..24.14 rows=1 width=33) (actual time=18.689..18.699 rows=13 loops=1)
22	Sort Key: consumidor.nome_consumidor, consumidor.cpf, compra.produto_id, consumidor.cidade
23	Sort Method: quicksort Memory: 26kB
24	-> Nested Loop (cost=13.78..24.13 rows=1 width=33) (actual time=18.309..18.673 rows=13 loops=1)
25	-> Nested Loop (cost=13.36..21.90 rows=4 width=16) (actual time=18.262..18.333 rows=23 loops=1)
26	-> Unique (cost=12.94..12.95 rows=2 width=4) (actual time=18.245..18.258 rows=9 loops=1)
27	-> Sort (cost=12.94..12.94 rows=2 width=4) (actual time=18.243..18.253 rows=9 loops=1)
28	Sort Key: compra_1.produto_id
29	Sort Method: quicksort Memory: 25kB
30	-> Nested Loop (cost=0.84..12.93 rows=2 width=4) (actual time=18.219..18.241 rows=9 loops=1)
31	-> Index Only Scan using compra_consumidor_cpf_produto_id_idx on compra compra_1 (cost=0.42..4.46 rows=2 width=16) (actual time=18.162..18.163 rows=9 loops=1)
32	Index Cond: (consumidor_cpf = '90093928519'::text)
33	Heap Fetches: 0
34	-> Materialize (cost=0.42..8.45 rows=1 width=12) (actual time=0.006..0.006 rows=1 loops=9)
35	-> Index Only Scan using consumidor_pkey on consumidor consumidor_1 (cost=0.42..8.44 rows=1 width=12) (actual time=0.029..0.030 rows=1 loops=1)
36	Index Cond: (cpf = '90093928519'::text)
37	Heap Fetches: 1
38	-> Index Only Scan using compra_pkey on compra (cost=0.42..4.46 rows=2 width=16) (actual time=0.005..0.006 rows=3 loops=9)
39	Index Cond: (produto_id = compra_1.produto_id)
40	Heap Fetches: 0
41	-> Index Scan using consumidor_pkey on consumidor (cost=0.42..0.56 rows=1 width=29) (actual time=0.013..0.013 rows=1 loops=23)
42	Index Cond: ((cpf)::text = (compra.consumidor_cpf)::text)
43	Filter: (((cpf)::text <> '90093928519'::text) AND ((cidade)::text ~~* ($2)::text))
44	Rows Removed by Filter: 0

Figura 5 - Plano de Execução Otimizado da Consulta 2.

5.3. COMPARAÇÃO

Para analisar as consultas, os testes foram realizados nas máquinas dos três autores. A tabela seguinte apresenta as especificações mas máquinas utilizadas.

	Gabriel	**Guilherme**	**Tales**
Versão do PostgreSQL	12.4	12.4	12.4
Processador	Intel(R) Core(TM) i5-4300U CPU @ 1.90GHz	Intel(R) Core(™) i5-3470 CPU @ 3.20GHz	Intel(R) Core(TM) i3-7020U CPU @ 2.30GHZ
Placa de vídeo	Intel Corporation Haswell-ULT Integrated Graphics Controller	RX 570 8GB GDDR5	Intel (R) HD Graphics 620.
Nº de Núcleos	2	2	2
Memória Ram (GB)	8	8	4
HD (GB)	350	1000	1000
Sistema Operacional	Manjaro Linux 20.1.1	Windows 7 Professional x64	Windows Home 10 x64

Tabela 2 - Especificações das Máquinas.

A seguir, temos as tabelas com os testes realizados e resultados encontrados. Os tempos estão em milissegundos (ms).

Consulta 1 Inicial - Gabriel						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	386.886	429.741	383.123	381.420	380.625	**392,359**

Tabela 3 - Execução Inicial da consulta 1 pelo Gabriel.

Consulta 1 Otimizada - Gabriel						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	71.451	71.478	76.927	70.943	65.935	**71.347**

Tabela 4 - Execução otimizada da consulta 1 pelo Gabriel.

Consulta 2 Inicial - Gabriel						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	51.676	57.486	49.140	48.417	51.746	**51.693**

Tabela 5 - Execução inicial da consulta 2 pelo Gabriel.

Consulta 2 Otimizada - Gabriel						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	2.854	4.941	3.918	3.552	2.671	**3.587**

Tabela 6 - Execução otimizada da consulta 2 pelo Gabriel.

Consulta 1 Inicial - Guilherme						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	1833.409	1828.224	3054.023	3124.395	1759.617	**2319,9336**

Tabela 7 - Execução Inicial da consulta 1 pelo Guilherme.

Consulta 1 Otimizada - Guilherme						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	74.125	72.666	70.642	75.544	73.911	**73.3776**

Tabela 8 - Execução otimizada da consulta 1 pelo Guilherme.

Consulta 2 Inicial - Guilherme						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	2094.732	1737.222	2069.013	2049.961	1880.762	**1966.338**

Tabela 9 - Execução inicial da consulta 2 pelo Guilherme.

Consulta 2 Otimizada - Guilherme						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	1.700	0.664	0.675	0.592	0.622	**0.8466**

Tabela 10 - Execução otimizada da consulta 2 pelo Guilherme.

Consulta 1 Inicial - Tales						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	6343.103	5207.491	5160.340	5170.668	4128.296	**5201.979**

Tabela 11 - Execução inicial da consulta 1 pelo Tales.

Consulta 1 Otimizada - Tales						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	499.400	276.956	386.736	459.356	329.858	**390.461**

Tabela 12 - Execução otimizada da consulta 1 pelo Tales.

Consulta 2 Inicial - Tales						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	2136.266	2213.958	2810.246	2134.078	3211.953	**2501.300**

Tabela 13 - Execução inicial da consulta 2 pelo Tales.

Consulta 2 Otimizada - Tales						
	1	**2**	**3**	**4**	**5**	**Média**
Tempo de Execução	1.321	1.565	0.796	0.886	0.778	**1.069**

Tabela 14 - Execução otimizada da consulta 2 pelo Tales.

De forma resumida, as tabelas 15 e 16 mostram a otimização das propostas deste projeto.

Consulta 1			
	Consulta Inicial	**Consulta Otimizada**	**Diferença (%)**
Tempo de Execução (Gabriel)	**392,359**	**71.347**	**81,82%**
Tempo de Execução (Guilherme)	**2319,933**	**73.377**	**96,83%**
Tempo de Execução (Tales)	**5201.979**	**390.461**	**92,49%**

Tabela 15 - Comparação de execução da consulta 1.

Consulta 2			
	Consulta Inicial	**Consulta Otimizada**	**Diferença (%)**
Tempo de Execução (Gabriel)	**51.693**	**3.587**	**93,061%**
Tempo de Execução (Guilherme)	**1966.338**	**0.846**	**99,95%**
Tempo de Execução (Tales)	**2501.300**	**1.069**	**99,957%**

Tabela 16 - Comparação de execução da consulta 2.

O cálculo da diferença entre o desempenho das consultas segue a seguinte fórmula: *((ConsultaInicial-ConsultaOtimizada)/ConsultaInicial) * 100*

Como pode ser analisado detalhadamente pelas Tabelas 3-14 e de forma resumida pelas Tabelas 15 e 16, as propostas de otimização foram bastante eficazes nos três ambientes de testes.

Em relação à primeira consulta, a alteração do esquema do banco de dados, com o objetivo de ter uma coluna para o atributo calculado, possibilitou uma otimização entre 81% e 96% dependendo do ambiente.

Sobre a segunda consulta, o índice composto sobre os atributos `consumidor_cpf` e `produto_id` da tabela `Compra` melhorou o desempenho da consulta entre 93% a 99%.

6. PROGRAMAÇÃO COM BANCO DE DADOS

Consulta 1

A função refere-se à primeira consulta: “Mostrar a média de notas de produtos de uma determinada categoria dentro de uma faixa de preços”. A situação de uso desse tipo de função pode ser imaginada na visão do consumidor, que está em busca de consumir os produtos com maior nota média .Com isso, ela recebe como parâmetros, três valores:

- $1, do tipo texto, que recebe o nome da categoria. Na implementação da função, temos que a busca por esse parâmetro é relativo, por isso se utiliza “categoria ILIKE $1”.
- $2 e $3, são do tipo double e são referentes a faixa de preço do produto, de tal forma que o preço esteja entre $2 e $3. Na implementação da função, temos que a busca por esse parâmetro é absoluto, por isso se utiliza “preco BETWEEN $2 AND $3”.

A saída dessa função é uma tabela, que contém o nome do produto, o preço, descrição do produto, categoria do produto e média de preço.

```
CREATE OR REPLACE FUNCTION media_notas(text, double precision, double
precision)
RETURNS TABLE(name varchar, price double precision, description text,
category varchar, average real) AS $$
BEGIN
      RETURN QUERY SELECT nome_produto, preco, descricao, categoria, media
      FROM produto
      WHERE categoria ILIKE $1
      AND preco BETWEEN $2 AND $3
      AND media IS NOT NULL;
END;
$$ LANGUAGE plpgsql;
```

Código 11 - Primeira *Stored Procedure*

Consulta 2

A função refere-se à segunda consulta: “Recuperar todos os consumidores que possuem um histórico de compras similar a um dado consumidor e que morem na mesma cidade, ordenado por maior nível de similaridade”. A situação de uso desse tipo de função pode ser imaginada em uma possível análise de dados, no qual é possível por exemplo ver os perfis dos usuários

que consomem os mesmos tipos de produtos. Com isso, ela recebe como parâmetros, dois valores:

- $1, do tipo texto, que recebe o cpf do usuário que terá as suas compras comparadas com os demais usuários (Note que o esse parâmetro recebe o cpf como texto, pois em nossa tabela, a coluna cpf é do tipo VARCHAR). Na implementação da função, temos que a busca por esse parâmetro é absoluta, por isso se utiliza “cpf = $1”.
- $2, do tipo texto, que recebe o nome da cidade do consumidor que terá as suas compras comparadas com os demais usuários como parâmetro. Na implementação da função, temos que a busca por esse parâmetro é relativa, por isso se utiliza “cidade ILIKE $2”.

A saída dessa função é uma tabela, que contém o nome do consumidor semelhante, cpf do consumidor semelhante, nome do consumidor procurado, cpf do consumidor procurado, cidade de origem e quantidade de produtos similares.

```
CREATE OR REPLACE FUNCTION consumidores_semelhantes(text,text)
RETURNS TABLE (consumidor_semelhante varchar, cpf_semelhante varchar,
nome_consumidor_procurado varchar, cpf1 varchar, cidade_origem varchar,
qtd_produtos_similares bigint)
AS $$
BEGIN
      RETURN QUERY SELECT nome_consumidor,
      cpf,
      (SELECT nome_consumidor FROM consumidor WHERE cpf=$1),
      (SELECT cpf FROM consumidor WHERE cpf=$1),
      cidade,
      COUNT(cpf) AS qtd_produtos_similares
FROM
      (SELECT DISTINCT nome_consumidor, cpf, produto_id, cidade
      FROM consumidor
      JOIN compra
      ON cpf=consumidor_cpf
      AND cpf <> $1
      AND cidade ILIKE (SELECT DISTINCT cidade FROM consumidor WHERE cidade
ILIKE $2 )
      AND (produto_id) IN
      (SELECT produto_id
      FROM consumidor
      JOIN compra
      ON cpf=consumidor_cpf
      AND cpf=$1)) AS resultado
```

```
GROUP BY nome_consumidor, cpf, cidade
ORDER BY qtd_produtos_similares DESC;
END;
$$ LANGUAGE plpgsql;
```

Código 12 - *Stored Procedure* para a segunda consulta.

7. CONTROLE DE ACESSO DE USUÁRIOS

Um dos aspectos mais importantes para manter a segurança dos dados é definir um controle de acesso de usuários. Assim, o respectivo usuário tem acesso somente aos dados relevantes para o seu uso.

No contexto do *e-commerce* e levando em conta o tema “Avaliação de produtos por consumidores”, os seguintes usuários foram considerados.

- Administrador: usuário dono do banco de dados, é o usuário que tem mais controle sobre o sistema.
- Gerente: usuário responsável por supervisionar o usuário SAC-BI-Marketing, por isso tem mais autorizações no sistema.
- Desenvolvedor: usuário responsável por fazer a manutenção da página da loja, ou seja, entre as suas atribuições estão: cadastrar novos produtos, atualizar os preços e remover produtos que não são mais vendidos.
- SAC-BI-Marketing: usuário padrão que tem autorização básica (somente leitura) nas tabelas do banco de dados.
- Consumidor: usuário que realiza compras no *e-commerce*.

O grafo da Figura X mostra as concessões realizadas no sistema. Como pode ser analisado, o usuário `Administrador` concede para os demais usuários os privilégios adequados para cada situação.

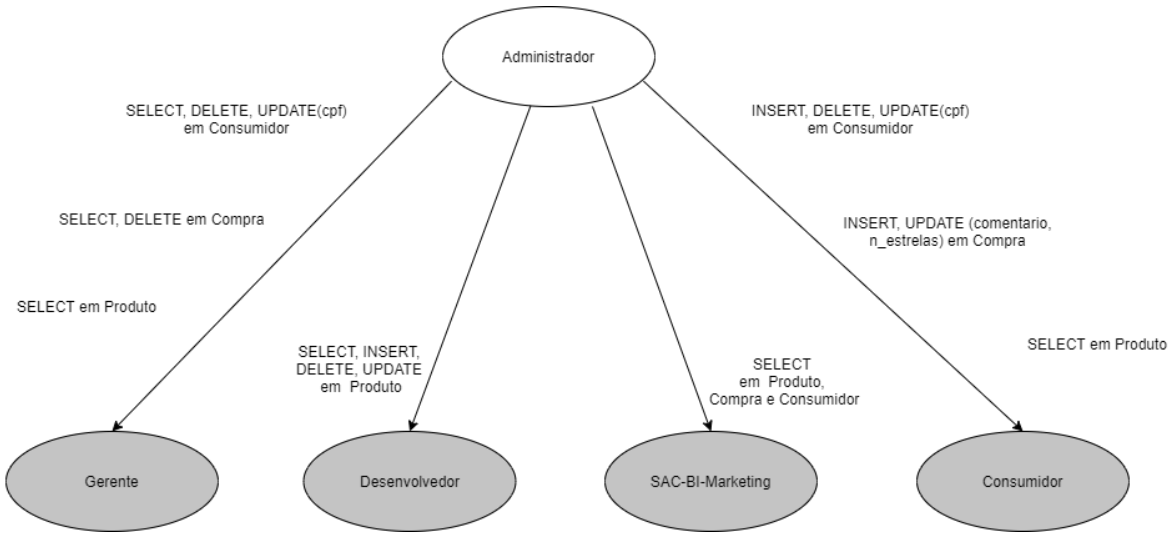

Figura 6 - Grafo de concessões do sistema.

A Tabela 17 mostra os privilégios que os usuários têm sobre cada tabela.

Usuário	**Consumidor**	**Compra**	**Produto**
Administrador(owner)	S,I,D,U	S,I,D,U	S,I,D,U
Gerente	S,D,U(`cpf`)	S,D	S
Desenvolvedor	-	-	S,I,D,U
SAC-BI-Marketing	S	S	S
Consumidor	I,D,U	S,I,U(`comentario`, `n_estrelas`)	S

Tabela 17 - Perfis de usuários e seus privilégios.

* Abreviações: SELECT (S), INSERT (I), DELETE (D) e UPDATE (U).

Cada tópico a seguir mostra as permissões de cada usuário.

- **Administrador**: o usuário Administrador tem autorização de leitura (S), inserção (I), remoção (D) e atualização (U) sobre qualquer atributo de todas as tabelas, porque é o usuário responsável pelo sistema.

- **Gerente**: o gerente é responsável por realizar operações que são mais sensíveis em relação ao SAC, por exemplo, realizar a alteração do cpf de um usuário, cancelamento de uma conta e cancelamento de uma compra. Portanto, possui permissão de leitura (S) para todas as tabelas, permissão de remoção (D) para tabela Consumidor e Compra e permissão de atualização (U) para a tabela Consumidor.
- **Desenvolvedor:** o usuário Desenvolvedor tem a permissão de leitura (S), inserção (I), remoção (D) e atualização (U) para a tabela Produto, pois é o funcionário responsável por administrar o site.
- **SAC-BI-Marketing:** o usuário SAC-BI-Marketing tem apenas a autorização de leitura (S) para todas as tabelas. Esse usuário se refere a funcionários que utilizam o BD apenas para visualização, utilizando informações referentes ao consumidor, compra e produto para informar quem necessita(SAC) ou seja para ter acessos de forma que seja possível fazer análises e tomar decisões estratégicas (BI-Marketing).
- **Consumidor:** o usuário Consumidor tem a autorização de inserção (I), remoção (D) e atualização (U) para a tabela Consumidor, pois se trata dos dados do próprio consumidor, no qual ele poderia se cadastrar, atualizar dados cadastrais ou excluir a própria conta. Para a tabela Compra, o usuário Consumidor tem a autorização de inserção (I), leitura(S) e atualização(U), pois um Consumidor pode realizar diversas compras, visualizar compras que ele já fez e atualizar comentários/notas. Para a tabela Produto o usuário possui apenas a autorização de leitura (S), pois é a forma que ele visualizará os produtos.

A Figura 7 mostra como seria a conexão entre uma aplicação e o banco de dados proposto. Como pode ser observado, para cada ação da aplicação, tem um conjunto de comandos realizados por determinados usuários do banco de dados.

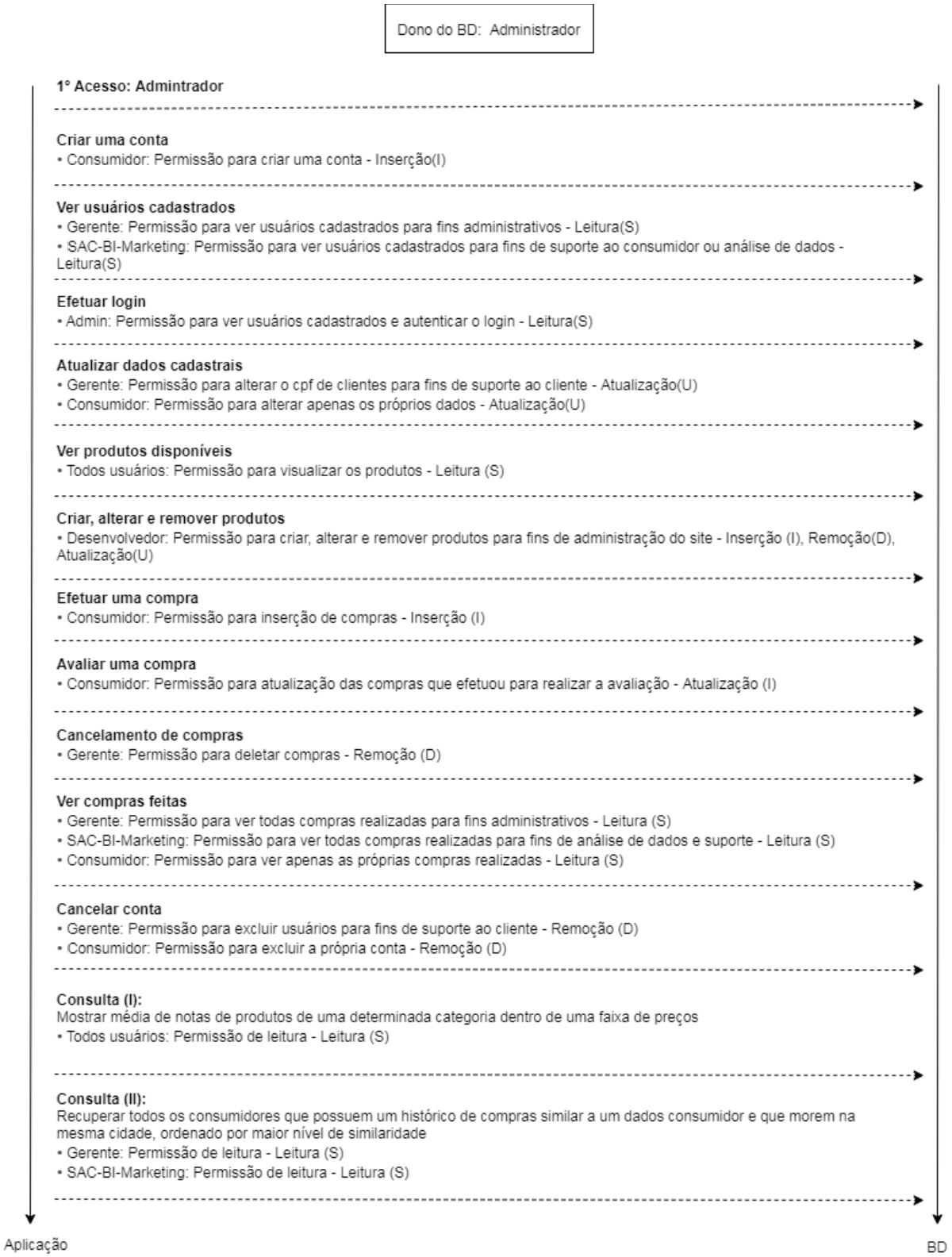

Figura 7 - Conexão entre aplicação e BD.

8. CONSIDERAÇÕES FINAIS

A experiência no projeto foi enriquecedora, principalmente em relação a preocupação com a qualidade e o tempo de execução das consultas, pois, principalmente em projetos de larga escala, isso é essencial para o desempenho e que nem sempre é tomado como prioridade pelos desenvolvedores.

Em relação a SQL e o PostgreSQL, os principais pontos de dificuldades no início foram as *transactions*, logo na parte de criação dos scripts do banco de dados nos deparamos com um problema de que toda vez que uma inserção dava errado ele ignorava o resultado do script como um todo (justamente por conta das transactions) e para corrigir o script, o funcionamento das transactions ficaram mais claras.

Outro ponto de dificuldade foi interpretar o plano de execução, já que há diversas opções de caminhos para o otimizador tomar durante as consultas, especialmente na utilização dos índices e a situação se agrava quando a consulta é um pouco mais complexa.

O principal aprendizado em relação a isso foi a necessidade de entender a consulta em um alto nível, procurando os pedaços de instruções que possuem operações mais custosas e utilizar o plano de execução como um complemento das conclusões em alto nível. Os roteiros que eram passados nas tarefas também foram de grande auxílio no projeto, pois seguindo os roteiros era possível entender os detalhes que no primeiro momento não ficaram tão claros.

Sobre o resultado do projeto como um todo, gostamos do que atingimos principalmente acerca do processo de otimização, que permitiu encontrar não só consultas SQL mais eficientes mas também detalhes na própria definição das tabelas que em primeiro momento não são percebidos, como por exemplo, o funcionamento do cálculo da média de avaliações dos produtos que anteriormente era necessário recalcular e no final, a decisão foi de manter a média atualizada na tabela.

ANEXO 1

Consumidor							
cpf	**nome_ consumidor**	**data_nasc**	**telefone**	**email**	**senha**	**endereco**	**cidade**
12345678911	Tiago	19/12/1990	976797009	tiago @gmail.com	1234	Rua 1, 33	São Paulo
12345678912	Sara	26/06/1999	996847895	sara @gmail.com	senha1234	Rua 21, 42	São Paulo
12345678913	Mario	24/05/1986	983676887	mario @gmail.com	12345	Rua 3, 767	São Paulo
12345678914	Fernando	27/03/1974	974586524	fernando @gmail.com	9456	Rua 4, 42	Sorocaba
12345678915	Pablo	23/06/1992	985657152	pablo @gmail.com	passwd456 7	Rua 9, 150	Sorocaba

Tabela 17 - Tabela de Consumidores.

Produto				
id	**nome_produto**	**preco**	**descricao**	**categoria**
0	Camiseta	36	Camiseta amarela	Vestimenta
1	Calça	158	Calça jeans	Vestimenta
2	Cadeira	35	Cadeira de sala de estar	Mobília
3	Sofá	410	Sofá para 3 pessoas	Mobília
4	Meia	15	Meia Branca	Vestimenta

Tabela 18 - Tabela de Produtos.

Compra						
produto_id	**consumidor_id**	**data**	**quantidade**	**entregue**	**n_estrelas**	**comentario**
0	12345678911	27/03/2020	2	TRUE	4	NULL
0	12345678912	14/05/2019	3	TRUE	5	NULL
1	12345678912	14/05/2019	5	TRUE	5	NULL
1	12345678911	27/03/2020	3	TRUE	5	NULL
3	12345678915	14/04/2020	1	TRUE	3	NULL

Tabela 19 - Tabela de Compras.

Consulta					
nome_consumidor	**cpf**	**nome_ consumidor**	**cpf**	**cidade**	**qtd_produtos_similares**
Tiago	12345678911	Sara	12345678912	São Paulo	2

Tabela 20 - Resultado da Consulta.

ANEXO 2

Como DDL, tivemos como resultado o seguinte código, já considerando as alterações vindas da otimização

```
CREATE TABLE consumidor(
      cpf varchar(11) PRIMARY KEY,
      nome_consumidor varchar(200) NOT NULL,
      data_nasc DATE NOT NULL,
      telefone varchar(16) NOT NULL,
      email text UNIQUE NOT NULL,
      senha text NOT NULL,
      endereco text NOT NULL,
      cidade varchar(32) NOT NULL
);
CREATE TABLE produto(
      id serial PRIMARY KEY,
      nome_produto varchar(200) NOT NULL,
      preco float NOT NULL,
      descricao text NOT NULL,
      categoria varchar(100) NOT NULL,
            media real,
            vendidos int
);
CREATE TABLE compra(
      produto_id int NOT NULL,
      consumidor_cpf varchar(11) NOT NULL,
      data TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
      quantidade int NOT NULL,
      entregue boolean DEFAULT FALSE,
      n_estrelas float CHECK(n_estrelas BETWEEN 0 AND 5) DEFAULT
NULL,
      comentario text DEFAULT NULL,
      PRIMARY KEY(produto_id, consumidor_cpf, data),
      FOREIGN KEY(produto_id) REFERENCES produto(id),
```

```
    FOREIGN KEY(consumidor_cpf) REFERENCES consumidor(cpf)
);
```

Código 15 - DDL

ANEXO 3

```
-- Criação dos usuários
-- Administrador

CREATE ROLE administrador
LOGIN
NOINHERIT
CREATEDB
CREATEROLE;

-- Gerente
CREATE ROLE gerente
NOINHERIT;

-- Desenvolvedor
CREATE ROLE desenvolvedor
LOGIN
NOINHERIT;

-- SAC_BI_Marketing
CREATE ROLE sac_bi_marketing
NOINHERIT;

-- Consumidor
CREATE ROLE consumidor
NOINHERIT;

-- Mudança de dono das tabelas. Necessário, porque o backup tinha o
usuário postgres como dono.
ALTER TABLE compra OWNER TO administrador;
ALTER TABLE consumidor OWNER TO administrador;
ALTER TABLE produto OWNER TO administrador;
```

Código 16 - Criação de usuários.

```sql
-- Troca de acesso no BD para o administrador.
SET ROLE administrador;

-- Gerente
GRANT SELECT, DELETE, UPDATE(cpf)
ON consumidor
TO gerente;
GRANT SELECT, DELETE
ON compra
TO gerente;

GRANT SELECT
ON produto
TO gerente;

-- Desenvolvedor
GRANT SELECT, INSERT, DELETE, UPDATE
ON produto
TO desenvolvedor;

-- SAC-BI-Marketing
GRANT SELECT
ON consumidor
TO sac_bi_marketing;

GRANT SELECT
ON compra
TO sac_bi_marketing;

GRANT SELECT
ON produto
TO sac_bi_marketing;

-- Consumidor
GRANT INSERT, DELETE, UPDATE
ON consumidor
```

```
TO consumidor;

GRANT SELECT, INSERT, UPDATE(comentario, n_estrelas)
ON compra
TO consumidor;

GRANT SELECT
ON produto
TO consumidor;
```

Código 17 - Concessão de privilégios.